Evolution and Adaptation of Web Services

by

Anand Iyer

Graduate Diploma in IT (QUT, Australia)

Thesis submitted in accordance with the regulations for Master of Information Technology (Research)

Programming Languages And Systems Research Centre
Faculty of Information Technology
Queensland University of Technology

April, 2003
Keywords

Microsoft .Net, Web Services, XML/XSLT, SOAP, Adaptation
Abstract

One of the main aims of Component adaptation [Szy97] is to help application developers reuse components so that they can plug-in third party components into their application. This research concentrates on this type of adaptation but in the context of Web Services. Web Services are becoming increasingly popular. Web Services often fit the requirements of being a component, and can be reused in a very similar manner. Hence there is a requirement for adaptation of Web Services just as there is the need for adaptation of software components. There are now quite a few adaptation techniques, but few of them have identified adaptation techniques for Web Services. This approach to adaptation allows for the modification of data and behaviour of existing Web Services. The approach to adaptation uses eXtensible Stylesheet Language (XSL) transformation applied to the message passed between Web Services. These messages are commonly in XML format, hence XSL can be used to modify them. The application of the transformation is guided by a specification written in XML. The adaptation is executed by a generic runtime system that uses these specifications which are referred to as Guiding Specifications. This has been demonstrated by way of a motivating real world example implemented on the .Net platform.

It is shown how an adapter can be specified using a simplistic Guiding Specification and related XSLT documents. This allows the implementation to work more efficiently than hand coding each adapter. It is the underlying generic runtime support that provides much of this benefit.

Component based software engineering (CBSE) constructs applications by assembling components together, CBSE has been of great help to application developers due to the very fact that tailor made components can be purchased from third party vendors and can be plugged-in to a system to form a working application. But in practice ‘as-is’ reuse is very unlikely to occur, and most components need to be changed in some way to match the requirements of the
application architecture and other components. The process of changing the component for use in a particular application is often referred to as Component Adaptation.
# Contents

## Keywords

Abstract

Declaration

Acknowledgements

1 Introduction
   1.1 XSLT based adaptation
   1.2 Investigations done for Web Service adaptation
   1.3 Outline

2 Foundations
   2.1 Introduction
   2.2 Component adaptation
      Requirements of Component Adaptation
      Adaptation Techniques
   2.3 Web Services
      Different scenarios where Web Service Adaptation would arise
      Creating a Web Service
   2.4 XML/XSLT
   2.5 Simple Object Access Protocol (SOAP)
   2.6 EDI
      Technical Details of EDI
      Transformation of EDI & XML documents
      Use of EDI in XSLT based adaptation
## 3 Preliminary Results

3.1 Adaptation using SOAP Extensions

3.2 Example Scenarios

  - Scenario 1: Data Manipulation
  - Scenario 2: Changing of Method Names
  - Scenario 3: Refactoring

3.3 Adaptation using Broker

## 4 Architecture & Design

4.1 High Level Design

4.2 Guiding Specification

4.3 Detailed Design

## 5 Example of broker based Adaptation

5.1 Introduction to the Online Share Trading System

5.2 Motivation for this demonstrator example

  - Solution

5.3 Adapter Implementation Details

## 6 Discussion of XSL based adaptation

6.1 Why Web Services & SOAP?

6.2 Evaluation

  - Comparison with BizTalk

6.3 Composition of Web Services

6.4 Composition of Web Services using multiple brokers

6.5 Asynchronous calls for accessing multiple services

## 7 Further work & conclusion

7.1 Future work

7.2 Conclusion

## A Example XSLT scripts

Bibliography
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Overview of the structure of the XSLT based adaptation</td>
<td>3</td>
</tr>
<tr>
<td>2.1</td>
<td>Wrapping a Component</td>
<td>13</td>
</tr>
<tr>
<td>2.2</td>
<td>Use of XML and XSLT in different platforms</td>
<td>24</td>
</tr>
<tr>
<td>2.3</td>
<td>Communication of EDI under the X12 Standard</td>
<td>29</td>
</tr>
<tr>
<td>3.1</td>
<td>Generic high level adaptation architecture</td>
<td>40</td>
</tr>
<tr>
<td>3.2</td>
<td>SOAP message stages using SOAP extensions on the Client Side</td>
<td>41</td>
</tr>
<tr>
<td>3.3</td>
<td>Adapting to different parameter types &amp; number of arguments</td>
<td>44</td>
</tr>
<tr>
<td>3.4</td>
<td>Mapping one-to-many calls</td>
<td>48</td>
</tr>
<tr>
<td>4.1</td>
<td>Guiding Specification for XSLT based adaptation</td>
<td>53</td>
</tr>
<tr>
<td>4.2</td>
<td>High Level Design of the Broker</td>
<td>55</td>
</tr>
<tr>
<td>4.3</td>
<td>Detailed Design of the Broker</td>
<td>57</td>
</tr>
<tr>
<td>4.4</td>
<td>Server Side Adaptation to access multiple services</td>
<td>58</td>
</tr>
<tr>
<td>5.1</td>
<td>Structure of the Guiding Specification</td>
<td>69</td>
</tr>
<tr>
<td>6.1</td>
<td>Difference showing the adaptation done on SOAP envelope</td>
<td>79</td>
</tr>
<tr>
<td>6.2</td>
<td>Four Web Service methods satisfying one client request</td>
<td>84</td>
</tr>
<tr>
<td>6.3</td>
<td>Adapting to a Web Service with more than one broker</td>
<td>86</td>
</tr>
</tbody>
</table>
List of Tables

2.1 X12 and EDIFACT message formats ................ 28
Declaration

The work contained in this thesis has not been previously submitted for a degree or diploma at any higher education institution. To the best of my knowledge and belief, the thesis contains no material previously published or written by another person except where due reference is made.

Signed: .................................. Date: ..........................

xiii
Acknowledgements

During the course of my Masters Thesis I was constantly motivated by my mother Lakshmy Viswanath without whom it would been impossible for me to complete this. Thank you very much Mother for your constant appreciation. I would like to thank Dr Glenn Smith and Dr Paul Roe for the cooperation and help they gave me to complete my Masters and also thank them for supervising my Masters candidature. I also acknowledge Greg Neverov and Joel Pobar who gave me some valid inputs during my candidature. This research has been supported by Mincom and Programming Languages and Systems Research Center. The financial support has been greatly appreciated.
Chapter 1

Introduction

As the size of software applications increase, it not only becomes infeasible to implement software systems entirely from scratch, it also becomes more difficult to maintain them. Component based software engineering (CBSE) constructs applications by assembling existing software components [GSS98]. CBSE benefits application developers as tailor made components can be purchased from third party vendors and plugged-in to a system to build a working application. However, in practice, ‘as-is’ reuse is very unlikely to occur, and most components need to be changed in some way to match the requirements of the application architecture and other components [Bos99]. However components can be adapted to allow application developers to reuse more components. The process of changing the component for use in a particular application is often referred to as Component Adaptation [Szy97]. Component based software provides the ability to meet the changing requirements of software by composing existing components to make software development easier, faster and less expensive. This allows software engineers to concentrate on value-added tasks and produce high-quality software within a short period of time. Component Adaptation provides software engineers with a wider choice of software components to use.

We use adapters in our everyday life; for example in the computer hardware a graphics adapter is used to isolate a monitor from the mainboard that drives it. The monitor’s interface does not make assumptions specific to a particular type of mainboard. Instead, all the architecture-specific properties are placed in the adapter. Thus it becomes possible to connect a monitor to many different
computer architectures, all that has to be done is to use the appropriate adapter.

In this research we investigate Component Adaptation and how it can be applied to the domain of Web Services. Software Components and Web Services are related in that both communicate via well defined interfaces. Adaptation described in this thesis will focus on the modification of messages passed between Web Services. The protocol investigated is the Simple Object Access Protocol (SOAP) [W3C00a], an XML based protocol for Web Services. The modification of the XML message will be achieved by using the XML stylesheet language transformation (XSLT).

1.1 XSLT based adaptation

This thesis describes a mechanism for Web Services adaptation using XSLT. XSLT is used for transformation of XML documents. XSLT based adaptation focuses on Web Service adaptation across heterogeneous environments where SOAP [W3C00a] is used. It is an XML based protocol which consists of three parts: a SOAP envelope which describes what is in a message, a set of encoding rules and convention for representing remote procedure calls and responses. The SOAP messages are transformed using XSLT. This is applicable to SOAP because SOAP envelopes are embedded on XML, and XSLT transformations can be applied on these XML documents. Other details of SOAP are described in Chapter 2 and section 2.5 of this thesis.

Web Services are becoming more and more popular. In the case of Web Services the interfaces of service provided needs to be matched with all the clients that require its service. Web Services describe these interfaces using the Web Service Description Language (WSDL). It is not feasible for the service provider to create a Web Service interface for each and every possible client requirement. This is the basic motivation behind the construction of adapters for Web Services. Web Service interfaces are described in Chapter 2 of this thesis.

Figure 1.1 illustrates the overall structure of the XSLT based adaptation mechanism described in this thesis. The diagram shows an example of adaptation between a client and a server. The adaptation layer intercepts requests from the client, and applies a Guiding Specification which determines which XSLT document from the library should be applied. The Guiding Specification and the XSLT library form the core of the adapter, as the Guiding Specifications contain
the required information about the requirements of the client and the services provided by the Web Service to apply an appropriate mapping. It is assumed that the client and server code have already been deployed, that they are fixed and cannot be changed. Adaptation seeks to provide a wrapper to cope with minor impedance mismatch. Where there is a wide discrepancy between the service provided by the server and expected by the client adaptation will not be considered.

Figure 1.1: Overview of the structure of the XSLT based adaptation

The Adaptation Layer in the Figure 1.1 acts as a intermediate proxy to overcome differences between the services provided by the server and the client requirements. The Guiding Specification used by the Adaptation Layer refers to a list of XSLT documents. The Adaptation Layer is responsible for receiving requests and responses from the client and server. The XSLT documents referenced by the Guiding Specification are responsible for transforming these requests and responses using the Guiding Specification. The details of this process are explained in Chapter 4 of this thesis.

Further detail of the adaptation process is provided by way of a demonstrative example that is described in Chapter 5 of this thesis.
1.2 Investigations done for Web Service adaptation

Preliminary work for this thesis surveyed the area of component adaptation parallel with Web Services. It was found that Web Services were very new to the industry and there were few adaptation techniques available for Web Services. Thus adaptation for Web Services was proposed. The surveyed literature mainly concentrated the underlying technologies of Web Services which includes XML, SOAP, and UDDI. Microsoft .Net had released the beta version which supported these technologies, and as .Net was a new technology there was more scope for research in this area.

Adaptation was the focus of this research, with the belief that there should be some means of adapting to the server’s interface by the client. As the messages transmitted using SOAP are in XML format, transforming this XML embedded SOAP packet could serve the purpose of adaptation. XQuery and XSLT are the two languages that are used to transform XML documents. As Xquery was not standardised it was proposed that XSLT would be used for transformation of XML documents.

Initially investigations were carried out on simple scenarios where XSLT was applied to SOAP packets and forwarded to the server. In these scenarios, adaptation was applied using SOAP extensions [Bal01], which worked according to our requirements but when complex examples were taken into consideration it was found that adaptation using SOAP extensions encountered technical difficulties. These are explained in Chapter 3. Adaptation using a broker was then considered, and an adapter was designed for the server side of communication. A prototype implementation was developed and used to demonstrate the concepts of XSLT based adaptation. XSLT based adaptation was tested by applying it to the real-world example (online share trading system). The experience gained from applying the example was used to provide insight into the success and difficulties of XSLT based adaptation, and to develop generic facilities to support the production of required adapters.
1.3 Outline

This thesis is divided into 7 Chapters. Chapter 1 describes the introduction. Chapter 2 describes the introduction to component adaptation and Web Services. It compares software components with Web Services and the motivation for choosing Web Services as components for adaptation. The next section of Chapter 2 involves adaptation techniques and the basic requirements of Component Adaptation. Comparisons with XSLT based adaptation are investigated in the later chapters. The later sections of Chapter 2 describe the background technologies used in this research. It introduces to the reader with the basic knowledge of Web Service, including technical details of how a simple Web Service is created using .Net, XML/XSLT and SOAP. Sample XSLT scripts are presented along with SOAP packets, and briefly explained. The related work on adaptation is also discussed in Chapter 2 which describes BizTalk, EDI and other adaptation instances. Chapter 3 describes the preliminary results of adaptation on Web Services. It demonstrates the initial investigations carried out using SOAP extensions with sample scenarios. Chapter 4 describes the high level and the detailed design of the adapter. For this purpose a Guiding Specification language together with the XSLT libraries was developed. Chapter 5 describes the demonstrator example, an online share trading system. This example uses the Guiding Specification and the design discussed in Chapter 4. This also has some sample SOAP requests/responses and sample XSLT scripts. It explains how the data travels from the client’s machine to the adapter and finally to the Web Service and vice versa. It also explains how the data will look in each of these stages. Chapter 6 evaluates this adaptation technique. The requirements of component adaptation are compared with the adaptation technique and evaluated against a certain criteria. Reasons for choosing SOAP for adaptation and XSLT for transformation of messages are also given. The applications of this adapter, like composition and synchronized calls to the Web Services, are summarised in the later sections of this chapter. Chapter 7 concludes this thesis and outlines some open issues. The appendix has some example XSLT scripts described in Chapter 5.

This chapter provides a basic introduction to XSLT based adaptation and a briefing of future chapters to follow. The next chapter will examine some of the important technologies relating to this thesis that form the foundations of this research.
Chapter 2

Foundations

In this chapter the background technologies that are important to the context of this thesis will be examined. The first part of this chapter will be the literature review of the thesis which will include introduction to software components, component adaptation, Web Services and some of the distributed systems like RMI, CORBA. The later sections of this chapter will demonstrate a general rationale and also some related work to this thesis. In related work, our discussion will concentrate on EDI, BizTalk and an evaluation of BizTalk with our adaptation technique.

2.1 Introduction

There are several definitions of a component, the definition by Szyperski will be used here which is as follows:

“A Software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third party”, [Szy97].

Szyperski further describes the characteristic properties of components; A component is a unit of independent deployment; a component is a unit of third party composition; a component has no persistent state. In order to explain these characteristics we can imagine a large heterogeneous system created with separate parts implemented on different platforms, each part constitutes a component.
Each component need not know the internal working of other components. Each component interacts with other components through an interface. This interface has clear specification of the component, what it requires and provides. If a component has a persistent state then multiple installations of the same component could be distinguished. If components were allowed to have state, then two installations could have different properties. This causes a problem when a new version of the component is released. Each upgraded component may require modification.

A key aspect for the success of components is that there exist standards to enable interoperation. These standards define the interfaces and overall architecture. An interface specification agrees on the pre and the post conditions and also the behavior of the component. With these specifications, combinations of number of components that interact with each other constitute a whole system. Thus it is not necessary for an application builder to build a system entirely from scratch but rather construct it from existing components.

2.2 Component adaptation

To use a component it is obvious that the components need to be connected with each other. It is also obvious that such connections need to follow standards to make it at all likely that any two components have compatible ‘connectors’ [Szy97]. For example the standard of power outlets and plugs for India and Australia are completely different, so any electrical appliance brought from Australia cannot be used in India with the same plug. Thus a bridging technology is essential to connect this electrical appliance, and ‘adapters’ can solve this problem to connect an Australian electrical appliance in an Indian power outlet. In the same way, in Component-Oriented Programming it becomes necessary to use adapters. In Component-Oriented technology there might exist a third party vendor who produces a software component similar to that an application developer requires. But it is unlikely this component provides an interface that is exactly what the system requires as it is difficult to reach complete agreement on standards, in fact more than one standard could exist. In order to integrate this third party component into the original system, the application developer needs to adapt this component to match the requirements of the system; this task might be complicated due to the fact that there may be syntactic or interface incom-
patibilities, as they need to communicate with each other. Another example is multiple vendors who produce similar software components. In order for them to be in the market and have a competitive edge they can provide some type of adaptation to allow access by a greater number of clients. For this to happen the third party vendor might apply mechanisms for adaptation without revealing the full details of design and implementation as the application developer would have no knowledge of the internal implementation of the component. An adapted component can be used as if it were a direct implementation. These examples motivate the need for adaptation.

There are quite a few Component Adaptation instances available in the industry. In this section some of them are discussed in brief.

1. **Superimposition** [Bos99]: Superimposition allows the application developer to adapt a component using a number of predefined adaptation behaviors that can be configured for the specific component. Since one may identify new types of adaptation behavior, the application developer can define new adaptation types. Finally, the application developer may compose multiple adaptation behavior types for a single component.

2. **Binary Component Adaptation (BCA)** [RK97]: Black-box adaptation as no source code required. Here the component can be adapted from the binary class file. BCA allows adaptation to be deferred until the load time of the component. Using BCA, application developers add new code to the component and can change attributes like method parameters. BCA works something like this: All the information regarding the adaptation of the component, new code to be added to the component, are included in a separate file. This file is then compiled and used by the class loader to perform the adaptation. This compiled file is called ‘delta’ file because it contains the differences between the standard class file and the application-specific variant. At load time the byte code is directly modified by the class loader allowing on-the-fly adaptation. Problems include:
   
   - This component adaptation technique has been implemented for Java, thus there are some problems when this is applied to other high-level languages (i.e. Language-dependent).
   - Binary code needs to have enough high level information.
3. **Dynamic Component Adaptation** [KUM97]: In component systems, finite number of components may be loaded dynamically to running programs. In Dynamic Component Adaptation [KUM97] the adapters also have to be provided dynamically. Adapters have to be provided by developers, however this adaptation can be performed either on the component or on the interface of the component. But this adapter must be able to be installed and removed on runtime; this means that the developer should not be bothered with the details of installing an appropriate adapter between two components. Thus Dynamic Component Adaptation is changing the components’ functionality at runtime. Components participating in a dynamic adaptation process must explicitly declare to which interfaces they adhere. On demand a component can support all the interfaces for which proper adapters exist. For this Dynamic Component Adaptation defines two algorithms, Converter and Binder. The converter converts a given component to particular interface by matching the given component/interface pair to an adapter on demand and the binder binds a given interface to an implementation instance which is either a component or a component wrapped by adapters.

4. **Conciliation** [Smi99]: Conciliation focuses on component adaptation across component environment boundaries. In the present arena components originate from different object worlds. In order for these objects to correspond with each other some mechanism is required for adapting the interfaces of one component to that of another. The term Conciliation is used to describe such a process and the term Conciliator to refer to the required mechanisms. In Conciliation each component environment requires the provision of a conciliation runtime support. The runtime support uses specifications to guide the conciliation process and utilizes cross environment bridging technologies [GSS98]. The specifications would mean mapping of the required functionality onto the available components. These specifications define the production and behaviour of conciliators. In summary it can be said that in a world of overlapping component standards, Conciliation addresses a semi-automatic approach of adaptation identifying structural as well as behavioural mismatch.
2.2. Component adaptation

Requirements of Component Adaptation

Components and adaptation to these components have been discussed. In this context it would be worthwhile to examine the basic requirements for adaptation. These requirements will be used to evaluate the adaptation technique in later chapters and examine whether these requirements have been met:

1. **Transparent** [Bos99]: The Component Adaptation type is to be transparent to both the client who is requiring the component and also to the components, which are adapted. They should be totally unaware of the fact that some type of adaptation mechanism is between them.

2. **Black Box** [Bos99]: The adaptation technique should be such that it need not have any knowledge of the component’s internal specification.

3. **Platform independent**: For a given component the adaptation mechanism should not depend on any particular platform.

4. **Language independent**: For a given component the adaptation mechanism should not depend on any particular language.

5. **Composable** [Bos99]: The adaptation mechanism should be composable with the component. The component should be compatible with other adaptations also, and the components should behave the same way as before the adaptations were applied to them. There should be no change in the original component because of this adaptation and also it should support multiple adaptations for that component.

Adaptation Techniques

One of the main focuses for Component Based Software Engineering (CBSE) is to increase the productivity and quality of software development by reusing the software components rather than re-inventing the wheel each time [Ohl99]. And the aim for Component Adaptation is to make it easier for application developers to reuse these software components and help integrate them.

In the past most of the applications were large monolithic applications. One of the main goals of CBSE is to create small assemblies of components so that application developers can concentrate more on value-added tasks, business logistics, quality and also reliability of the final system. Some of the important goals of Component Adaptation can be summarised below:
- Enable efficient and systematic reuse of software components.
- Bring greater quality and dependability.
- Take less time and also test the new systems that may constitute different components composed using multiple adapters.
- Provide easy adaptation of the final system with other off-the-shelf components efficiently.
- Provide cost-cutting options to the organisation using tailor-made components.

Having seen the main goals of adaptation, how these goals can be met using different adaptation techniques is examined. Some scenarios to determine whether a particular technique is feasible or not are also examined. The two main adaptation approaches are black-box and white-box adaptation. In the case of black-box adaptation the application developer requires very little or no knowledge of the original component, and in case of white-box adaptation the application developer requires more knowledge of the original component. The minimum information required to reuse a software component is its interface specification [Smi99].

1. Wrapping: This is a black-box adaptation technique. The developer adapting the component needs little or no knowledge of the component. He/she wraps the component into a new component and thus adapts it according to the needs of the application. In the case of wrapping, the component cannot be accessed directly as it acts like a filter to the original component. The application passes all the messages to the adapted component and this in turn passes to the original component and vice versa. Wrapping is considered to be very easy as it needs only the interface specification of the original component. Problems that may be encountered include:

   - More coding work to be done by the application developer adapting the original component, as all of the functionalities need to be handled by the wrapper.
   - All the elements need to be wrapped irrespective of the need for adaptation.
2.2. Component adaptation

- All the messages need to be passed through the wrapper, this means that a client which wants to access the component directly cannot do so, since the wrapper completely encapsulates the adapted component.
- The wrapper needs to handle all the messages sent to the component including those that do not need to be adapted.

![Diagram of wrapping a component](image)

Figure 2.1: Wrapping a Component

2. **Proxy** [Ohl99]: This is a black-box adaptation technique. In this type of adaptation technique there exists a proxy component between the original component and the application thus making both interfaces (original component and application) compatible with each other. Like wrapping, this technique is not transparent as the proxy acts as a filter and the application has to communicate with the proxy. The main advantage of proxy over the wrapper is that the adapted component (proxy) is not embedded on to the original component, so, if need be, the original component can be directly accessed by the application.

3. **Inheritance**: Object-Oriented languages like C++, Java, C# support Inheritance. In this technique the adapted component acts as a derived class of the original component which is the super class. As in this case the developer requires enough knowledge of internals of the original component this technique can be described as white-box adaptation. The main problem in using Inheritance for adaptation is that the developer needs to know the internals of the original component and needs a detailed knowledge of the super class methods. Problems in using this technique include:

- Internal specifications of the original component are required by the application developer.
• Not composable in the sense that it is difficult to associate multiple adaptations with a class.

2.3 Web Services

The evolution of the modern computer started from PC and DOS based systems, then came the GUI based applications and the WWW. The early days of the WWW include static HTML pages and standards like HTTP. When the WWW became more mature there were more dynamic pages, user responses, client side and server side scripts. Web usage has grown and has become a key place for information retrieval and business activity. In the initial stages these were more of information gathering and information provision, which has now become a place for trading and e-commerce. However, users expect something more than trading, retrieving and providing information. For example, a user logs on to amazon.com to buy a book which is not found on the site, then amazon should have a service where it can contact its suppliers and check for the stock and provide the information about when it is available. Businesses should be able to gather data, integrate their processes, join forces to offer customised comprehensive solutions to their customers. The user must be able to acquire information irrespective of his location, platform or computing device. This vision is yet to be achieved. In most cases today the user manually navigates between web sites, application, and rarely can a user carry data between web sites. Tasks such as arranging an appointment with a doctor and automatically updating the patient’s diary are yet to be seen in common use.

As a result of the change in how business and personal consumers use the web, there exists a need for a new computing model that enables a standard way of building applications and processes to connect and exchange information over the web. Solving these problems would be the key challenge for the next generation of the Internet. Web Services try to solve these types of problems and enable applications, machines, and business processes to work together in a revolutionary way.

The Component Object Model (COM), CORBA/IIOP, Enterprise Java Beans (EJB) object models are tools that allow creation of reusable software components. It would be time consuming and prove expensive to integrate programs across these technologies. CORBA in particular was designed specifically for
2.3. Web Services

cross-platform integration. The benefit that Web Services offers is the use of HTTP and XML, therefore not requiring a specific middleware library at each end. Web Services address this problem through HTTP, a standard Internet protocol supported by almost all the platforms, and XML, a platform independent and robust definition vocabulary.

RMI and DCOM have been around for some time and support communication over the Internet but all these depend on an underlying object model. Web Services use HTTP and XML for communication over the web. HTTP and XML are widely accepted standards that most hardware platforms fully support. A Web Service can be written on any platform, and modules running on different platforms can call it. SOAP serves to define a specific standard that Web Service requests and responses must adhere to.

The examples created and investigated in this research are using the .NET framework. Thus this section of Web Services concentrated more on XML Web Services and .NET related technologies. Microsoft’s manifestation of the Web Services concept is the .NET Framework [Cor01c]. By using .NET technologies and practices, developers can build solutions that combine services from all over the Internet. .NET services can run on any platform that the framework supports. .NET supports C#, VB.Net, C++ and these programs will be compiled to a platform independent format called Intermediate Language (IL). Microsoft has developed its own compiler for .Net for languages such as C#, VB, C++; the compiled result is called MSIL (Microsoft Intermediate Language) however there are several organisations who have developed .NET compilers for a number of languages. Because of its association with .NET, Web Services have generated a lot of interest, but Web Services are not .NET-specific. .NET provides special support for writing SOAP-based services; for example, SOAP Extensions [Bal01]. (SOAP Extensions are examined in a later chapter). Similar infrastructure is available from other vendors, including Oracle and Sun. Regardless of the platform, internal implementation and vendor, all Web Services share an unprecedented ability to work together as system building blocks.

Web Services are black-box software components that provide functionality via the web using standard Internet protocols. Architecturally, a Web Service is any application that can be accessed through a URL [Esp01]. Web Services support HTTP, which let clients issue calls and receive results by providing the underlying network transportation between clients and services. Web Services
typically use XML to package method names, input parameters, and return values. In principle, you can define your own XML-based protocol to invoke a particular service, but there is a need for a standardised protocol. The Simple Object Access Protocol (SOAP) is the protocol designed for this task. SOAP is the real-world standard for describing a remote method call; it’s an XML vocabulary whose tags and attributes describe the method to invoke on the service, and provides the arguments. What travels over the network is simply an XML string. SOAP has been developed through the cooperation of Microsoft, IBM, Lotus, UserLand Software and DevelopMentor. Designed for application-to-application communication, SOAP can be implemented from any programming language or operating system. SOAP is an alternative to other techniques for performing remote procedure calls. Through the use of XML and SOAP, an application does not need to know the operating environment of the service it is using, and vice versa. It is a true black-box application. The operating system on the services end can be virtually anything, as can the consumer or client of the service’s Operating System.

Web Services are served via web servers, so can be invoked across the Internet. The application must be able to parse the XML payload it receives and extract the information about the method and the arguments. After such a method executes, the return value is sent in another XML-SOAP packet. Web Services can communicate with any module running on any platform. A benefit of such flexibility is the ability to extend existing distributed systems to expose functionality as Web Services. As HTTP, XML, and SOAP are open internet standards and do not depend on any underlying object models such as DCOM, CORBA or RMI, such services are immediately ready for the hosting system and any client that accesses the network to use.

Web Services consist of functions/methods exposed on the web server which can be called by the clients from anywhere. Several components are required to make a Web Service work, these include: a way to discover a service provider on the web; a way to discover what services are available on a particular site; a way to describe how to interface with the service; a way to execute the functions provided by the service; a standard messaging format; verifiable means to prevent data interception and corruption (Encryption); and last but not least, a way to represent the data in the exchange. In order for the Web Services to work properly all these elements should work together. In short, Web Services are self-contained,
2.3. Web Services

modular applications that can be described, published, located, and invoked over a network, on the Internet.

The thesis focuses on Component Adaptation using Web Services. We examine how Web Services can be compared to software components. Web Services can be compared to any other software application that is delivered as a service. In order to use this service it should be made available to all clients by publishing or advertising. A Web Service can be consumed by clients and can be reused many times. In order to consume a service the client has to locate its addressing information and access it through the URL. One of the main motivations for consuming a Web Service is that it can be published and consumed by any programming language on any platform as it uses common Internet standards such as HTTP (for transport) and XML (for data representation) for the intermediate communication.

Web Services can be compared to software components in that they both provide well defined interfaces, can be reused, and can be accessed by different client applications. Software components connect to other applications or other components via interfaces [Smi99]. Web Services connect to other applications or other Web Services via the Web Service’s URL using standard Internet protocols. Many Web Services can be combined into one according to the application’s needs and the entire combination can appear to be one single system in the same way as different software components can be combined together to create one single application. Thus Web Services can also be considered as a form of software component, which can be reused, and data from heterogeneous applications can communicate across different technologies.

In the previous sections we discussed Web Services and requirements of component adaptation. As Web Services is also one form of a software component we examine why these (section 2.2) requirements of component adaptation can be extended to Web Service world:

- Web Service adaptation needs to be transparent as both the client and server should have no knowledge of any adaptation being applied on them.

- Web Service adaptation technique should not have any knowledge of the Web Service’s internal specification.

- Web Service adaptation technique should not depend on any platform.

- Web Service adaptation technique should not depend on any language.
A Web Service should behave the same way as before the adaptations were applied to them. There should be no change in the original Web Service because of this adaptation.

This literature concentrates more on Component Adaptation. As seen in the previous section, Web Services are important for business application integration. As more and more companies use web technology, the choice of Web Services increases. The services provided should be able to better match with the user’s requirements. However the proliferation of Web Service makes the standardisation of interfaces more difficult. Thus, as the number and complexity of choice increase, but the percentage of connectable Web Services may decrease. Hence adaptation is required to overcome mismatch. This will benefit the service providers by giving them access to a wider market. The customers will be benefited with a wider choice of solutions existing in the market. Thus Web Services and adapted Web Services benefit both the clients and the service providers by getting the best out of the Web Services approach and infrastructure.

Web Services are based on XML and HTTP, which are industry standards. XML is used for description of data, and developers can describe these data between PCs, smart devices, applications and web sites. The data presentation in different devices or browsers needs to be adapted so that the data can reach the client application in the correct format. Apart from the presentation point of view there may be quite a few instances where the data itself needs to be modified to be compatible with a particular client or a Web Service. As Web Service uses XML to describe data, our solution of XSLT based adaptation makes fair sense. These XML based data can be transformed into HTML or WML for presentation in different browsers or applications and from XML to XML for changing the structure of the document. It is proposed that these transformations be done using XSLT.

Different scenarios where Web Service Adaptation would arise

1. Simple scenarios: A simple scenario can be as simple as a client that requires an 'Address of a person including a post code', but the Web Service provides with two different functions that returns an address and another returns a post code. Thus an Web Service adaptation would be necessary to merge two different functions of the Web Service into one.
2. Application Integration [Cor01d]: In a world of disparate applications, Web Service adaptation can be used to integrate Web Service applications. It is not an easy task to create a functional composite from these applications without using adaptation. Using Web Services can expose the functionality and data of existing applications. Thus a composite application can be created using Web Service adaptation to enable cooperation between constituent applications.

3. Workflow solutions: Another scenario of Web Service adaptation could be workflow solutions where business to business transaction takes place. BizTalk server is a good example of a Web Service adaptation which adapts to different data formats.

Creating a Web Service

In this section creating a Web Service and how the client interacts with the Web Service are examined. A typical Web Service running on .NET platform uses:

1. XML for data exchange and description.
2. SOAP protocol for calling a Web Service, the Web Service requests and responses which are in an XML format are embedded in a SOAP envelope.
3. WSDL for describing the Web Service [Got00].
4. UDDI (Universal Description, Discovery and Integration) is a central organisation for registering, finding and using Web Services.

For the purpose of research, the Web Service was created using the .NET framework. A Web Service in .NET can be created in any language that the .NET Common Language Runtime (CLR) supports. When a client needs to gain access to a Web Service the first and foremost thing it has to do is to locate the Web Service. UDDI provides this service for registering, finding and using a Web Service. Once the client locates the Web Service using UDDI it has to know the details of the Web Service such as the name of the Web Service method, the return type, if any, the parameter types of the Web Service methods. WSDL (Web Service Description Language) is in a XML format that describes the Web Service. For all the Web Service created the .NET runtime creates a corresponding WSDL file. The WSDL consists of the details of the web methods.
provided by the service. It has the names of the methods along with the number
d parameters; types of parameters, and possible return types.

The client gains access to a specific service using a client side proxy to interact
with the Web Service. The .NET framework creates a client side proxy on the
client side using a tool called wsdl.exe. This client side proxy is created based on
the WSDL; thus when the client creates a proxy it needs to give the path of the
WSDL of the Web Service it needs to access. As the WSDL has all the required
information about the Web Service, the client side proxy created using this WSDL
also has all the required information to interact with the Web Service. The proxy
is another a C# or a VB.NET class which has the names of the methods, return
types, parameters and also has the addressing information on the Web Service.
All these are done automatically once the client runs the wsdl.exe tool.

When the client needs to invoke a Web Service method, it creates an instance
of the client side proxy and calls the proxy method locally. For the client it is as
if it calls a local method. The client side proxy packages this call, creates a SOAP
envelope and sends this request to the web service. In the same way the Web
Service packages the response in a SOAP envelope and sends this envelope to
the client proxy which marshals it into a state which the client understands and
passes on to the client. The message stages of SOAP as to how the messages are
serialised and de-serialised are examined in the later chapters of this literature.

In short we can say that Web Services are self-contained, modular applications
that can be described, published, located and invoked over a network, generally
the web. Several essential activities need to happen in any service-oriented envi-
ronment [Got00]:

- A Web Service needs to be created, and its interfaces and invocations meth-
ods must be defined.

- A Web Service needs to be published to one or more intranet or Internet
repositories for potential users to locate.

- A Web Service needs to be located to be invoked by potential users.

- A Web Service needs to be invoked to be of any benefit.

- A Web Service may need to be unpublished when it is no longer available
or needed.
2.4 XML/XSLT

An issue in today’s computing world is data sending or receiving to or from heterogeneous systems, applications or platforms. Different applications should be able to easily communicate and interact across different technologies. There are different browsers which support HTML, there are Wireless Access Protocol (WAP) enabled mobile phones which understand a markup language called Wireless Markup Language (WML), there are palmtops with a different markup language. In order to serve all these client applications there should be a single markup language which is understood by most of these common applications. One solution to all these is XML where a XML document transmitted over the Internet using standard Internet protocols can be transformed into a markup language which is understood by these devices. XML can be converted into HTML, WML or even to another XML document with a different result tree. Thus, using XML, one can have different views to the same data available. In this thesis to convert XML documents XSL (extensible stylesheet language) transformations are used. More than one XSLT can be used for the same XML document and also one XSLT can be used for more than one XML document. The content information will be the same in all cases but the presentation would be different. Users can define their own tags as there is no limitation as in HTML except it must be well formed. XML can also be described as hierarchical breakdown of information. To summarize we can say that XML represents content information and markup languages like HTML or WML represents presentation information.

XML is a data description language. It supports the description of tree-like data. XML data comprises elements and attributes. The latter represent atomic data, the former may be composite: data comprising further elements and attributes. XML may be typed though XML schemas. These enable guarantees to be made about data attributes and elements being present, in the right number, and having the right form. Thus we may guarantee that any XML data claiming to represent, for example, a person has a *name*, an *address* and optionally an *email address*. XML must be syntactically well formed and may require conformance to a schema; it is precise; there is no concept of trying to recover from badly formed XML unlike browsers’ interpretation of HTML. XML forms the core in the context of Web Services. There is also Document Object Model (DOM) that is used to navigate through a XML document programmatically. Using DOM, a new XML document can be created or an existing XML document can be
manipulated programmatically. We have used DOM in the adapter to create and manipulate XML documents using C#. All the investigations are done on adaptations for Web Services by transforming the XML document using XSLT.

XML is platform independent thus can be used extensively in cross platform integration. This makes XML appropriate for B2B transactions, e-commerce and other applications. As developers can use meaningful tags in XML describing the actual content of data the clients using this data on the Internet can search the product or service irrespective of the platform that serves the data or the platform the client is using.

XSLT is a transformation language. It can convert XML to other formats like HTML or WML. It is very popular when it comes to converting XML documents with other formats compliant by web browsers. It can also convert one XML to another XML but with a different structure. In this work we use XSLT to convert one XML to another XML to do adaptations, for example filtering some data from the source to form a new well formed XML document. Though XSLT is extensively used to convert one XML to another XML, converting XML to HTML or WML is out of the scope of this literature. For that reason we examine a simple XML document and transform this into another XML document possibly with a different structure. We would use XSLT for transformation.

Source XML document:

```xml
<?xml version="1.0"?>
<contact type ="business" gender="male">
  <name> Anand Iyer </name>
  <email> iyeranandv@hotmail.com </email>
  <phone> 33497675 </phone>
</contact>
```

We change the structure of the XML document in such a way that we create a new element `contact_info` instead of `contact`, remove all the attributes of `contact` which are `type` and `gender` and replace them as new elements under `contact_info`. So the new XML document would like this:

```xml
<contact_info>
  <type> business </type>
  <gender> male </gender>
</contact_info>
```
The XSLT file for the above transformation:

```xml
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="/">
    <contact_info>
      <xsl:for-each select="contact">
        <type> <xsl:value-of select="@type" /></type>
        <gender> <xsl:value-of select="@gender" /></gender>
        <name> <xsl:value-of select="name" /></name>
        <email> <xsl:value-of select="email" /></email>
        <phone> <xsl:value-of select="phone" /></phone>
      </xsl:for-each>
    </contact_info>
  </xsl:template>
</xsl:stylesheet>
```
1. Matches the root element (contact)
2. New element <Contact_info>
3. For each select of <contact> insert the following
4. Insert element <type>
5. Insert the value of type, '@' is for type is not a element but a attribute of element contact
6. Insert the value of element name, no '@' as name is an element and not an attribute

The Figure 2.2 illustrates how XML and XSLT are useful in different platforms. The most popular use of XSLT is to convert XML documents into the data format which is recognised by different web browsers. As said earlier we have used XSLT to transform XML documents. The elements of an XML document might be reused, repositioned, sorted, split or merged with other content, or even transformed into attribute values. Most of the XSLT written have been used in combination with XPATH.

![Figure 2.2: Use of XML and XSLT in different platforms](image)

XPATH is a separate standard that defines a general purpose language for interrogating XML document structures. Using XPATH, specific formatting can be applied to elements when they appear at a significant place within the document structure. This kind of advanced analysis and manipulation of data is performed using this expression language. It is also used as a query language and for advanced hypertext linking schemes. The following XPATH expression
identifies $ServerInterface$ elements within $ClientInterface$ elements that have a $name$ attribute value of $GetProductPrice$:

ClientInterface//ServerInterface[@name='GetProductPrice']

### 2.5 Simple Object Access Protocol (SOAP)

The Simple Object Access Protocol is a simple and lightweight protocol, particularly suited to use across the Internet [W3C00a] [Sko00] [Spe01]. It is an XML based protocol which consists of three parts: a SOAP envelope which describes what is in a message; a set of encoding rules; and convention for representing remote procedure calls and responses. It encodes remote procedure calls using XML and uses XML schema as an interface description language; that is XML schema describes Web Services. Web Services Description Language (WSDL) is basically an XML syntax which describes the Web Services, this is same as the Interface Definition Language (IDL) is to CORBA. When we compare SOAP with IIOP or DCOM they are binary protocols whereas SOAP uses XML to communicate. SOAP is designed for decentralised distributed computing. It does not encompass a distributed object model requiring garbage collection etc. In general, SOAP is hidden from the developer; stubs and proxies convert the developers native calls and callbacks into SOAP requests and responses. Through the use of XML and SOAP, an application does not need to know the operating environment of the service it is using, and vice versa. It is the true black-box application. The operating system on the services end can be virtually anything, as can the consumer or client of the service’s operating system.

SOAP uses XML to represent the data, which means that an XML document is embedded in a SOAP message. A SOAP message has a HTTP header and a SOAP envelope. The SOAP envelope can be divided into two parts; SOAP header and the SOAP body. The SOAP header consists of the XML header data for example the namespaces. The SOAP body, in case of a Web Service request has the information of the name of the Web Service method to be invoked, the parameters passed and the data; in case of a response it contains the response received from the web service. All the information in the SOAP envelope is in an XML format. Following is an example of a SOAP packet including the HTTP header.

Web Service Request
POST /ProductDetails.asmx HTTP/1.1 Connection: Keep-Alive
Content-Length: 317 Content-Type: text/xml;
charset=utf-8 Expect:
100-continue Host: localhost
User-Agent: Mozilla/4.0 (compatible;
MSIE 6.0; MS Web Services Client Protocol 1.0.3215.11)
<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope
xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
<soap:Body>
<GetProductDetails xmlns="http://tempuri.org/">
<productId>2</productId>
</GetProductDetails>
</soap:Body>
</soap:Envelope>

Response from the server

POST /ProductDetails.asmx HTTP/1.1 Connection: Keep-Alive
Content-Length: 317 Content-Type: text/xml;
charset=utf-8 Expect:
100-continue Host: localhost
User-Agent: Mozilla/4.0 (compatible;
MSIE 6.0; MS Web Services Client Protocol 1.0.3215.11)
<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope
xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
<soap:Body>
<GetProductDetailsResponse xmlns="http://tempuri.org/">
<GetProductDetailsResult>Gear, 10, gear.bmp,13
</GetProductDetailsResult>
</GetProductDetailsResponse>

Response from the server

POST /ProductDetails.asmx HTTP/1.1 Connection: Keep-Alive
Content-Length: 317 Content-Type: text/xml;
charset=utf-8 Expect:
100-continue Host: localhost
User-Agent: Mozilla/4.0 (compatible;
MSIE 6.0; MS Web Services Client Protocol 1.0.3215.11)
<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope
xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
<soap:Body>
<GetProductDetailsResponse xmlns="http://tempuri.org/">
<GetProductDetailsResult>Gear, 10, gear.bmp,13
</GetProductDetailsResult>
</GetProductDetailsResponse>
In the above example a *GetProductDetails* request is sent to a *ProductDetails* service. The name of the method to be invoked is *GetProductDetails* that is specified inside the SOAP body which takes in a numeric value `productId 2` and returns a string that is inside the *GetProductDetailsResult* element in the response packet. If the Web Service couldn’t respond due to some problem then the SOAP message returns the client with a SOAP fault code and a SOAP fault string which is again in the SOAP body element. Thus a SOAP response must contain either the result from the Web Service or the SOAP fault code. The Web Service request and response are both valid XML documents. All the SOAP messages contain an *envelope* element that contains a minimum of one *body* element.

A SOAP application should include the proper SOAP namespace on all elements and attributes defined by SOAP in messages that it generates, it must be able to process SOAP namespaces in messages that it receives, it must discard messages that have incorrect namespaces and it may process SOAP messages without SOAP namespaces as though they had correct SOAP namespaces [W3C00a].

SOAP, which is a protocol for a client-server communications across a network, uses HTTP and XML to send and receive messages. Through the use of HTTP, clients and servers can pass through the firewalls. SOAP defines how to access services in a platform independent manner using HTTP and XML. All the adaptations investigated in this thesis are done using transformation of XML document that are embedded on a SOAP envelope, thus removing language dependence.

Until now we examined Web Services and technologies that are used in Web Services. The technologies examined will be used in this thesis for adaptation of Web Services. The rest of this chapter will focus on related technologies of XML(EDI) and an existing adaptation technique on Web Services(BizTalk).

## 2.6 EDI

Electronic Data Interchange (EDI) [fMASII99] is the exchange of standardised business documents between computers. Documents such as purchase orders and invoices are transmitted from one computer to another in a mutually agreed
upon electronic format. In short, purchase orders or invoices are processed by the supplier or the buyer without human intervention. EDI uses Value Added Network (VAN) for communication. VAN is responsible for moving the transaction through a network to the addressee. There are also Value Added Services (VAS) which provides translation services, conversion from Fax to Internet to EDI.

The basic components involved in EDI are EDI translator, the communication protocol (VAN, VAS), software applications and hardware (LANs, gateways etc.). These components are briefly explained later. Apart from these there are standards for EDI. The most common standards are X12 and EDIFACT. These standards set some definite rules regarding the syntax, editing rules and conventions, and mutually defined transmission protocols. As each of these standards have different message formats, a message format under X12 is called a transaction set and in EDIFACT it’s called a message.

When using EDI for transmitting data, the sender needs to transform the transactions from the format, which the applications understand into the standard EDI format. The receiver of the document needs to transform these EDI documents into a format understood by the receiver’s applications. Figure 2.3 [fMASII99] examines a general communication of an EDI between two organisations under the X12 standard.

Though EDI has been used extensively the basic problem with EDI is to produce a single universal standard. The EDI standards process is cumbersome and this has led to overcomplicated standards with many different interpretations [Wea01]. A very detailed technical negotiation needs to made for implementing EDI with each new partner. There has always been the problem of compatibility when communicating across networks, this led to the very fact that EDI messages are mainly communicated via proprietary communication networks which proves to be relatively expensive. However there are some advantages to this technology. Advantages on the seller’s and the buyer’s side can be identi-
When purchase orders are processed without human intervention both the supplier (seller) and the buyer are benefited in the following ways [Wea01]:

For the supplier:

- As the data is processed without human intervention there is no need for data to be keyed in the order. This saves a lot of time and also eliminates the errors resulting from this process. Thus the accuracy of the data is maintained at the same time as it saves cost and time.

- Due to the fact that the accuracy of the data is maintained there is less potential for incorrect deliveries. Less incorrect deliveries means less returns which in turn contributes to less handling and processing costs.

- The goodwill of the supplier is maintained in view of less returns and errors.

For the buyer:

- As the data is processed electronically the transaction costs would be far less than normal costs where manual intervention is required.
• After the advent of the Internet the buyer can have the luxury of having dealings with internationally accredited suppliers easily which indirectly contributes to globalisation of the economy in which the buyer is located.

• There are other advantages, such as review of historical records with a specific supplier, easy search capabilities, shipment tracking.

**Technical Details of EDI**

In this section the steps and stages of EDI messages are examined:

• The sender’s application system creates a formatted transaction document.

• The EDI translator transforms this document in a format which is agreed upon between the buyer and the seller using appropriate segments and elements.

• The transformed document that is the EDI document is sent via the network. For example the sender’s computer will connect to a Value Added Network (VAN) communication and route this EDI document to the receiver’s box.

• The EDI translator on the receiver’s side receives the document which parses the EDI document and checks if it matches with the agreed upon format, and translates this EDI file to a business application format, which the business system understands.

When the sender’s application system creates a transaction document it would possibly be in a human readable format. Translating this human readable format to a specific format (ANSI X12 or EDIFACT) which an automated receiver can understand is the main job of an EDI translator and this translated document is the EDI file. We examine how these elements are created and how these elements are combined together to form EDI messages that are transmitted in electronic envelopes.

The first part of EDI is the data element. The data element describes the type of data, whether it’s numeric or alphabetic, the minimum and the maximum length allowed for a specific data and the conditional values that must be adhered to with a particular type of data. Thus an address is not a data element as it can be broken down into several more elements like the *street name, suburb name,*
state and country. A street name cannot be further divided thus represent an element. This address as a whole, which has several elements, represents one logical group of information and this as a group is the segment of the EDI file. If there is more than one address in a purchase each address is referred to as one segment and each segment is identified by a code. Thus the name of the product, unit price, quantity may form another segment, thus all the segments combine a together form an EDI message or what is often referred to as a transaction set. This message with the segments and the data elements can be one complete business document such as a purchase order or invoice. The message definition would have information such as which are the segments to be used, the order or the sequence of the segments to be used and possibly have more information depending on the standards agreed upon by the two trading partners.

When two trading partners want to begin to share documents then both would possibly interchange the guidelines and format of the EDI to each other and eventually agree on a format as there can be different formats and standards for EDI messages. Thus the definitions would differ depending on the standards used. So each message definition might be different from the other. EDI uses several levels of envelopes in order to ensure that each message document is correctly identified and only the like documents are grouped together. An envelope can contain more than one message document and also each envelope can contain inner envelopes. The inner envelopes contain different message documents. One can imagine an outer envelope containing smaller envelopes each for a set of invoices, product information, and set of purchase orders. Each of these inner envelopes would have the addressing and the routing information. Thus each of these envelopes can use different formats of EDI messages.

Thus the main elements of an EDI message would consist of a data element, data segment, message definition and an envelope.

**Transformation of EDI & XML documents**

An EDI document is not in an XML format. There are ways where we can transform an EDI file to an XML format. These are discussed in the later part of this chapter. Transformation of EDI documents is governed by a software component called EDI translator. The main services provided by EDI translator are data mapping and standards formatting. The primary role of an EDI translator is standard formatting. In order for this to happen the translator needs to under-
stand the format of the data. This is achieved by the translator providing a tool called a data mapper that allows the user to specify the format of the data. Data mapping reduces the programming required to integrate the translator with a business application. This can be compared to Microsoft’s BizTalk which also provides a mapper tool which allows the user to map the source and destination fields. The only difference in Biztalk is that the data is in XML format and transformation is performed using XSLT. The EDI translator also needs information of how to access the data to which most translators support a file interface. For example when a buyer is sending a document, the buyer’s application writes the transaction data to a flat file, the translator formats the data according to the format and the EDI syntax rules agreed upon by the seller, and produces an EDI file. The seller’s translator that receives this document verifies the standard, the compatibility of the format, and translates the EDI file to a flat file for the application as output.

The transformation of XML documents is done using XSLT. XSLT in this case is the translator to XML documents. In most of the cases the XML document that is created by the application system is directly transformed using XSLT unlike in EDI where the transaction data is transformed to a format, which is understood by the EDI translator that transforms it into an EDI file. We examine some basic simple advantages and disadvantages of XSLT as most of XSLT have been already covered in the previous chapters. The advantages being portability, reusability, language independence, can transform from XML to text, HTML, WML, or XML. It is useful for making the data compatible to different browsers. Disadvantages include only being able to work with a wellformed XML document, not very user friendly, case sensitive, not suited for translations from text to XML.

After discussing the main components of EDI we now examine some of the differences between XML and EDI. One of the most important advantages of XML over EDI is XSLT which provides easy translation of data between different message formats. However EDI also has a EDI translator; thus when two companies want to share data using EDI they can create new mapping from their existing format to an agreed upon EDI format. But the main drawback of this would be that XML has schemas, which saves time from having to hard code some of basic validation. In XML, defining new data types is easy, but in the case of B2B communication, the schemas for different industry applications are under development. However in the case of EDI it uses data types defined within
standards for example EDIFACT, ANSI X12. But in the past, adhering to these standards has been a problem for small organisations due to the fact that each new partner adheres to different standards and negotiations need to be done with trading partners. XML has proved to be excellent for high volume transactions. In EDI all the data segments and data fields have some meaning and organisations adhering to a specific standard can create and use these data segments which have specific meaning in each and every message packet. But in the case of XML messages there needs to be an agreement on the meaning of data elements, which data elements are required by different business process, and how to transmit the XML based messages. These problems connected to XML has been reasonably solved through ebXML (Electronic Business XML). The aim of ebXML has been to create a global electronic market place where organisations of any size irrespective of their location can conduct business with each other through the exchange of XML based messages. The problem of transmission of XML data has been solved through the use of SOAP by ebXML. There is also provision of an online repository that lists data elements and their meaning, together with definitions of business messages using those data elements. Thus an organisation that receives an ebXML message can search the repository to find the meaning of the various elements within the message. Similarly an organisation wishing to create an XML message can search the repository to see the data element that fits their requirement. Thus the basic problems of XML regarding transmission, data elements have been solved by ebXML.

The above discussion was not to prove which technology is more superior than another but give a comparison between these two technologies. It is seen that both have advantages as well as drawbacks thus cannot be used interchangeably.

**Use of EDI in XSLT based adaptation**

This thesis mainly constitutes adaptation using XSLT on Web Services. EDI has been used for over 20 years. As seen in the above discussion there are advantages as well as drawbacks in both the XML and EDI technologies. This section aims to relate EDI to XSLT and examine how it can be used in this XSLT based adaptation and also what would be the problems faced for transforming from an EDI technology to XML.

In order to use XSLT one has to transform all the present documents to a XML format. In the case of EDI, the company’s documents need to be transformed into
XML. However transformation of EDI to XML is not straightforward and faces some problems. In order to apply adaptation using XSLT on an EDI document the following problems must be dealt with:

- XML has the luxury of DTD and XML schemas but as of yet EDI has lacked the formal equivalent of DTD/schemas for describing the standard messages or implementation guidelines in a complete way. Thus it is hard to expect any standard transformation approaches from EDI to XML as there aren’t any standard starting points. However a number of organisations like X12XML are working on specific guidelines to transform EDI data dictionaries into XML. But the ideal technical approach has not been adopted so far.

- In addition to business content, EDI messages typically contain the delivery destination and information about workflow or “choreography” such as the message identifier to which the message is a reply, whether acknowledgments are expected, and other information that is needed to deliver the message in accordance with the agreements between the trading parties. Many XML architects would argue that separating the message content from this addressing and workflow information as separate documents and conveying them using multipart MIME would allow better technical approaches for message routing, security, error handling, authentication, confidentiality, and so on. However, no standards yet exist for how XML documents should be “wrapped” with this delivery metainformation and the rules by which it is used to ensure that messages are delivered and processed as they are intended.

- Another problem is also a limitation in transforming EDI to XML. EDI came into existence mainly for e-commerce application. Thus EDI has e-commerce related semantic information that XML DTD cannot handle. However this has been overcome by XML schemas, which contain primitive and user defined data types which can preserve the semantics of EDI messages.

These are some of the basic problems which can be faced when transforming the EDI data dictionaries to XML format. If some of these basic problems are overcome then the old and primitive data can be transformed into XML and also
the semantics of the EDI documents preserved. Once this is done one can apply adaptation on these XML documents using XSLT.

2.7 BizTalk

Microsoft’s BizTalk Server 2000 [Cor01a] [Res01] provides a higher-level assembly of .NET Web Service solutions. Using Microsoft Visio 2000 [Cor01b] as a front end, developers and designers ‘draw’ solutions connecting Web Services into applications. Microsoft Visio 2000 is a software that can be used by designers and developers to draw diagrams such as UML. The BizTalk engine then ‘orchestrates’, or handles the communication and translation of data between the Web Services. While developers will still need to create the Web Services using Visual Studio or other development tools, BizTalk lets non-programmers design the business rules and data flow for applications and solutions. The three main tools of BizTalk server are Editor, Mapper and the Application designer are described below:

**Editor**: This is a tool for creating document specifications that represent Extensible Markup Language (XML), flat files, and Electronic Data Interchange (EDI) files, as well as importing existing Document Type Definition (DTD) and XML-Data Reduced (XDR) schema. This is the place where data elements are defined.

**Mapper**: Mapper is a tool that enables the user to map the source and the destination fields. This is very ideal for mapping document contents between different formats. The BizTalk Mapper would play a key role in mapping the source and destination files, after compilation and depending on the mapping it creates a XSLT file dynamically for adaptation. Mapper is discussed later in this chapter.

**Application Designer**: Application Designer is a tool for Business process logic that the data will move through; this is done with the help of Microsoft Visio 2000 [Cor01b].

Visual Tools let the designer graphically define transformations rules between different document structures so that XML, EDI or flat file documents moving between systems can automatically be converted at runtime, this allows integration with nearly any existing application today without the time, cost and risk of having to modify it no matter what structure of information format it utilises.
Chapter 2. Foundations

To transform data mapping it can use functoids. Functoids are small programs which help transform and manipulate data in customisable ways. Functoids help transform data from documents. Using functoids, data can be added together, data can be concatenated, data time information can be modified or other similar operations can be performed. In short it can be said that BizTalk can help translate or map data from one format to another and deliver those messages.

BizTalk server employs XML as its internal data file format. All inbound documents are parsed and stored as XML, regardless of their formats (EDI, delimited text). Outbound documents are serialised from XML into the format appropriate for the receiver. These documents are parsed using the schemas that are shared by the trading partners. BizTalk completes this process using a pair of function sets: Orchestration and Messaging. Orchestration handles business functions. It lets you create processes graphically and connect them to code constructs capable of carrying them out. Messaging is a set of facilities that performs basic data integration functions such as data description and field mapping from one application to another. To make this process work, you need to tell the system the data definitions of files you plan to use (fields, data types, etc.); how to map fields from one data set to the other; and how to process data flows using which communications channels to which destinations and with which, if any, imposed conditions.

BizTalk Editor is the tool to describe the data structure (records, fields etc.) of all kinds of files, including database, EDI formats, XML and variously delimited flat files. It helps users define specifications to describe both inbound and internal documents.

BizTalk Mapper is the tool used to connect analogous fields between source and destination files. The Mapper provides a graphical way to create maps. Maps are created by connecting records or fields in the source document specification to records or fields in the destination document specification. Transformations are inserted into the field between the source and the destination with translation objects called “functoids”. For example, an e-commerce system requires a customer’s address data in separate fields such as Street Name and Street Number but the provider stores the data in a single string called CustomerAddress. We can use a functoid to concat the street address fields together and map the result to the CustomerAddress field. Functoids are grouped functionally and presented on a palette. There are functoids for string conversion, logical operations, database
operations, mathematical operations, and so on. If requiring complex business logic in your translations, you can create your own functoids. Custom functoids can even call COM components, allowing you to bring other development tools into your toughest data translation problems.

In this chapter, we considered Component Adaptation and Web Services, and how software components can be compared to Web Services. A basic motivation for Web Services was discussed, informing why we chose Web Services for adaptation. Then we discussed some of the existing adaptation techniques in the industry and requirements of component adaptation. These requirements will be evaluated with our XSLT based adaptation in Chapter 6. As this research was done on Microsoft’s .NET platform, discussion on .NET platform and how Web Services are created on .NET was important. The later section of this chapter included XML, XSLT and SOAP. It also demonstrated an example changing the structure of a XML document using XSLT. The adaptation here is done by changing the structure of SOAP requests/responses; in the discussion of SOAP we examined some example Web Service requests/responses and example XSLT document looking at how to change the structure of these requests that are in XML format using XSLT. We discussed EDI, how EDI formats can be used in adaptation, then we compared EDI with XML and transformation of EDI using XSLT. The important part of related work is BizTalk because adaptation used in BizTalk is similar to our adaptation technique.

This chapter demonstrated some of the important tools used in Web Services. Using these tools it is proposed to construct an adaptation technique for Web Services. Before we conclude this chapter it is important to have a discussion of how a Web Service adaptation should ideally be done using these tools. XML and XSLT form the core of the Web Service adaptation. So the following technique would use these two tools as the base for adaptation and XSLT for transformations.

- The adapter component is configured to send and receive messages from the client and the Web Service.

- The adapter component has the mappings related the Client’s requirements and services provided by the Web Service. This can be established by XML schemas of client requirements and the Web Services provided.

- The URI specification of the XML schemas can be stored in the Guiding
Specification that proposed in this thesis.

- When the client request comes in, the adapter maps the XML schemas with the Guiding Specification and automatically generates a XSLT document on-the-fly. This XSLT is then applied to the XML schema mapped for the Web Service on the Guiding Specification to create a final request. The final request being another XML document is sent to Web Service.

- The response from the Web Service can be transformed in the same way as the request is done and the responses send back to client.

To conclude, this chapter included some of the basic technologies used in this research and discussion on the work related to this thesis. In the next chapter a detailed design of XSLT based adaptation and its preliminary results will be examined.
Chapter 3

Preliminary Results

In the previous chapter we saw some of the technologies and related work used in this research. This chapter will concentrate on the broad design, research methodology and approach done for XSLT based adaptation using those background technologies. Apart from the preliminary results and a broad picture this chapter also focuses on evaluation of XSLT based adaptation against the requirements of component adaptation discussed in the previous chapter.

This section presents examples of XSL based adaptation done initially. Investigations have been done using the client as well as the server to see how adaptation may be performed. Fundamentally adaptation assumes the client and server code has already been deployed, are fixed and cannot be changed [AIP02]. Our approach to adaptation seeks to provide a wrapper to cope with minor mismatch between client and server interfaces. If there is a wide discrepancy between the provided and required service it is considered to be outside the scope of this work. The preliminary work and the results are explained in the following sections.

At the core, adaptation can be thought of as a transformation of the messages being passed between the client and the provided services. With SOAP these messages are encoded in XML, hence the transformation of these XML messages suffices to provide adaptation. It is proposed that the Extensible Stylesheet Language Transformations (XSLT), which is a language for transforming XML documents [W3C99] [W3C00b] is to be used for this purpose. Each SOAP message is considered as a separate XML document, and the adaptation required for each message is described using XSLT. The advantage of this system is that
the transformations are platform independent. They can work for most of the clients. The XSLT transformations can be applied to all incoming and outgoing messages.

Figure 3.1 shows generic high level adaptation architecture. It does not place adaptation either on the client or on the server side. In reality, adaptation will occur at either the client or the server side of the intranet or Internet. The rest of the section discusses the investigations and the implementations done to adapt to a Web Service. Adaptation is considered on the client as well as on the server side.

The above diagram is a general examination of a high level adapter. The adapter is applied either on the client or on the server side. This generic adapter has a list of addressing information of the Web Services that it needs to utilise and also XSLT libraries which are used to transform the XML messages.

When the adaptation is applied on the client side the XSLT libraries and the adapter need to be deployed on all the clients where adaptation is needed for a particular Web Service. In the case of server side adaptation the adapter and the XSLT libraries need to be deployed on the server only once for all the clients to which a Web Service needs to be adapted. To cater for additional clients the adapter simply requires the addressing information, client requirements and the additional XSLT scripts for its libraries. We now examine the detailed design of this adapter and explain how it works.
3.1 Adaptation using SOAP Extensions

Web Services can communicate via SOAP. SOAP packages XML documents to be sent to and from the server. The requests sent from the client and responses received from the Web Services via SOAP pass through different SOAP message stages. Before we explain how client side adaptation can be achieved using XSLT we examine the different SOAP message stages.

SOAP Message Stages:

The messages sent and received via SOAP are passed through distinct stages and at each stage the form of the SOAP packet may be different. The transmission of a SOAP packet can be observed at 4 stages. This transmission of a SOAP message and the observed stages are detailed in Figure 3.2

![Figure 3.2: SOAP message stages using SOAP extensions on the Client Side](image)

The four stages are described below:

1. **Before Serialise**: The client invokes a Web Service method using an instance of the client side proxy.

2. **After Serialise**: The client side proxy with the help of the .Net runtime creates a XML document that contains the name of the method to be invoked and its arguments, which are embedded in a SOAP envelope to be sent to the Web Service. The addressing information is then added to the Web Service by the client proxy that has the URL address of the Web Service.

3. **Before De-serialise**: When the request has been received on the server side the .Net runtime transforms the XML/SOAP message into the native language in which the Web Service is written. The service sends the response, which is again transformed and packaged in a SOAP envelope and sent to the client. Once the response is transformed into a XML document it enters the Before De-serialise stage.
4. **After De-serialise**: The client proxy receives the response and transforms this back to the native language and sends this data back to the client.

These SOAP messages and the stages are transparent to the developer. When the client invokes a Web Service method what goes along the wire to the Web Service is an XML document embedded in a SOAP envelope. The developer requires no knowledge of the data and has no direct access to the SOAP packet which travels on the network. SOAP extensions [Bal01] are used to access the XML data format on the client side, before it is de-serialised into objects. Using SOAP extensions the developer can access the data stream and the contained messages. This access is essential to allow manipulation of the data. Once the SOAP envelope is accessed it can be transformed into another XML document using XSLT, re-packaged and sent to the Web Service.

An example of a SOAP message accessed using SOAP extensions on the client side is shown below:

```xml
<?xml version="1.0"?>
<soap:Envelope
    xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
    <soap:Body>
        <GetProductDetails xmlns="http://tempuri.org/" 1
            <productId>2</productId> 2
        </GetProductDetails>
    </soap:Body>
</soap:Envelope>
```

1. Web Service method invoked by the client.
2. Parameter passed by the client.

### 3.2 Example Scenarios

The rest of this chapter examines how adaptation can be applied to Web Services in a variety of scenarios. The purpose of this is to illustrate and explain the
preliminary results of adaptation for Web Services using XSLT. The adaptations use SOAP Extensions at the client side proxy, and experiments carried out are using transformations of XML message that is transmitted by SOAP.

**Scenario 1: Data Manipulation**

This example illustrates how the number and type of parameters can be changed or transformed on the client side using XSLT. In this case the number of method calls remains the same. The client requires the price of a product using the method `GetPrice(ProductDetails det)` where `ProductDetails` type is an object which contains a `id`, `name` of the product. The Web Service provides a service with the same method name but accepts parameters of type `int id` and string `name`. Thus the adaptation needs to extract `id` and `name` from the `ProductDetails` object. This is explained by examining the body of the SOAP message. The client would send a SOAP message where the body would look similar to this:

```xml
<GetPrice> 1
<ProductDetails> 2
   <id> 2 </id>
   <name> Gear </name>
</ProductDetails>
</GetPrice>
```

1. Name of the Method
2. Parameter Type

In the above SOAP message the Web Service cannot understand the tag `ProductDetails`. Thus the type of `ProductDetails` has to be converted into `int & string`. The SOAP body of the message to be sent to the Web Service would look similar to this:

```xml
<GetPrice> 1
   <id> 2 </id> 2
   <name> Gear </name>
</GetPrice>
```
1. Name of the method
2. Parameter Type

In this scenario we have kept the name of the method the same but have changed the type of arguments and also the number of arguments.

![Diagram showing the adaptation process](image)

Figure 3.3: Adapting to different parameter types & number of arguments

Scenario Details:

SOAP extensions are used to capture the message on the client side proxy. The client side proxy is created automatically by the wsdl.exe utility available on .Net. The adaptation applied is done at the After Serialise stage for the SOAP message. There has been only one XSLT used in this transformation.

- The client request is accessed at the After Serialise stage which is an XML embedded SOAP envelope.
- A user-defined XSLT is loaded depending on the adaptation required.
- This XSLT is applied to the XML/SOAP envelope.
- The transformation applied is on the After Serialise stage of the SOAP message cycle.

The XSLT used for this transformation:

```xml
<?xml version='1.0'?>
<xsl:stylesheet
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    xmlns:tempuri="http://tempuri.org/" version="1.0">
    <xsl:output method="xml" encoding="us-ascii" />  
    <xsl:output indent="yes"/>
```
3.2. Example Scenarios

<xs:template match="/">
  <soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
    xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    <soap:Body>
      <xs:apply-templates select="/soap:Envelope/soap:Body/*"/>
    </soap:Body>
  </soap:Envelope>
</xs:template>

1. Apply the SOAP envelope and SOAP body
2. Copies the current element (GetPrice)
3. Selects the nodes that are to be processed
4. Copy the value of the current element, for each selected nodes

Scenario 2: Changing of Method Names

In this scenario the parameters and return type of the method are the same but are adapted to a method with a different name. In this case there is a one-to-one mapping, but the same adaptation is required as part of splitting a single call into multiple calls. The client requires GetProductPrice (int id) but the Web Service provides for a method GetPrice(int id). Both method return a string which contains the price of the product. As both methods have the same number of arguments and are of the same type the adaptation requires only the change of the method name. The stages and the steps as to how the messages are sent and transformed are examined below:
• The client invokes a method in the client proxy.

• Client proxy packages the method call in a SOAP format along with the parameter, if any. In this case,

  <GetProductPrice> <id> 2 </id>
  </GetProductPrice>

• XSLT transformation is applied on the XML/SOAP packet of the After Serialise stage of the SOAP cycle before it is sent into the network cloud. Now it looks like,

  <GetPrice> <id> 2 </id> </GetPrice>

• GetPrice(int id) request is sent to the Web Service.

• GetPrice(int id) response is sent from the Web Service to the requesting client.

• Response reaches the client side proxy but cannot understand the element

  <GetPrice>

in the SOAP body sent by the Web Service, now the SOAP message cycle is in the Before Deserialise stage.

• One more XSLT transformation is applied on the Before Deserialise stage which transforms

  <GetPrice>

  to

  <GetProductPrice>.

• The client side proxy sends the transformed response in the same language compatible to the client.

XSLT is used for this transformation; we have omitted the first part of XSLT for brevity.
3.2. Example Scenarios

In the above scenario the adaptations have again been done using SOAP extensions and XSL transformations, however this is different to scenario 1 in the sense that the transformations are applied to outgoing and incoming messages, that is, transformations occur when sending to the Web Service and also transformation is done while the response is sent from the Web Service to client. Thus there has been a use of two XSLT documents. In the previous example the transformations were applied only to the outgoing SOAP packets and also only one XSLT document was used. The return XSLT would look similar to what we examined above, except the names GetPrice would be replaced by GetProductPrice and vice versa. This ensures the client receives a response in the expected format.

Scenario 3: Refactoring

All the previous examples were pertaining to changing parameters and changing method names however accessing multiple services was not covered. Accessing multiple services would mean that the adaptation applied should involve mapping one-to-many calls. In this scenario we examine an adaptation that requires access to multiple Web Service methods for a single client method call. Figure 3.4 provides an illustration of this scenario.

The client requires details of a product via a method call GetProductDetails which takes an id of an integer type. The details of a product constitute name, price, image and quantity. But the Web Service provides four different methods
one each for name, price, image and quantity with method names *GetName*, *GetPrice*, *GetImage* and *GetQty* respectively all of which takes in an *id* as a parameter.

The adaptations accept the request from the client and create four calls to the Web Service, and the response from the server is marshalled into one result and returned to the client. Here again the SOAP extensions are used to intercept the message and transformations applied on the XML document using XSLT. Investigations attempted to complete adaptation SOAP extension for this kind of a scenario. Following were the issues found when trying to adapt to multiple services:

- A separate XSLT document is required for each call because when implementations are done using XSLT document then we need to use four XSLT documents to make four calls to the Web Service. This is because creating more than one XML document on one source XML document using XSLT is not possible. However this is under consideration for future versions of XSLT (requirements for XSLT 1.1).

- SOAP extensions allow the use of only one XSL transformation when these
are applied on the client side proxy. This is due to the fact that SOAP extensions are divided into message stages. Each stage takes only one XSL transformations at one time when marshalling requests or responses, and in the above examples SOAP extensions are used as an attribute to the method that the client requires.

- When speaking of creating an adapter of this type for a generic purpose adaptations applied on the client side proxy would be considered cumbersome because every time a client wants some kind of adaptation to happen all the necessary XSLT files and the source code for SOAP extensions needs to be installed on the client side.

- The addressing information cannot be changed; this means that messages cannot be redirected to some other Web Service. This is because it is difficult to access the HTTP header using SOAP extensions that have the addressing information of the Web Service.

Hence it is very unlikely to complete this refactoring using SOAP extensions. Taking the above factors into consideration the same kind of investigations are considered for adaptations without the use of SOAP extensions.

### 3.3 Adaptation using Broker

The rest of this thesis will consider applying adaptation using a broker. Adaptation can be achieved for all the above scenarios with the adaptation being done using a broker. Due to the drawbacks of SOAP extensions, SOAP extensions were not used when applying adaptation, instead a broker that utilises existing Web Services was introduced. We have called this a broker as it acts as an intermediary between the client and the existing Web Services. The adaptation happens in two stages, one for the client request and the other for the server response. In stage one, all the requests sent by the client are directed to the broker, the broker applies adaptation and forwards modified requests to the existing Web Services. In stage two responses received from the Web Services are again interpreted by the broker that creates a new SOAP packet and this new SOAP response is sent to the client.

Premilinary results included initial work done on SOAP extensions and how adaptation that offers generic runtime support can be possible. We saw that for
a generic kind of a scenario adaptations using SOAP extensions were difficult and not practical for a real-world scenario. This is the point from where the design for adaptation using a broker started. In the next chapter we will examine a high level design for adaptation using a broker that will look at all the scenarios where adaptation was difficult using SOAP extensions.
Chapter 4

Architecture & Design

The last chapter saw some of the early work done on adaptation. There was a detailed discussion on the SOAP message life cycle and how adaptation can be applied using SOAP extensions. This chapter will examine the architecture proposed for adaptation using a broker.

4.1 High Level Design

This section will examine the basic design proposed for adaptation of Web Services. It describes the architecture that is designed for XSLT based adaptation for Web Service. This adapter is transforming XML embedded SOAP messages using XSLT. This performs data translation in a single step without needing to translate into any kind of intermediate format. It has a generic Guiding Specifications which has a list of addressing information, the XSLT file to be invoked and also a SOAP action attribute which adds a header information to the messages sent and received to and from the broker.

Before the discussion of a high level design it would be worthwhile to look at the most important component of this architecture, the Guiding Specification.

4.2 Guiding Specification

The Guiding Specification forms the core part of this broker, this is because the specification contains all the required information about what the client needs and
what the Web Service provides. We can also say that it acts as an interface between the requirements of the client and the services provided by the Web Service methods. This section will discuss the elements inside the Guiding Specification that will explain how this works with the broker. The XSLT libraries contain all the XSLT files that are mentioned in the Guiding Specification. Figure 4.1 provides an example of a Guiding Specification.

We now examine the structure of the Guiding Specification in detail. The ClientInterface element contains all the necessary information for one client. The adapters have no limit for the number of ClientInterface elements. The number of ClientInterface elements consists of the number of clients this adapter serves. In this example it serves two clients. The SoapAction attribute of the ClientInterface contains the SOAP action header of the client. This is where the Xpath expression compares this header with that of the actual client request. So the Xpath navigator goes to each and every ClientInterface element and compares the SOAP action header. Once this is matched with that of the client request it enters the ServerInterface element. The ClientInterface elements also have no bar of having a number of ServerInterface elements. When there is more than one ServerInterface element inside the ClientInterface it specifies that the broker would access more than one Web Service method in order to satisfy all the information needed for the client. In the above example for the first ClientInterface the adapter sends two Web Service requests to satisfy the client’s one request. The ServerInterface element has the attributes:

URL: this is used for the broker to send the request. This is the URL of the actual Web Service to be accessed by the client, but the broker interprets the request, changes the format of the SOAP packet by applying appropriate XSLT which is compatible to the Web Service, and sends it to this URL.

Name: name of the Web Service method. In case of mapping one-to-many calls the broker using the DOM creates a new XML document which contains all the responses together in one XML document. In order to know which response is from which Web Service method, the broker separates each response with the value of the name attribute.

SoapAction: SOAP action header of the Web Service method. Each Web Service request has its own SOAP action header. Thus when the broker sends the request to the client it replaces the client request’s SOAP action header with this one.
<?xml version="1.0" encoding="utf-8" ?>
<adapters>
  <ClientInterface SoapAction="http://tempuri.org/GetProductDetails">
    <ServerInterface
      url="http://localhost/webservicesupdatedbeta2/ProductDetailsService.asmx"
      name="GetProductQty" SoapAction="http://tempuri.org/GetProductQty"
      file="http://localhost/XSLTLib/ProductQty.xsl">
      <PreTransformation> </PreTransformation>
      <PostTransformation> </PostTransformation>
    </ServerInterface>
    <Response file="http://localhost/XSLTLib/MergedProducts.xsl">
    </Response>
  </ClientInterface>

  <ClientInterface SoapAction="http://tempuri.org/BiAdd">
    <ServerInterface
      url="http://localhost/anand/tempasp/BiAddService.asmx"
      name="BiAdd" SoapAction="http://tempuri.org/BiAdd"
      file="http://localhost/XSLTLib/ChgParam.xsl">
      <PreTransformation> </PreTransformation>
      <PostTransformation> </PostTransformation>
    </ServerInterface>
    <Response file="http://localhost/XSLTLib/responseadd.xsl">
    </Response>
  </ClientInterface>
</adapters>

Figure 4.1: Guiding Specification for XSLT based adaptation

*File:* This contains the name of the file for transformation. This file attribute maps the appropriate XSLT file from the XSLT libraries. Once the broker reaches the file attribute it maps the name with the XSLT libraries, when found it applies the transformation on the source XML document, which would be the client’s
The Response element in the Guiding Specification is another child of the ClientInterface element. The broker sends as many requests as the number of ServerInterface elements and possibly receives the same number of responses. After receipt, all the responses are merged into one XML document separated by their respective Web Service method name. The file attribute in the Response element contains another XSLT file that is responsible for transforming this merged XML response document into a XML embedded SOAP response packet, which is compatible with the client. The broker maps this file to the XSLT library, picks the file from the library and applies transformation on the merged XML document. After this is done the transformed SOAP response is sent back to the client.

Figure 4.2 demonstrates the broker and the Guiding Specifications with the help of a diagram; the client proxy sends a SOAP request to the broker that acts as a Web Service to the client. The broker which consists of the Guiding Specification has a list of XSLT documents. We discussed in the previous section that the Guiding Specification holds the addressing information of the actual Web Service, the name of the XSLT file to be invoked, a SOAP action attribute that adds header information to the messages sent and received to and from the broker. The XSLT libraries have a number of XSLT files that are mapped to the Guiding Specification.

4.3 Detailed Design

In this section we examine the detailed design of the broker and examine the important components of the adapter. Figure 4.3 demonstrates a diagrammatic representation of the adapter.

The Client UI is dependent on the client proxy since it calls specific methods on the proxy, although the communication is two-way, in the sense that the proxy returns data, the proxy is not aware of who is calling it and thus is not dependent on the client side user interface. In contrast the communication between the broker and the server is based on mutual awareness, so the communication dependency is two-way.

The Client UI invokes the client side proxy method that is created using the WSDL of a specific Web Service. The proxy marshals the request, creates a SOAP
4.3. Detailed Design

Figure 4.2: High Level Design of the Broker

The broker is created on the server side using the C# language. Once the request reaches the broker it loads the Guiding Specification document from the library, and it uses various Xpath expressions to access the elements on the Guiding Specification to compare the SOAP action header of the client request with that of a SOAP action attribute from the Guiding Specification and also to navigate to the next or previous element in the Guiding Specification. For example the Xpath expression is used to access information such as the name of the method, the name of the XSLT file to be invoked, and the SOAP action header. Once these are received it applies XSLT transformation based on the client request and sends it across to the server. In the case of accessing multiple Web Service
methods the broker creates as many request packets as required and sends them across. When the responses are received from the server it uses the DOM to create a new XML document for merging all the responses received from the Web Service. In case of more than one response the XML document is separated with a new XML element tag which is same as the name of the method. It receives this information from the Guiding Specification where the ServerInterface element has an attribute called name. Once a new XML document is created on the memory it applies another XSL transformation based on the XSLT file from the Response element of the Guiding Specification. This new XML embedded SOAP response is finally sent via HTTP that is understood by the client.

The implementations using a broker also involved mapping one-to-many calls in addition to the examples described previously. These mainly involved mapping one-to-one calls with differences such as parameter, method names, number of arguments passed. In this section we consider the mapping of one-to-two calls as a special case of the one-to-many mapping. The problems involved in accessing multiple services (mapping one-to-many calls) using SOAP extensions are solved as:

- SOAP extensions are not required; we have been using SOAP extensions on the client side proxy that acts as a proxy component between the client and the Web Service. SOAP extensions were used to access the SOAP messages which were sent by the client side proxy. Now that we have our own proxy component which acts as an intermediary between the client and the Web Service, SOAP extensions are not required.

- As SOAP extensions are not used we are able to use more than one XSLT to transform one XML document, hence able to utilise multiple existing Web Services.

- The broker acts as a standard Web Service to the client; it sends more than one request to the final Web Service, it sends requests until it receives all the required information for the client.

- Once the broker receives all the required information, it marshals all the responses into one SOAP response packet and sends it to the client.

- These factor mean that the Guiding Specifications supports syntactic as well as semantic adaptation.
Figure 4.3: Detailed Design of the Broker

Figure 4.4 can be explained as thus: the client requires the `GetProductDetails(int id)` method which returns a structure of the product's `name`, `image`, `price` and `quantity`. But the available Web Service provides four different methods one each for the `name`, `image`, `price` and `quantity`. A client proxy is created using the WSDL for the Web Service. As the WSDL pertains to the Web Service, the client proxy which is created also contains the four Web Service methods provided `GetName(int id)`, `GetImage(int id)`, `GetPrice(int id)` and `GetQty(int id)` and would not have any method called `GetProductDetails(int id)`. So when the client invokes this method using the instance of the client proxy, it would obviously not find any `GetProductDetails(int id)`. Thus we write a method on the client proxy called `GetProductDetails(int id)` which has similar implementation of the `GetName`, `GetImage`, `GetPrice` and `GetQty` methods.

The client invokes the method `GetProductDetails(int id)`, the client proxy marshals the client request, creates a SOAP packet and then sends it to the Web Service. It sends a SOAP message to the Web Service to receive a SOAP response which is translated to the native language by the .Net runtime. In
Figure 4.4: Server Side Adaptation to access multiple services

the above scenario the client needs to access four web methods to satisfy the required information. The client proxy sends one SOAP request to the broker as though it is the required service, the broker then acts as a client to the actual Web Service. As soon as the request reaches the broker it takes in the data (parameters passed in the method), creates four new SOAP requests using XSLT which are `GetName(int id)`, `GetImage(int id)`, `GetPrice(int id)` and `GetQty(int id)` and sends it to the Web Service. The `id` passed is the data that is received from the client. As the broker sends in four SOAP requests it possibly receives four SOAP responses from the Web Service. The broker applies another XSL transformation to merge these four responses into one that is compatible to the client, and finally the merged response is sent to the client and looks as though the response is coming from the Web Service. The differences between adaptation using SOAP extensions and this one are:

- Use of a broker instead of using SOAP extensions.
- The broker acts a Web Service to the client and acts as a client to the actual Web Service.
- Use of more than two XSLT files for this kind of adaptation. In this case we
use four XSLTs for transforming the client request into four and one XSLT for merging the response to be sent to the client.

We have already examined XSLT where we change the name of methods (Scenario 2). Thus the XSLT for creation of one client request to many Web Service calls would be similar to what we saw earlier. Here is an example of a merged XML document created by the broker for the responses received from the Web Service.

```xml
<merged>
  <GetName>
    <soap:Envelope
      xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:xsd="http://www.w3.org/2001/XMLSchema">
      <soap:Body>
        <GetNameResponse xmlns="http://tempuri.org/">
          <GetNameResult>Gear</GetNameResult>
        </GetNameResponse>
      </soap:Body>
    </soap:Envelope>
  </GetName>
  <GetImage>
    <soap:Envelope
      xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:xsd="http://www.w3.org/2001/XMLSchema">
      <soap:Body>
        <GetImageResponse xmlns="http://tempuri.org/">
          <GetImageResult>gear.bmp</GetImageResult>
        </GetImageResponse>
      </soap:Body>
    </soap:Envelope>
  </GetImage>
</merged>
```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <soap:Body>
    <GetPriceResponse xmlns="http://tempuri.org/">
      <GetPriceResult>15.0</GetPriceResult>
    </GetPriceResponse>
  </soap:Body>
</soap:Envelope>

<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <soap:Body>
    <GetQtyResponse xmlns="http://tempuri.org/">
      <GetQtyResult>11</GetQtyResult>
    </GetQtyResponse>
  </soap:Body>
</soap:Envelope>

It's seen that a merged element is created to show that it is not the final SOAP packet to be sent to the client, though it contains all the information needed for the client. However this needs to be processed in such a way that elements such as merged, GetName, GetImage, GetPrice, GetQty do not appear and these are replaced by GetProductDetails as this is one which the client understands, and also the data needs to be extracted from these four responses and should be made one under GetProductDetails. Following is the XSLT that takes care of the above factors.
4.3. Detailed Design

<?xml version="1.0" encoding="utf-8" ?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
xmns:tempuri="http://tempuri.org/"version="1.0">
  <xsl:output method="xml" encoding="us-ascii" />
  <xsl:output indent="yes"/>
  <xsl:template match="/merged" >
    <soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
xmns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmns:xsd="http://www.w3.org/2001/XMLSchema">
      <soap:Body>
        <GetProductDetailsResponse xmlns="http://tempuri.org/">
          <GetProductDetailsResult>
            <xsl:value-of select="concat(GetName, ' ; ', GetImage, ' ; ', GetPrice, ' ; ', GetQty)"/>
        </GetProductDetailsResult>
      </GetProductDetailsResponse>
    </soap:Body>
  </xsl:template>
</xsl:stylesheet>

1. Match the root element merged
2. Replace the elements that the client understands
3. Extract data from the GetName, GetImage, GetPrice & GetQty
   and concat it with a separator

We have discussed how a Web Service could be adapted when adaptation is
applied on the client side proxy using SOAP extensions and adaptation using a
broker. At this point it is meaningful to say that for simpler scenarios like chang-
ing method names and mapping one-to-one calls both these types of adaptation
techniques overlap with each other.

In this chapter we examined the detailed design of the XSLT based adaptation
with the help of some simple scenarios. We also examined some of the prelimi-
nary work done on XSLT based adaptation in the previous chapter. A prototype
was made to demonstrate this preliminary work using some real-world scenar-
ios. But this chapter concentrated more on the detailed design and the Guiding Specification of the broker. How this Guiding Specification is useful and provides a generic runtime support during the execution of the adapter was examined. The example scenarios proved the workings of this generic Guiding Specification. Chapter 5 will present a real-world example using this Guiding Specification and the broker.
Chapter 5

Example of broker based Adaptation

In the previous chapter we examined the proposed design for adaptation of Web Services. We saw how adaptation is applied using a broker to adapt to a Web Service. This design was proposed to offer a generic runtime support which means that it can adapt to many Web Services. The Guiding Specification formed the core of the broker. This Guiding Specification has all the details necessary such as addressing information and XSLT library to map, in order to make adaptation easy and more generic. In this chapter we will demonstrate a real-world example of applying adaptation to Web Services. We will see how the Guiding Specification offers generic runtime support and thus makes adaptation easier.

The following example outlines the implementation of a demonstrator for our adaptation technique using the broker and design discussed in Chapter 3 and Chapter 4. Also discussed are examples of XML messaging systems such as Financial Information eXchange Markup Language (FIXML) [Cov00a] and Straight Through Processing Markup Language (STPML) [Cov00b]. The example considers a financial system and will look at the adaptation of services that use FIXML and STPML messaging formats. This example demonstrates an Online Share Trading system which is exemplary of those used in stock market trading. Two Web Services based on the share trading applications have been defined. There is a Web Service required by the client and service provided by the provider. The general functionality of these Web Services enable the buying and selling of
shares, providing transaction histories, listing of cancelled and outstanding orders and provide current price quotes. The example demonstrates how the provided Web Service can be adapted to suit the client’s needs.

5.1 Introduction to the Online Share Trading System

The demonstrator example is based on a share trading system similar to that used by the Australian Stock Exchange (ASX). The ASX appoints human brokers to act on behalf of the stock exchange to buy and sell stocks for clients. The stock exchange hosts a Web Service that allows brokers to buy and sell stocks online. Brokers use an online client integrated with the ASX. When clients request a share purchase; the broker service starts a business process that first checks the real time stock price from the ASX by performing a quote request. After determining the price, the broker’s Web Service checks the client’s account balance and if the client has enough available funds, the trade is executed, which is done by invoking the ASX Web Service. This depicts the use of a Web Service where the data travels to more than one place (one client accessing more than one Web Service). In this case the data travels from the client’s machine to the broker’s Web Service and finally to the stock exchange’s Web Service.

We have created a Web Service using the ASX example on the local network for demonstration purposes. A database that consists of all the stocks in the ASX was created. For realistic data the security code and rates were downloaded daily from the ASX web site. The database has been maintained using SQL Server 2000. In order to run the system the client logs in to the broker’s web site. The user interface has been kept simple with all the basic utilities needed by an individual to trade shares. The client can buy and sell securities, obtain quotes of stocks that will have the day’s high, low and present trading value and check order histories. The client needs to look at up-to-the-minute transaction records of cancelled, executed and outstanding orders. They can also get their own portfolio, which means that the client can retrieve the list of stocks he or she holds along with the quantity.

A list of client program utilities is as follows:

- Perform Buy order on a security
5.2 Motivation for this demonstrator example

This example demonstrates a real-world scenario where XSLT adaptation can be applied. In Chapter 3 and Chapter 4 we discussed the different ways for this kind of adaptation to be applied with simple scenarios to test individual cases. This online trading system is to provide a more sophisticated motivating proof of concept for adaptation using Guiding Specifications and XSLT libraries and that it can be implemented for a real-world scenario.

XML messaging over HTTP has been used for quite some time. Currently it is commonly used in the financial services industry where many XML specifications have been written to cope with inter-system messaging. Such technologies include STPML and FIXML. Comparison of these technologies indicates that there are many inconsistencies with using different markup specifications on different platforms. XSLT based adaptation performs transformations on XML messages hence can be used to address these differences. In this example there will be message transformations on the client message from a subset of FIXML to STPML. The STPML reply will also be transformed into a format that the client can understand.

Before we go to the demonstrator example we take into consideration a scenario without adaptation where a client buys/sells securities from his broker. The Web Service provided by the server uses STPML, similarly the client uses only FIXML. Other problems that may arise are differences in the functions provided.

- Perform Sell order on a security
- Obtain a history of transactions
- List executed orders
- List cancelled orders
- List outstanding orders
- Obtain a snap quote on multiple securities (high, low, buy rate and sell rate)
- Display of security name for a given security code.
by the server and functions required by the client. For that reason a list of func-
tions have been defined to be different for the client and the server so that XSLT
based adaptation can be easily demonstrated. There is a Web Service required by
the client and the Web Service actually provided. The interfaces to these services
are defined as follows:

Web Service required by the Client:

```c
struct snapquote{
    string securitycode
    string securityname
    int buyrate
    int sellrate
}
enum enordertype (executed, cancelled, outstanding)

struct order{
    string securitycode
    string securityname
    string type (buy, sell)
    enordertype ordertype
    int rate
    int howmany
}

struct stportfolio{
    string securitycode
    int rate
    int howmany
}

snapquote[] quote(string securitycode[])
order[] outstanding(int clientid)
order[] executed(int clientid)
order[] cancelled(int clientid)
order[] history(int clientid)
```
5.2. Motivation for this demonstrator example

```
stdportfolio[] portfolio(int clientid)

Web service provided by server:

struct snapquote{
    string securitycode
    int buyrate
    int sellrate
}

struct order{
    string securitycode
    string type (buy,sell)
    enordertype ordertype
    int rate
    int howmany
}

snapquote quote(string securitycode)
order[] outstanding(int clientid)
order[] executed(int clientid)
order[] cancelled(int clientid)
stdportfolio[] portfolio(int clientid)
```

The problems that may arise due to the differences in the functions provided by the client and the server are:

- The server does not support the `history` method required by the client: the client cannot retrieve the history of transactions done.

- The server does not support multiple quote requests within one call: the quote method in the client requires an array type and the quote in the server does not provide an array type. For this reason the client needs to make more than one call to the Web Service. Thus each element of an array needs to be a different call to the Web Service.
Chapter 5. Example of broker based Adaptation

- The structures used to pass messages are different: when sending requests to the server the client needs to add or delete one or more of its parameters in order to be compatible with the Web Service.

- The client requires the securityname field in the return value of the snapquote and order methods.

Perhaps the most interesting feature of the adaptation is the parsing of the securitycode array found in the client’s quote Web Service request and the construction of an array result by using multiple calls to the server’s actual single quote Web Service. Also, the notion of the adapter being able to construct new functionality is interesting. This is shown in the client’s history request. The Web Service can use the server’s outstanding, executed and cancelled Web Services to create the history result for the client.

Solution

Guiding Specifications were discussed in Chapter 4, here also we have a similar Guiding Specification which abides to a XML schema and solves the above problems. Figure 5.1 shows a basic structure of the Guiding Specification used for this example.

- PreTransformation element contains the location of the local file, which holds the XSL stylesheet. The stylesheet is applied to the client’s request and the result is sent to the server (specified in the URL element, with the Soap Action value defined in the respective element).

- PostTransformation element has the location of the local file, which holds an XSL stylesheet that performs transformations on the response from the server. This element is useful to match the interface that the client expects. This transformation is only applied to the server’s result.

- PostAction specifies that the client request will be a posted XML message instead of a Web Service request. The XpathExpression attribute allows the unique identification of the client’s request. Any XML based message that the adapter receives will be evaluated against all ClientInterface XpathExpression elements. The first XPathExpression that results in true when evaluated will be used as the unique specification for that message.
5.2. Motivation for this demonstrator example

<Adapters>
  <ClientInterface SoapAction="" PostAction="" XpathExpression="">  
    <ServerInterface name="">  
      <SoapAction/></SoapAction>
      <PreTransformation/></PreTransformation>
      <PostTransformation> </PostTransformation>
      <URL> </URL>
    </ServerInterface>
  </ClientInterface>
</Adapters>

Figure 5.1: Structure of the Guiding Specification

- PostAction contains the value attached to the URL request. This is attached to the HTTP payload of the resulting outbound server message. This is used when the XML message is not a SOAP packet. This is useful where the messages are in the format of FIXML or STPML. It is the same case for an XPathExpression.

- Dependencies: There may be cases where one Web Service request is dependent on another Web Service response (Composition Chapter 6). The Dependencies tag is used to specify dependencies for the specific server interface. This allows multiple server calls to be made before creating the final server request. For example, if a client request is calling a procedure that requires an id number of an element in the database, but the server interface requires a name instead (database: table (id, name)), then a dependency may be used to first map the id to a name. Another Web Service would be needed to provide the name given an id, which would then be specified, in the dependency section. Overall, the broker would process the dependency first (taking the client’s id, sending a request to the mapping
Chapter 5. Example of broker based Adaptation

Web Service, retrieving the name instead) then given the name, create a Web Service request to the server.

The elements ClientInterface, ServerInterface & SoapAction have been discussed in the Chapter 4 of this literature.

5.3 Adapter Implementation Details

The differences in the client and the server have been discussed, now it is up to the adapter to match the interfaces of the client and server. In this section how the adapter matches and creates a compatible interface between the two is discussed. Some sample XSLT scripts and XML messages are also shown in order to explain them clearly. As it is not feasible to show all the Web Service methods in this literature, only one of the Web Service method is taken that is more challenging and interesting for adaptation.

The Web Service required by the client is: order history(int clientid). The server does not provide such a Web Service method. However the server does provide the following Web Service methods:

order[] executed(int clientid)
string securityCodeToName (string securitycode)

The client expects a security name along with the securitycode, buy rate, sell rate. But the Web Service does not have a securityname as a member however it provides a method to return the securityname when securitycode is passed as a parameter. Thus in order for the history method to be satisfied the adapter needs to call the executed method which provides with the buy and sell rates for all the securities of the specific client, and then it also needs to invoke the securitycodeToName Web Service to return all the names of the securities. This is a sample XML message sent from the client when history method is invoked:

<soap:Envelope xmlns:tempuri="http://tempuri.org/"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
5.3. Adapter Implementation Details

```xml
<soap:Body>
   <executed xmlns="http://tempuri.org/">
      <clientid>A0001</clientid>
   </executed>
</soap:Body>
</soap:Envelope>

Following is the Guiding Specification for the executed method:

```xml
<ClientInterface SoapAction="http://tempuri.org/executed">
   <ServerInterface name="executed">
      <SoapAction>http://tempuri.org/executed</SoapAction>
      <PreTransformation>
         http://localhost/XSLTLib/executed_pretransformation.xsl
      </PreTransformation>
      <PostTransformation>
         http://localhost/XSLTLib/executed_posttransformation.xsl
      </PostTransformation>
   </ServerInterface>
</ClientInterface>

The PreTransformation element calls the executed_pretransformation.xsl: In order for brevity only the SOAP body is examined.

```xml
<soap:Body>
   <DSCRIPT function="inject">
   </DSCRIPT>
   <SoapAction>http://tempuri.org/executed</SoapAction>
   <PreTransformation>
      http://localhost/XSLTLib/executedpretransformation.xsl
   </PreTransformation>
   <PostTransformation>
      http://localhost/XSLTLib/executedposttransformation.xsl
   </PostTransformation>
</soap:Body>
```
DSCRIPT is a scripting language that was introduced to communicate with the adapter after XSL transformations. Because XSL is limited in that it can only transform documents, a small language was needed to dynamically specify procedures after transformations have taken place. After each transformation has taken place (pre/post transformations) the adapter looks for a DSCRIPT element within the resulting document. If this tag is found, the broker then performs the required procedure.

An example of a DSCRIPT element is below:

```xml
<DSCRIPT function="inject">
  <url>http://localhost:8080/getNumbers/Service1.asmx</url>
  <SoapAction>http://tempuri.org/square</SoapAction>
  <PreTransformation>http://localhost/XSLTLib/square/pretransform.xsl</PreTransformation>
  <PostTransformation>http://localhost/XSLTLib/square/posttransform.xsl</PostTransformation>
  <input><xsl:copy-of select="." /></input>
</DSCRIPT>
```

The DSCRIPT element looks much like another `ServerInterface` element. It has the same rules and restrictions as the Guiding Specification (multiple URL elements can be specified).

**Motivation for using DSCRIPT:** DSCRIPT allows adaptation to be somewhat generic. The Dependencies elements are useful for static adaptation, meaning the XSL programmer knows what to expect. However, if the document to be transformed has dynamic elements (such as an array of unknown size) then adaptation performed on these singular elements cannot be achieved with Dependencies.

An example of DSCRIPT is below:

This example details a client request of an array of numbers. Each of these numbers must be squared and the result placed in an array. However, the server interface doesn’t allow for arrays, only static integers.
Client: int[] square(int[] arrayOfInts);
Server: int square(int number);

As seen above, these interfaces do not match up and require adaptation. A separate request is needed for each element in the array to the server. The result must also be placed back into an array. Using generic XSL, this cannot be achieved, because XSL cannot programatically specify web requests. However, XSL can replace each of the elements in the array with a DSCRIPT element. Using DSCRIPT the broker can then interpret the result and make the necessary server requests. The following XSLT script is for creating the request to the final Web Service. This transformed XML document invokes executed method and passes clientid as the parameter.

Executedpretransformation.xsl

```xml
<soap:Envelope>
  <soap:Body>
    <executed xmlns="http://tempuri.org/">
      <clientid>
        <xsl:apply-templates />
      </clientid>
    </executed>
  </soap:Body>
</soap:Envelope>
```

The Web Service request that reaches the server will look like:

```xml
<soap:Body>
  <executed>
    <clientid>A001</clientid>
  </executed>
</soap:Body>
```
This request is sent to the ‘URL’ which is specified in the DSCRIPT function. In this case it would be:

http://plas2000.fit.qut.edu.au:8080/ShareTradingServer/ShareServer.asmx. Once this request is processed at the server, it responds with a SOAP envelope:

```xml
<soap:Body>
  <executedResponse>
    <executedResult>
      <order>
        <securitycode>AAC</securitycode>
        <ordertype>Buy</ordertype>
        <rate>0.6</rate>
        <howmany>100</howmany>
      </order>
    </executedResult>
  </executedResponse>
</soap:Body>
```

The above response cannot be sent to the client as it has no security name. Now the broker has to extract all the security code from the order array and send to the Web Service method (securityCodeToName) to get the names of the securities for the corresponding security codes.

The broker also transforms the security code to the corresponding name as required by the service. The last transformation before the end result reaches the client, creates an XML/SOAP packet which is compatible with the client. The struct order had no SecurityName in the server side but the client expected one SecurityName in order to display it in the datagrid. The broker extracted the SecurityName by invoking another Web Service method using the DSCRIPT function.

In the same way the Web Service sends the responses of the executed, outstanding and cancelled Web Service methods. The broker ‘collects’ these three responses, however if these responses are sent to the client directly the interface of the client would not match with that of the server. Hence there should be another transformation applied at this stage that combines all these responses into
one and generate a new XML/SOAP response which looks as though it has come from the server. At this stage the broker applies one more transformation on the combination of all the responses and creates one XML/SOAP packet which looks as though the server had one history method.

This example shows how XSLT adaptation can be applied to a motivating, more real-world example. It also demonstrates how additional functionality can be added by use of DSCRIPT. DSCRIPT is not discussed elsewhere as it is an implementation detail, however this demonstrates one way to achieve the adaptation required. To summarise this chapter it can be said that adaptation can be achieved for many Web Services where there are minor mismatches between the client and server. This has been effectively done using the Guiding Specification and the broker. The next chapter will focus on the evaluation of Guiding Specification and design.
Chapter 5. Example of broker based Adaptation
Chapter 6

Discussion of XSL based adaptation

Chapter 5 demonstrated some real-world scenarios where adaptation can be applied using the Guiding Specification and the broker. This chapter will evaluate this design and compare with the evaluation criteria discussed in Chapter 2 under requirements of component adaptation. This chapter will also discuss how adaptation can be applied using multiple brokers and how composition of Web Services is addressed using adaptation.

At the core, adaptation can be thought of as a transformation of the messages being passed between the client and the available services. In SOAP these messages are encoded in XML, hence it is the transformation of an XML message that is required. It has been shown that the Extensible Stylesheet Language (XSL) Transformations (XSLT), which is a language for transforming XML documents into other XML documents, [W3C99] [W3C00b] can be used for this purpose. Each SOAP message is considered as a separate XML document, then the adaptation required for that message can be described using XSLT.

Rather than using client specific plug-ins, transformation may be applied to the XML message directly. This uses the XSL transformation language to transform these messages. The advantage of this system is that the transformations become platform independent. The XSL transformations can be applied to all incoming and outgoing messages. Currently the system checks the message type of all messages to decide on what transformations to apply.
6.1 Why Web Services & SOAP?

In the previous chapters Web Services and SOAP were the center of discussion for XSLT based adaptation. In this section we discuss why these two technologies were used for adaptation as opposed to CORBA, Enterprise Java Beans (EJB) or .NET.

Web Services as reusable software components was also discussed. In the past and also in the present scenario .NET, CORBA/IIOP, and EJB object models have been used as models for the creation of reusable software components. Each of these models work well provided that cross platform integration is not required. It would be time consuming and prove expensive to integrate programs across these technologies. Web Services solves this problem through HTTP, a standard Internet protocol supported by almost all the platforms and XML, a platform independent and robust definition vocabulary. This is one of the reasons for choosing Web Services for adaptation.

CORBA, RMI and DCOM have successfully supported distributed computing on intranets but have failed to gain acceptance on the Internet. The reason for this is that such protocols have a high infrastructure requirement and also depend on an underlying object model. This is fine for intranets controlled by a single organisation but not the Internet as it infeasible that all cooperating parties would agree to use the same technology. SOAP on the other hand has been deliberately designed to have low infrastructure requirements. Thus it is considerably simpler than these other protocols but far more likely to gain acceptance on the loosely coupled and rather chaotic Internet. Web Services use HTTP and XML for communication over the web. HTTP and XML are widely accepted standards supported by many platforms. Web Services can be written on many platforms, and clients running on different platforms can call it. For example a client written in Java using Java Beans can call a Web Service written in C# using .NET, this is because SOAP uses XML as a common intermediate language for communication. Thus adaptation applied to the XML messages reduces the number of adapters required. SOAP serves to define a specific standard that a Web Service requests and responses must adhere to. In this literature we proposed the adaptation of SOAP messages. Figure 6.1 shows the basic difference between the adaptation to SOAP messages and adaptation to messages on EJB or CORBA.

In Figure 6.1 a client written in .NET requiring a service written in EJB needs to route all the messages through an adapter that is aware of the different
6.2 Evaluation

Figure 6.1: Difference showing the adaptation done on SOAP envelope component models. In a similar manner there is possibly more than one adapter for the same client/server. This is because of the fact that there exists a need for transforming the original client message into an intermediary format before any kind of data adaptation is done. In these types of cases the adaptation required would be for two purposes:

- Adaptation for language or platform dependency into some intermediary format.
- Adaptation due to the differences in the data between the client’s requirements and service provided or syntactical differences.

It is clear from Figure 6.1(B) that adaptation applied on SOAP messages would be far easier and less time consuming than adaptation to messages between .NET, EJB and CORBA. This is simply because SOAP messages are platform and language independent.

6.2 Evaluation

Web Services adaptation can address differences in platform and user requirements. This adaptation proposed above can address differences in data and content of SOAP messages which can be transformed using XSL transformations. The broker can also be used for behaviour adaptation. For example it is possible to make multiple calls to an existing Web Service to allow sets of results to be requested rather than single results only. It can also take care of the presentation adaptation where the XML is to be transformed into HTML using XSLT in order to display it in a web browser in a different way, however this has not been
tested. The current design focuses on data adaptation. For this kind of adaptation different strategies can be used like adaptation using SOAP extensions or adaptation using a broker. Our proposed adaptation addresses adaptation using a broker that performs the following steps:

- Collects requests from the client
- Processes requests (transforming SOAP messages by selecting appropriate XSLT documents)
- Sends transformed requests to the server
- Collects responses from the server
- Processes responses (transforms SOAP responses that are compatible to the client)
- Send the transformed response to the client.

The requirements of Component Adaptation discussed in Chapter 2 are evaluated for this adaptation technique. In this section the term ‘Component’ is to be read as ‘Web Service’

1. **Transparent:** The adaptation mechanism is transparent to both the client and the component. The client has very little knowledge of the adaptation being applied to the component that the client application requires. This adaptation technique is not completely transparent because the developer needs to modify the client side as discussed in Chapter 4

2. **Black-box:** The adaptation here is black-box in the sense that the adapter/broker has little knowledge of the internal specification of the component, in this case the Web Service. There is little knowledge because the adapter needs to know the name of the Web Service method in order to add in the Guiding Specification.

3. **Platform-independent:** For a given component the adaptation mechanism should not depend on any particular platform. This adaptation technique has been tested in Microsoft’s .NET platform however tests have not been conducted on different platforms such as J2EE. However the basic design is constituted by the Guiding Specifications and XSLT libraries which
are platform independent. Taking this into account this adaptation mechanism is platform independent.

4. **Language-independent:** For a given component the adaptation mechanism should not depend on any particular language. As the adaptation mechanism is based on transformation of SOAP messages embedded in XML using XSLT, it does not depend on any language.

5. **Composable:** The adaptation technique proposed here is composable in the sense that the component would behave in the same way as before the adaptations were applied to them and there is no change in the original component because of this adaptation. This also takes into account the supporting of multiple adapters where the one adapter passes the client request to another adapter for further processing.

The adaptation described in this thesis meets all major requirements as listed. It has been successful in achieving these goals. It also provides a generic base for the specification of adapters. However there are also some drawbacks to this approach that are identified as below:

- For every transformation to take place, XSLT has to be manually written. The XSLT files are not generated on the fly.
- For every adaptation to a Web Service a new entry has to be manually added in the Generic Specifications.
- Major differences in formats cannot be handled. For example, Transactions.

**Comparison with BizTalk**

In this literature the adaptation, which is called XSLT based adaptation, also investigated similar scenarios with the only difference being that the XSL transformations have to be done manually which depends on Guiding Specifications provided to the application designer. The main goal of XSLT based adaptation to date is overcoming data translation problems like data mapping, accessing multiple Web Services in one method call, accessing multiple web servers in the same call, composition of Web Services where the response of one method call is fed into the second Web Service request. In data translation our adaptation
technique does not take into consideration the use of multiple adapters whereby the adaptation is done on the adapted component.

This section discusses some basic differences between BizTalk and our adaptation technique.

**Generation of XSLT:** In BizTalk the creation of XSLT is dynamic. The user is required to map the source and destination fields using the Mapper tool, after which it is compiled. After compilation, an XSL transformation is dynamically created depending on the mapped source and destination fields. In our adaptation technique a Guiding Specification is created according to the differences between the client and the server. The user adapting to a particular Web Service writes an XSLT manually and specifies the application of the transformations in the Guiding Specifications. Thus the creation of XSLT is not dynamic.

**Data type Mapping:** In both the adaptation techniques data type mapping is possible. Again the mapping in BizTalk is done by the Mapper tool, functoids also help the user in the mapping of data. In our adaptation technique the data mapping is specified in the Guiding Specifications and the XSLT is generated according to the rules specified in the Guiding Specifications. However there are other minor differences in both the techniques. Apart from XML, BizTalk also supports data formats such as EDI and flat files. It does not support Value Added Networks (VAN) for transport, however EDI parser supports X12 and EDIFACT envelope formats thus users can use their existing EDI investments and transform the data to and from XML. Thus all the EDI documents need to be transformed to XML before applying XSLT for transformation. Thus needs an intermediate format (in this case XML) before the final data mapping can be done. The problems that can arise in transforming EDI documents to XML are discussed in the later section of this chapter. Our adaptation technique supports only XML documents.

**Security:** Security is one important factor in intranets and internets. As our adaptation technique concentrates on Web Services, adapting to another Web Service may require the use of different security mechanisms. We have not yet deployed any security mechanisms for this adaptation, however this has been considered for the future. BizTalk employs security mechanisms for adaptation, and uses the Windows 2000 security Model including PKI for signing, encryption and decryption of messages.

**Support of Web Services and SOAP:** Both the adaptation techniques
support Web Services and SOAP. In BizTalk, Web Services is supported within
in the BizTalk Orchestration environment where a business process can directly
access a Web Service as an activity in that process using the SOAP Toolkit 2. In
our adaptation technique SOAP is built under .NET.

Support for DTDs/ Schemas: As the XSLT based adaptation is based on
XML messages using SOAP and transformation of these messages using XSLT,
it supports DTDs as well as XML schemas. We discussed that this adaptation
technique does not support other file formats thus the question of implementa-
tion for other formats does not arise. However in BizTalk, Microsoft have made
available a number of schemas for X12 and EDIFACT but do not provide a full
complement for them.

6.3 Composition of Web Services

There are Web Service composition languages like XLANG [Tha01] and Web
Service Flow Language (WSFL) [F.L01] but these were replaced by Business
Process Execution Language for Web Services (BPEL4WS) [SS02] which was
released by BEA, IBM, and Microsoft in July 2002. This is used to describe
executable business process which relies on the import and export of Web Services
exclusively.

When considering Composition of Web Services there are two scenarios. The
first one is where the response of one Web Service method is to be fed into the
request of the second Web Service method, essentially creating a chain of requests.
The second type of composition is where multiple adaptations are required for a
single request.

We first examine composition of Web Services where the response of the first
web method is fed into the request of the next web method. This type of adap-
tation aims at composition of Web Services where the SOAP response of one
request is to be fed into the request of another SOAP request. This basic archi-
tecture is similar to the one that was mentioned in Chapter 3 except that this
one creates an on-the-fly XML (SOAP responses) document internally of all the
responses received from the Web Service and also the initial Web Service request
from the client. In order to explain this we will modify the example examined in
Chapter 3.

In the scenario depicted in Figure 6.2, four Web Service methods satisfy one
client request. This means that the adaptation is necessary to map this one call to four calls which we have already examined in the last chapter. But this one is different in the sense that not only the parameters for each web method are changed but a new parameter is added to the next web method and the name of the methods are changed. This means that the state of all the Web Service responses is preserved and the data received as a response from the Web Service is added into the next Web Service request.

The client requests for GetProductDetails with an id as a parameter, the broker marshals the request (as seen in Chapter 4) and creates a new request for GetProductName passing the same id as the parameter. It now waits (synchronized) for the response to come in before sending the next request. The next web request GetProductPrice takes in the parameter id and also an additional parameter name to get the price, but the broker has no information about the name. This means that the response received from GetProductName has to be fed into the request of GetProductPrice as one of the parameters apart from id. Thus the broker takes the response data of GetProductName which possibly returns the name of the product and feeds this name into the request packet of the next web request along with the id which the client had sent. Unlike the examples seen in Chapter 4 where the broker has to consult a single SOAP packet, which is the client request, in this case the broker has to extract data from more than one XML document, one from the client request. In this case the data is the id and from the response packet the name returned by the Web Service method GetProductName. The broker does the same processing for all the other remaining methods GetProductImage & GetProductQty. For the former it takes the data from GetProductPrice, GetProductName and the client request, for the latter the broker extracts the responses from all the above Web Service methods and also the request.
6.4. Composition of Web Services using multiple brokers

The working of the composition using adaptation uses the same architecture that was explained in the earlier chapter, however there are some modifications to the broker. The modifications are: the broker is built such that all the responses and the client request are stored in a separate XML document. As the request stage progresses the broker retrieves the relevant information (data from the response) from this XML document, which is stored temporarily.

The initial client request is the non-transformed SOAP message, which is received by the adaptation layer from the client. The adapter maps this with the XSLT (specified in the XML specifications) and extracts the data from the SOAP message and sends the transformed SOAP message to the Web Service method GetProductName, which takes in only one parameter. Once the response is received from the Web Service it is stored in the memory. Now the broker applies another XSLT document to extract the id from the initial client request and the response data (name) received from the Web Service to create a new SOAP request message to be sent to Web Service method GetProductPrice, which takes two parameters (id & name). This process goes on until the broker finishes all the ServerInterface elements in the Guiding Specifications.

6.4 Composition of Web Services using multiple brokers

The second view of composition of Web Services is adaptation using more than one broker. The idea here is that for the client to be satisfied the broker sends some part of the request to the final Web Service and the remaining to another broker which in turn receives responses from another Web Service. The first broker uses the second broker as if it were any other Web Service and for the second broker the first one is equal to any other client.

In all the cases examined investigations were done using only one broker. There have been instances where this one broker invokes methods from one Web Service and also from different Web Services. However there may be quite a few instances where this one adapter itself cannot satisfy a client. It may need to invoke methods from different Web Services which is possible from one broker but it might not have the required information for the other Web Service. In these cases the first broker may require the services of some other middle layer or any other broker that would have the required information to access or invoke
The basic motivation for adaptation using multiple brokers is reusability and efficiency. As it is known that this architecture of XSL based adaptation uses XSLT for transformation of requests, but is not created on-the-fly, it means that the user who wants to adapt to a Web Service should manually create one XSL transformation. Thus when the user knows about the location, name, and parameters of a Web Service they can create an XSL transformation based on the Guiding Specification. However, if somebody else has already created a broker for this Web Service that has the same behavior, then the very purpose of reusability and efficiency is defeated. Thus the user has to simply locate the addressing information of the broker which does this and specify them in the Guiding Specification. This might be one case where multiple brokers may be used. The second case would be where the original broker has no knowledge of a Web Service but has located the addressing information of a broker that can satisfy it’s needs. These are some of the cases where there might be a basic need for using multiple brokers, however we examine a scenario that might explain this more clearly.

In Figure 6.3 the client requires a method, \textit{GetProDetails\text{\texttt{}}(id)} which might be satisfied with more than one Web Service where the location of one Web Service is different from the other. Also as the client’s one request contains information such as \texttt{name}, \texttt{age}, \texttt{qty} and \texttt{price}, adaptation is necessary at the middle layer, out of which Web Service 1 provides \texttt{qty} and \texttt{price} and Web Service 2 provides the \texttt{name} and \texttt{age}. Under normal circumstances Adapter 1 can marshal the client request into four method calls using XSLT and send these to different locations. But this adapter provides \textit{GetQty}, \textit{GetPrice} and \textit{GetCustDetails} which is a struct of \texttt{name} and \texttt{age}. Thus Adapter 1 cannot understand \texttt{name} and \texttt{age} and the Web Service 2 cannot understand \textit{GetCustDetails}. In this case there needs to be another middle level layer to the existing middle level layer in order to marshal \textit{GetCustDetails}

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure6_3.png}
\caption{Adapting to a Web Service with more than one broker}
\end{figure}
into \textit{GetCustName} and \textit{GetCustAge} which Web Service 2 provides. This means that Adapter 1 acts a client to Adapter 2, which in turn acts as a service to the former. Adapter 1 knows with whom it is communicating but thinks as though it is the final Web Service.

### 6.5 Asynchronous calls for accessing multiple services

Synchronous calls can be compared to the students who stand in a queue to receive a student card. The service assistant serves the students one by one, when serving one student the service of other students are blocked. Thus in technical terms, when a synchronous call is made the calling thread is blocked and has to wait until the method completes. But on the other hand asynchronous calls allow more parallelism. In the case of enterprise world asynchronous processing becomes necessary where systems need to handle many requests in a very quick time. However it is left up to the developer when to use synchronous and asynchronous processing of method calls.

Web Services support both synchronous and asynchronous communication between the client and the server. We have already seen in the previous chapters how synchronous processing works on Web Services. In this section we examine invoking Web Service methods asynchronously. The issue behind asynchronous processing is that as the calling process does not wait for the delivery of the message it may never hear about certain error conditions. However asynchronous processing deals with problems relating to bandwidth constraints and network latency, which is necessary in an enterprise world.

Even with the advancements implemented in .NET framework, successfully developing asynchronous programming logic is not trivial. Thus an application developer needs to determine when the use of asynchronous events is beneficial. In our previous examples asynchronous processing would be reasonably efficient except for the cases that were observed in composition. In the case of composition where the response of one call is to be fed into the next web method call, the adapter has to wait for one response to be received before it can process the next request. In this type of case asynchronous calls would not be possible.

With reference to existing work done on asynchronous Web Services, a simple asynchronous Web Services Architecture [New03] has a asynchronous SOAP
message transport layer that handles transportation of SOAP messages irrespective of a connection is available or not. It considers each SOAP message as a file and stores in the SOAP messaging layer and later whenever the connection is available the SOAP message is transferred immediately and executed.

In Chapter 4, Figure 4.4 we examined where the broker was able to access multiple services to satisfy one client request. In this section we take the same scenario, the only difference being that the calls here would be made asynchronous. This means that all the calls are invoked at the same time, unlike in Chapter 4 example where the broker waits for the response to be received from the first Web Service before processing a request to the second Web Service. In this case all the Web Service requests would be sent to the Web Service at the same time, and possibly the response from those Web Services would also be received at the same time.

This adaptation is also mapping one-to-many calls except for the fact that the broker processes all the web requests at the same time and possibly all the web methods are invoked at the same time.
Chapter 7

Further work & conclusion

As we reach the end of this thesis it is worthwhile to discuss what further work can be done in this area and how the proposed Guiding Specification can be used effectively to make adaptation more generic.

7.1 Future work

We have been using XSL for transformation of XML documents. XML Query Language (XQL) is a notation for addressing and filtering the elements and text of XML documents. XQL is a natural extension to the XSL pattern syntax [Spe01]. Both XSL and XQL can be used for query and transformation of XML documents. Both approaches are block structured and template oriented. Both offer the ability to return trees or graphs, create new elements in the output and query XML. The biggest difference between them is the syntax, however both are useful in its own right. We take a simple example to illustrate the differences in XSL and XQL.

XML document:

```
<Address>
  <StreetNo> 2-66 </StreetNo>
  <StreetName> Pear Street </StreetName>
  <Suburb> Greenslopes </Suburb>
  <State> QLD </State>
</Address>
```
Using XSL we select Suburb in Address where State is 'QLD':

```
<xsl:for-each select = "Address[State = 'QLD']/Suburb">
    <xsl:value of />
</xsl:for-each>
```

Selecting the above using XQL:

```
WHERE <Address>
    <State> QLD </State>
    <Suburb> $a </Suburb>
</Address>
CONSTRUCT $a
```

In the above we can see that the basic functions derived from XSL are present in XQL as well. So can the XQL be used as a transformation language instead of XSL for our adaptation technique? XQL can be considered as future work for this adaptation work. At present we were not able to use this because as at today, .NET doesn’t support XQL, and as of now XQL is not a W3C standard.

However XQL can be considered as a language for transformation of XML documents for the future. It is likely that .Net might support XML Query language in its future implementations.

At present our adaptation is ‘functional’. We do not take into account issues such as transactions and distributed transaction management. For example if one Web Service is refactored into two, the two resulting calls may need to be made into an atomic transaction. Another important issue is security. Adapting one Web Service to another may require the use of different security mechanisms. Indeed we may need to adapt the actual security mechanisms.

Finally, the construction of XSL transformation is manual. This means that once the client requirements and the service provided by the server have been identified, the differences if any, are mapped through the Guiding Specifications and an XSL transformation is manually written for the transformation of XML documents. What we would really like is to have these XSL transformations created dynamically, and the differences identified in the Guiding Specifications. However this cannot be done for all the cases, but could be done for common cases like Data Manipulation and changing of method names (Chapter 3). Work could easily proceed on refining tools to support the generation of adapter specifications. The strength in the adaptation approach presented is its generality.
7.2 Conclusion

The specifications used are efficient in construction of adapters, and tool support would make this form of adaptation even more attractive.

7.2 Conclusion

This thesis has demonstrated an approach to the adaptation of Web Services, in particular where there is a reasonable overlap in structure and behaviour of the required and provided Web Services. The motivation for applying adaptation to Web Services arises from the benefits of adaptation of Software Components. This includes a higher level of reuse of Software Components, and further promotion of general benefits of Component Based Software Engineering. Adaptation is generally considered as an integral step in Component Based Software Engineering. Software Components and Web Services both communicate via well defined interfaces, hence applying adaptation to Web Services appears to be equally beneficial.

The demonstrated approach to adaptation uses XSLT documents to modify the messages passed between Web Services to achieve the desired adaptation. The transformations required for a particular adaptation are defined in specifications written in XML. The runtime elements of the adapter use these Guiding Specifications to guide the application of the XSLT documents to the Web Service messages. This thesis has demonstrated the feasibility of this approach by the implementation of the demonstrator example.

The use of XSLT and the Guiding Specifications written in XML provide a complete specification of the required adaptations. The use of these higher level specifications allows the implementor to work more efficiently than if each adapter were to be individually coded. The underlying runtime system that uses the specifications and applies the XSLT relieves the implementor of much low level implementation detail. It has been demonstrated that the underlying runtime system can be used across a wide variety of scenarios. Thus the adaptation approach used is generic, yet allows the implementor to work at a higher level, hence is efficient in implementation. The implementation of the underlying generic runtime support is a significant achievement of this work.

The examples presented in this thesis demonstrate simple adaptation of Web Services, such as the transformation of data, for example the extraction of elements from structured arguments. The examples also demonstrate modification
of behaviour via adaptation of the Web Services, such as the refactoring of existing methods to compose more complex methods. The ability to provide new behaviour from existing methods is a distinguishing point for this work. In a more broad setting this adaptation mechanism could be used to provide completely new services composed from existing services. The simplicity of implementation could make this a useful tool in this domain, however this is a side effect not pursued in this work.

Finally, the benefits of asynchronous access to Web Services via an intermediate broker were investigated. The possibility of providing asynchronous access to multiple Web Services where no dependency exists between the invocations was investigated. While there seemed to be a potential for decreased processing time due to parallelism, difficulties in handling error conditions require further work. However this could lead to the provision of brokers solely for the purpose of efficiency via parallelism, or improved fault tolerance via redundant computation.

To conclude, this thesis demonstrated the ability to effectively use XSLT and XML to specify Web Service adapters using the Guiding Specification, and to apply these via a generic runtime mechanism. This is demonstrated by way of example. Further work in this area is likely to provide additional benefits.
Appendix A

Example XSLT scripts

This section demonstrates some of the XSLT scripts used in the example demonstrated in Chapter 5.

Executedposttransformation.xsl

```xml
<?xml version="1.0" ?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    xmlns:tempuri="http://tempuri.org/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    <xsl:output method="xml" encoding="us-ascii" />
    <xsl:output indent="yes" />
    <xsl:template match="/">
        <ArrayOfOrder xmlns="http://tempuri.org">
            <xsl:apply-templates />
        </ArrayOfOrder>
    </xsl:template>
    <xsl:template match="tempuri:order">
        <order>
            <xsl:apply-templates />
        </order>
    </xsl:template>
    <xsl:template match="executedResponse">
        <xsl:apply-templates />
    </xsl:template>
</xsl:stylesheet>
```
Appendix A. Example XSLT scripts

<xsl:template match="executedResult">
  <xsl:apply-templates />
</xsl:template>

<xsl:template match="tempuri:securitycode">
  <securitycode>
    <xsl:value-of select="." />
  </securitycode>
</xsl:template>

<DSCRIPT function="inject">
  <url>
  </url>
  <SoapAction>
    http://tempuri.org/securityCodeToName
  </SoapAction>
  <PreTransformation>
    d:\\xsl\\securityCodeToNamepretransformation.xsl
  </PreTransformation>
  <PostTransformation>
    d:\\xsl\\securityCodeToNameposttransformation.xsl
  </PostTransformation>
  <input>
    <xsl:copy-of select="." />
  </input>
</DSCRIPT>

<xsl:template match="tempuri:ordertype">
  <ordertype>
    <xsl:value-of select="." />
  </ordertype>
</xsl:template>

<xsl:template match="tempuri:rate">
  <rate>
    <xsl:value-of select="." />
  </rate>
</xsl:template>
In the above XSLT a new request is created to be sent to another Web Service method (securityCodeToName) that takes a security code as a parameter.

securityCodeToNamePretransformation.xsl

```xml
<?xml version="1.0" ?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    xmlns:tempuri="http://tempuri.org/" version="1.0">
    <xsl:output method="xml" encoding="us-ascii" />
    <xsl:output indent="yes" />
    <xsl:template match="/"
        <soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
            xmlns:xsd="http://www.w3.org/2001/XMLSchema"
            xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
        <soap:Body>
            <securityCodeToName xmlns="http://tempuri.org/">
                <securitycode xmlns="http://tempuri.org/">
                    <xsl:apply-templates />
                </securitycode>
            </securityCodeToName>
        </soap:Body>
    </xsl:template>
</xsl:stylesheet>
```

The above XSLT makes no transformation to the SOAP envelope generated because of the request sent to the securityToCodeName Web Service.
SecurityCodeToNamePosttransformation.xsl

```xml
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    xmlns:tempuri="http://tempuri.org/" version="1.0">
    <xsl:output method="xml" encoding="us-ascii" />
    <xsl:output indent="yes" />
    <xsl:template match="/">
        <securityname xmlns="http://tempuri.org/">
            <xsl:apply-templates />
        </securityname>
    </xsl:template>
    <xsl:template match="securityCodeToNameResponse">
        <xsl:apply-templates />
    </xsl:template>
    <xsl:template match="tempuri:securityCodeToNameResponse">
        <xsl:apply-templates />
    </xsl:template>
    <xsl:template match="securityCodeToNameResult">
        <xsl:value-of select="." />
    </xsl:template>
</xsl:stylesheet>
```

executed_posttransformation.xsl

```xml
<?xml version="1.0" ?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    xmlns:tempuri="http://tempuri.org/" version="1.0">
    <xsl:output method="xml" encoding="us-ascii" />
    <xsl:output indent="yes" />
    <xsl:template match="/">
        <soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
            xmlns:xsd="http://www.w3.org/2001/XMLSchema"
            xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">  
            <soap:Body>
                <executedResponse xmlns="http://tempuri.org/">
                    <executedResult>
```
<xsl:apply-templates />
</executedResult>
</executedResponse>
</soap:Body>
</soap:Envelope>
</xsl:template>
<xsl:template match="order">
<order xmlns="http://tempuri.org/">
 <xsl:apply-templates />
</order>
</xsl:template>
<xsl:template match="securitycode">
<securitycode xmlns="http://tempuri.org/">
 <xsl:value-of select="." />
</securitycode>
</xsl:template>
<xsl:template match="securityname">
<securityname xmlns="http://tempuri.org/">
 <xsl:value-of select="." />
</securityname>
</xsl:template>
<xsl:template match="ordertype">
<ordertype xmlns="http://tempuri.org/">
 <xsl:value-of select="." />
</ordertype>
</xsl:template>
<xsl:template match="rate">
<rate xmlns="http://tempuri.org/">
 <xsl:value-of select="." />
</rate>
</xsl:template>
<xsl:template match="howmany">
<howmany xmlns="http://tempuri.org/">
 <xsl:value-of select="." />
</howmany>
</xsl:template>
</xsl:stylesheet>
Bibliography


[W3C00b] W3C. Xsl version 1.0, w3c candidate recommendation, November 2000.