Approaches and Best Practices in Web Service Style, Data Binding and Validation

Asif Akram and David Meredith
(a.akram@dl.ac.uk, d.j.meredith@dl.ac.uk)
CCLRC e-Science Centre, Daresbury Laboratory, Warrington, Cheshire, WA4 4AD, UK

TABLE OF CONTENTS

1. Overview

In this chapter we provide a critical evaluation of the different Web service styles and approaches to data-binding and validation, citing examples and recommendations based on our experiences in developing Web services for large-scale scientific facilities in the UK. We provide implementation examples using current SOAP standards for Java, including JAX-RPC and the newly released JAX-WS standard. We assess the advantages and disadvantages associated with 'loose' verses 'tight' data typing and when decoupling the binding / validation framework from the SOAP engine implementation. We also assess the different approaches to Web service development, citing both the 'code first' and 'contract driven' approaches, and outline some best practices for WSDL authoring / modularization. We show how the WSDL interface style (RPC / Document), strength of data typing and the level of support for data modelling, binding and validation must be carefully considered in order to successfully address the particular requirements of the application. As an example, data-centric services will usually require different approaches to Web service development when compared to more general services, for example, those designed for the delivery and sharing of generic files and documents. In the former scenario, a strongly typed and tightly bound Document orientated approach would be suitable, whereas a loosely typed RPC approach using MIME attachments could be more suitable for the latter. For the most part however, we recommend the use of the Document/ literal wrapped Web service style with a 100% XML schema compliant data-model that can be separated from the WSDL definitions. We found that this encouraged collaboration between the different partners involved in the data definition process and also assured interoperability. This also leverages the advanced capabilities of XML schema for precisely constraining complex data when compared to RPC and SOAP encoding styles.

Key words: Web Services, WSDL, Document, RPC, XML Schema, Data Binding, Validation, Scientific Applications

2. Introduction

A Service Oriented Architecture (SOA) is an architectural style whose goal is to achieve loose coupling among interacting software agents (services and clients). A SOA achieves loose coupling by employing two architectural constraints: 1) a small set of well-defined interfaces to all participating software agents and, 2) ensuring the interfaces are universally available to all providers and consumers. In the context of a SOA, a service is a function that is self-contained and immune to the state of other services. These services can communicate with each other, either explicitly through messaging, or by a number of master services that coordinate or aggregate activities together, typically in a workflow. In order to realise the principle of loose coupling, and to cope with the continual changes of how business and consumers use the Web today, the industry has converged on a new platform agnostic communication technology known as 'XML Web services.' This new Internet-based technology enables applications, machines, and business processes to work together and share information in a revolutionary way. Platform independence is achieved through the use of XML to define both common, interchangeable messages and service interfaces. The widespread support of XML assures that businesses will cooperate in the Internet-based economy. Web services can be described as:

> *A Web service is programmable application logic accessible using standard Internet protocols. Web services combine the best aspects of component-based development and the Web. Like components, Web services represent black-box functionality that can be reused without worrying about how the service is implemented. Unlike current component technologies, Web services are not accessed via object-model-specific protocols, such as DCOM, RMI, or IIOP. Instead, Web services are accessed via ubiquitous Web protocols (i.e. HTTP) and data formats i.e. XML).*

A Web service has five essential attributes [5]:

1) It can be described using a standard service description language (Web service Description Language, WSDL) [1];
2) It can be published to a registry of services, usually a UDDI (Universal Description, Discovery and Integration) [2] registry;
3) It can be discovered by searching the service registry;
4) It can be invoked, usually remotely, through a declared API;
5) It can be composed with other Web services allowing integration into complex workflows that may span multiple domains and organizations.

At the heart of Web service interoperability is the Basic Profile 1.0 (BP) [4], published by the Web services Interoperability Organization (WS-I). The BP provides a set of rules that govern how applications use common Web services related technologies. The first deliverable of the WS-I was the Basic Profile 1.0, which details how to use four primary Web service related specifications together: XML, WSDL, SOAP [3] and UDDI. The BP conformance rules define in concrete terms how to use these technologies to register, describe and communicate with Web services. The BP makes Web services interoperability practical, and coverage of it is a critical aspect of this chapter.

3. Web Service Style / WSDL Binding

It is important to understand how the WSDL binding ‘style’ and ‘use’ attributes dictate how a SOAP message is constructed and formatted into XML when transmitted over the wire. This has serious implications upon Web service data binding, validation and interoperability. Collectively, the process of generating SOAP messages according to different bindings is referred to as ‘SOAP encoding’ or ‘WSDL binding.’

- The WSDL binding ‘style’ attribute indicates whether a <wsdl:operation> is RPC-oriented (SOAP messages containing parameters and return values) or document-oriented (SOAP messages containing XML instance documents).
- The ‘use’ attribute dictates how the input and output data of a <wsdl:message> is serialised into XML.

Listing 1 shows how the binding style and use are both specified within the <wsdl:binding>.

3.1. The ‘style’ attribute

The optional style attribute specified within the <soap:binding> element defines the default style for each contained <wsdl:operation>. The style attribute has the value ‘document’ or ‘rpc.’ If no style is specified, it is assumed to be ‘document.’ If a different style attribute is specified on a contained operation, it overrides the <soap:binding> style. These two Web service styles represent the RPC-centric and Message-centric approaches. Much of the documentation available today focuses on the simpler RPC-centric viewpoint, which often gives the misleading impression that SOAP and Web services are just another way of doing RPC.

3.2. The ‘use’ attribute

The required ‘use’ attribute is specified within the nested <soap:body> element of the <wsdl:input> and <wsdl:output> elements and has the value ‘encoded’ or ‘literal.’

use = encoded

Each WSDL message part references an abstract type using the ‘type’ attribute. These abstract types are used to produce a concrete SOAP message according to the encoding rules located at the URIs listed by the ‘encodingStyle’ attribute. Each URI is listed from the most to least restrictive. The most common encoding is SOAP encoding, which is a set of serialization rules defined in SOAP 1.1 which specify how objects, structures, arrays and object graphs should be serialized into XML. Applications using SOAP encoding focus on RPC. According to the specifications, if the referenced encoding style allows variations in its format (SOAP encoding does), then all variations must be supported. However, most SOAP implementations ignore this recommendation and have limited support for variation.

use = literal

Each WSDL message part references a schema defined in the <wsdl:types> using either the ‘element’ or ‘type’ attribute. When the message part references an element, it will appear directly beneath the <soap:body> element in a concrete SOAP message (Document style) or under an ‘accessor’ element named after the message part (RPC style). When the message part references a type, the message part is itself defined by the referenced schema type. The optional ‘encodingStyle’ attribute is usually required only when the ‘use’ attribute has value ‘encoded.’

Listing 1 – The WSDL binding ‘style’ and ‘use’ attributes dictate how a SOAP message is constructed and formatted into XML when transmitted over the wire.

```
<wsdl:binding .... >
      <soap:binding transport="uri"? style="rpc|document"?/>
      <operation .... >
        <input>
           <soap:body parts="nmtokens"? use="literal|encoded"?
                    encodingStyle="uri-list"? namespace="uri"?>
        </input>
        <output>
           <soap:body parts="nmtokens"? use="literal|encoded"?
                    encodingStyle="uri-list"? namespace="uri"?>
        </output>
      </operation>
</wsdl:binding>
```

Using different combinations of the binding style and use attributes, different Web service implementation styles are possible. These are assessed in Table 1. Each style is further discussed in the following sections (Sections 4 to 5). It must be stated from the outset however, that the Document encoded style is considered a non-functional binding style and will not be commented upon further.

Table 1 - Advantages and Disadvantages of Each WSDL Binding Style

Style	Advantages	Disadvantages
RPC Encoded	• Simple WSDL • Operation name wraps parameters	• Complex types are sent as multipart references meaning <soap:Body> can have multiple children • Not WS-I compliant • Not interoperable • Type encoding information generated in soap message • Messages can't be validated • Child elements are not fully qualified
RPC Literal	• Simple WSDL • Operation name wraps parameters • <soap:Body> has only one element • No type encoding information • WS-I compliant	• Difficult to validate message • Sub elements of complex types are not fully qualified.
Doc Literal	• No type encoding information • Messages can be validated • WS-I compliant but with restrictions • Data can be modelled in separate schema	• WSDL file is more complicated • Operation name is missing in soap request which can create interoperability problems • <soap:Body> can have multiple children • WS-I recommends only one child in <soap:Body>
Doc Wrapped	• No type encoding information • <soap:Body> has only one element • Operation name wraps parameters • Messages can be validated • WS-I compliant	WSDL file is complicated – request and response wrapper elements may have to be added to the <wsdl:types> if original schema element name is not suitable for Web service operation name.

4. RPC Binding Style

The RPC binding style specifies that the <soap:body> of a SOAP request message contains an element named after the service method being invoked (the wrapper element). This element, in turn, contains <part> elements for each parameter (Section 7.1 of the SOAP specification). The wrapper element namespace is defined by the 'namespace' attribute of the <soap:binding> element. Each message part is named after the corresponding parameter of the call. Parts are arranged in the same order as the parameters of the call unless specified differently in the WSDL document by the 'parameterOrder' attribute. For a SOAP response message, the <soap:body> contains an element named after the service method being invoked with 'Response' appended to the name. This element contains a <part> element for the return value.

The following summary examines the key features of the RPC WSDL binding style and the format of a resulting SOAP message. Table 2 illustrates these key points (schema examples are numbered and referred to in the text).

4.1. RPC (applies to encoded and literal)

- An RPC style WSDL file contains multiple <part> tags per <message> for each request/ response parameters (10b, 11b).
- Each message <part> tag defines type attributes, not element attributes. (message parts are not wrapped by elements as in Document style WSDL files) (10b, 11b).
- The type attribute in each <part> tag can either; a) reference a complex or simple type defined in the <wsdl:types> section, e.g. <part name="x" type="tns:myType">, or b) define a simple type directly, e.g. <part name="x" type="xsd:int"> (10b, 11b respectively).
- An RPC SOAP request wraps the message parameters in an element named after the invoked operation (2a). If a namespace is locally declared in the <wsdl:soapBody> element, it overrides the target namespace of the WSDL file (24b), otherwise the WSDL target namespace is applied.
- An RPC SOAP response wraps the message parameters in an element named after the invoked operation with 'Response' appended to the element name (12a).
- The difference between RPC encoded and RPC literal styles relates to how the data in the SOAP message is serialised/ formatted when sent over the wire. The abstract parts of the WSDL files are similar (i.e. the <wsdl:types>, <wsdl:message> and <wsdl:portType> elements – refer to Section 3). The difference relates to the definition of the <wsdl:binding> element. The binding element dictates how the SOAP message is formatted and how complex data types are represented in the SOAP message.

4.2. RPC/ encoded

- An RPC/ encoded WSDL file specifies an 'encodingStyle' attribute nested within the <wsdl:binding> (24b). Although different encoding styles are legal, the most common is SOAP encoding. This encoding style is used to serialise data and complex types in the SOAP message. (http://schemas.xmlsoap.org/soap/encoding).
- The use attribute, which is nested within the <wsdl:binding> has the value 'encoded' (24b).
- An RPC/ encoded SOAP message has type encoding information for each parameter element. This is overhead and degrades throughput performance (4a, 7a, 8a) and is not strictly required by the server.
- Complex types are SOAP encoded and are referenced by 'href' references using an identifier (3a). The 'hrefs' refer to 'multiref' elements positioned outside the operation wrapping element as direct children of the <soap:Body> (6a). This complicates the message as the <soap:Body> may contain multiple 'multiref' elements.
- RPC/ encoded is not WS-I compliant [4, 6] which recommends that the <soap:Body> should only contain a single nested sub element.

4.3. RPC/ literal

- RPC/ literal style improves upon the RPC/ encoded style.
- An RPC/ literal WSDL does not specify an encodingStyle attribute.
- The use attribute, which is nested within the <wsdl:binding>, has the value 'literal' (24b).
- An RPC/literal encoded SOAP message has only one nested child element in the <soap:Body> (12a). This is because all parameter elements become wrapped within the operation element that is defined in the WSDL namespace.
- Validation can be problematic when using the RPC/ literal style because qualification of the operational wrapper element comes from the WSDL namespace definitions, not from the individual schema elements defined in the <wsdl:types> section. This introduces additional complexities when integrating nested sub-elements with custom schema defined namespaces; i.e. data binding and validation must also account for the RPC element naming conventions and WSDL namespaces, rather than solely binding and validating plain XML instance documents (as in the Document orientated approach).
- As a result of this (see above bullet), the namespace information for each nested parameter sub-element may be dropped in a SOAP message by the SOAP engine implementation (19a, 20a, 21a). Indeed, we have noticed that in most Java SOAP-engine implementations, often only the operational wrapper element remains fully qualified (it must be noted however, that this does not always apply to other SOAP engine implementations including .NET). This can mean all parts and elements in an RPC/ literal SOAP message share the same namespace and lose their original schema namespace definitions. Consequently, validation may only be possible for limited scenarios, where original schema elements have the same namespace as the WSDL file.
- RPC/ literal is WS-I compliant [6] but has validation limitations.

The main weakness with the RPC style is its lack of support for the constraint of complex data and data validation. The RPC/ encoded style usually adopts the SOAP encoding specification to serialize complex objects, which is far less comprehensive and functional when compared to standard XML schema. Validation can also be problematic when using the RPC/ literal style as data binding must also account for the RPC element naming conventions and WSDL namespaces, rather than solely binding and validating plain XML instance documents.

5. Document Binding Style

In contrast to RPC, Document style encoding provides greater functionality for the validation of data by using standard XML schema as the encoding format for complex objects and data. The main feature of a Document style Web service is that each message part references a concrete schema using the 'element' attribute. The schema is defined in the <wsdl:types> section and can be embedded or imported (refer to Section 12). The contents of the SOAP request/response body can be described completely using the schema without any additional encoding rules or WSDL namespace dependencies. The following summary examines the key features of the Document WSDL binding style and the format of a resulting SOAP message. Table 3 illustrates these key points (schema examples are numbered and referred to in the text).

5.1. Document (applies to literal and wrapped)

- Document style Web services use standard XML schema for the serialisation of XML instance documents and complex data.
- Document style messages do not have type-encoding information for elements, and each element in the soap message is fully qualified by a Schema namespace by direct declaration (34a, 41a, 54a), or by inheritance from an outer element (35a, 36a, 46, 47, 48a).
- Document style services leverage the full capability of XML Schema for data validation.

5.2. Document/ literal

- Document/ literal messages send request (34a, 38a) and response (41a) parameters to and from operations as direct children of the <soap:Body> .
- The <soap:Body> can therefore contain many immediate child sub elements (34a, 38a).
- A Document/literal style WSDL file may therefore contain multiple <part> tags per <message> (37b, 38b).
- Each <part> tag in a message can specify either a type or an element attribute, however, for WS-I compliance, it is recommended that only element attributes be defined in <part> tags for Document style WSDL (37b, 38b).
- This means that every complex type parameter should be wrapped as an element and be defined in the <wsdl:types> section (33b).
- The main disadvantages of the Document/ literal Web service style include: a) the operation name is removed from the <soap:Body> request which can cause interoperability problems (33a), and b) the <soap:Body> will contain multiple children (34a, 38a) if more than one message part is defined in a request/ response message (37b, 38b).
- Document/ literal is not fully WS-I compliant [4, 6], which recommends that the <soap:Body> should only contain a single nested sub element.

5.3. Document/ literal wrapped

- An improvement on the Document/ literal style is the Document/ literal wrapped style.
- When writing this style of WSDL, the request and response parameters of a Web service operation (simple types, complex types and elements) should be 'wrapped' within single all-encompassing request and response elements defined in the <wsdl:types> section (61b - akin to the RPC/ literal style).
- These 'wrapping' elements need to be added to the <wsdl:types> section of the WSDL file (61b).
- The request wrapper element (61b) must have the same name as the Web service operation to be invoked (this ensures the operation name is always specified in the <soap:Body> request as the first nested element).
- By specifying single elements to wrap all of the request and response parameters, there is only ever a single <part> tag per <message> tag (71b).
- A Document/ literal wrapped style WSDL file is fully WS-I compliant [4, 6] because there is only a single nested element in the <soap:Body> (45a).
- Document/literal wrapped style messages are therefore very similar to RPC/ literal style messages since both styles produce a single nested element within a <soap:Body>. The only difference is that for Document/ literal wrapped style: each element is fully qualified with a Schema namespace.

The main advantage of the 'literal' styles over the 'encoded' styles (RPC and Document) is the abstraction/ separation of the type system into a 100% XML Schema compliant data model. In doing this, several important advantages related to abstraction of the data model are further realised which are discussed in the next section.

Table 2 - A Comparison of the RPC Binding Style and Different SOAP Encoding

Style	SOAP Request/Response	WSDL
RPC Encoded	1a <soapenv:Body> 2a <getIndex xmlns:="urn:ehtpx-process">	<definition ….> 1b <types>
Request	3a <admin href="#id0"/> 4a <URL xsi:type="xsd:string"> </URL> 5a </getIndex> 6a <multiRef id="id0" ……> 7a <email xsi:type="xsd:string"> </email> 8a <PN xsi:type="xsd:string"> </PN> 9a </multiRef> 10a </soapenv:Body>	2b <complexType name="AdminT"> 3b <sequence> 4b <element name="email" type="enc:string"/> 5b <element name="PN" type="enc:string"/> 6b </sequence> 7b </complexType> 8b </types> 9b <wsdl:message name="getIndexRequest">
Response	11a <soapenv:Body> 12a <ns1:getIndexResponse soapenv:encodingStyle="http://.../" xmlns:ns1="urn:ehtpx-process"> 13a <getIndexReturn xsi:type="soapenc:string" xmlns:soapenc="http://../"> …………….. 14a </getIndexReturn> 15a </ns1:getIndexResponse> 16a </soapenv:Body>	10b <wsdl:part name="admin" type="tns:AdminT"/> 11b <wsdl:part name="URL" type="enc:string"/> 12b </wsdl:message> 13b <wsdl:portType name="IndexService"> 14b <wsdl:operation name="getIndex"> 15b <wsdl:input message="impl:getIndexRequest" name="getIndexRequest"/> 16b <wsdl:output message="impl:getIndexResponse" name="getIndexResponse"/>
RPC Literal		17b </wsdl:operation> 18b </wsdl:portType>
Request	17a <soapenv:Body> 18a <getIndex xmlns="urn:ehtpx-process"> 19a <admin xmlns=""> 20a <email> </email> 21a <PN> </PN> 22a </admin> 23a <URL xmlns=""> </URL> 24a </getIndex> 25a <soapenv:Body>	19b <wsdl:binding name="IndexSoapBinding" type="impl:IndexWS"> 20b <wsdlsoap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http"/> 21b <wsdl:operation name="getIndex"> 22b <wsdlsoap:operation soapAction=""/> 23b <wsdl:input name="getIndexRequest"> 24b <wsdlsoap:body encodingStyle="http://../" namespace=" urn:ehtpx-process" use="**encoded | literal**"/>
Response	26a <soapenv:Body> 27a <getIndexResponse xmlns=" urn:ehtpx-process "> 28a <getIndexReturn> ……………… 29a </getIndexReturn> 30a </getIndexResponse> 31a </soapenv:Body>	25b </wsdl:input> 26b <wsdl:output name="getIndexResponse"> 27b <wsdlsoap:body encodingStyle="http://..if use is encoded only" namespace=" urn:ehtpx-process" use=" **encoded | literal** "/> 28b </wsdl:output> 29b </wsdl:operation> 30b </wsdl:binding> <wsdl:service> ......</wsdl:service> </defination>

Table 3 - A Comparison of the Document Binding Style and Different SOAP Encoding

Style	SOAP Request/Response	WSDL
Doc Literal		<definition ….> 31b <types> 32b <complexType name="AdminT">… </complexType> 33b <element name="admin" type="tns:AdminT"> 34b <element name="URL" type=" enc:string"> 35b </types> 36b <wsdl:message name="getIndexRequest"> 37b <wsdl:part name="in0" element="tns:admin"/> 38b <wsdl:part name="URL" element="enc:string"/> 39b </wsdl:message> 40b <wsdl:portType name="IndexService"> 41b <wsdl:operation name="getIndex"> 42b <wsdl:input message="impl:getIndexRequest" name="getIndexRequest"/> 43b <wsdl:output message="impl:getIndexResponse" name="getIndexResponse"/> 44b </wsdl:operation> 45b </wsdl:portType> 46b <wsdl:binding name="IndexSoapBinding" type="impl:IndexWS"> 47b <wsdlsoap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/> 48b <wsdl:operation name="getIndex"> 49b <wsdlsoap:operation soapAction=""/> 50b <wsdl:input name="getIndexRequest"> 51b <wsdlsoap:body use="literal"/> 52b </wsdl:input> 53b <wsdl:output name="getIndexResponse"> 54b <wsdlsoap:body use="literal"/> 56b </wsdl:output> 57b </wsdl:operation> 58b </wsdl:binding> <wsdl:service> ......</wsdl:service> </defination>
Request	32a <soapenv:Body> 33a 34a <admin xmlns="urn:ehtpx-process"> 35a <email xmlns=""> </email> 36a <PN xmlns=""> </PN> 37a </admin> 38a <URL xmlns=""> </URL> 39a </soapenv:Body>	
Response	40a <soapenv:Body> 41a <adminReturn xmlns="http://paperwstest" ………. 42a </adminReturn> 43a </soapenv:Body>	
Doc Wrapped		<definition ….> 59b <types> 60b <complexType name="AdminT"> ...</complexType> 61b <element name ="getIndex"> 62b <complexType> 63b <sequence> 64b <element name ="admin" type ="tns:AdminT"/> 65b <element name =" URL " type ="xsd:string" /> 66b </sequence> 67b </complexType > 68b </element> 69b </types> 70b <wsdl:message name="getIndexRequest"> 71b <wsdl:part name="in0" element="tns:getIndex"/> 72b </wsdl:message> 73b <wsdl:binding name="IndexSoapBinding" type="impl:IndexWS"> 74b <wsdlsoap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/> 75b <wsdl:operation name="getIndex"> 76b <wsdlsoap:operation soapAction=""/> 77b <wsdl:input name="getIndexRequest"> 78b <wsdlsoap:body use="literal"/> 79b </wsdl:input> 80b <wsdl:output name="getIndexResponse"> 81b <wsdlsoap:body use="literal"/> 82b </wsdl:output> 83b </wsdl:operation> 84b </wsdl:binding> <wsdl:service> ......</wsdl:service> </defination>
Request	44a <soapenv:Body> 45a <getIndex xmlns=" urn:ehtpx-process"> 46a <admin> 47a <email xmlns=""> </email> 48a <PN xmlns=""> </PN> 49a </admin> 50a <URL xmlns=""> </URL> 51a </getIndex> 52a </soapenv:Body>	
Response	53a <soapenv:Body> 54a <adminReturn xmlns="http://paperwstest" ………. 55a </adminReturn> 56a </soapenv:Body>	

6. Data Abstraction

Abstraction of the Web service type system into a 100% XML Schema compliant data model is relevant for both RPC and Document style services, and produces several important advantages;

- *Separation of Roles*
  The type system can be fully abstracted and developed in isolation from the network protocol and communication specific details of the WSDL file. In doing this, the focus becomes centred upon the business/ scientific requirements of the data model. In our experience, this greatly encourages collaboration between the scientists who are involved with the description of data.
- *Data Model Re-usability/ Extensibilty*
  Existing XML Schema can be re-used rather than re-designing a new type system for each new Web service. This helps reduce development efforts, cost and time. A good example is the Semantic Web, which requires many additional data annotations to map data to externally declared ontology's (e.g. RDF, DAMLS or OWL). Separation of the data from the interface makes it easier to adjust with future requirements of these different specifications.
- *Isolation of Changing Components*
  In our experience, the data model is the component that is most subject to change, often in response to changing scientific requirements. Its isolation therefore limits the impact on other Web service components such as the concrete WSDL file implementation (see Section 12).
- *Avoid Dependency on SOAP Namespaces and Encoding Styles*
  Manual modeling of XML Schema may constitute extra effort but this gives the developer the most control and avoids using SOAP framework dependent namespaces and encoding styles. Most of the SOAP frameworks are traditionally RPC-centric and create WSDL based on the RPC/ encoding style, which is not WS-I compliant. This also applies to languages other than Java.
- *Full XML Schema Functionality*
  The XML Schema type system leverages the more powerful features of the XML Schema language for description, constraint and validation of complex data (e.g. XSD patterns/ regular expressions, optional elements, enumerations, type restrictions etc). In our experience, this has proven invaluable for the description and constraint of complex scientific data.
- *Can Choose to Use Data Binding/ Validation Framework Implemented by SOAP Engine, or Can Choose to Separate Data Binding/ Validation from SOAP Engine.*
  Having the choice to either delegate data binding/ validation to the SOAP engine or to use a separate binding and validation framework introduces a number of additional advantages and disadvantages. An internal SOAP engine data binding/ validation framework offers simplicity, but introduces dependencies on the SOAP engine for data-centric operations. Use of a separate data binding and validation framework is more complex, but cleanly separates the roles of the SOAP engine and binding/validation framework into 'communication' and 'data-centric' roles. In doing this, the binding/validation framework can be used in isolation of the SOAP engine for non-Web service or 'out-of-band' operations (e.g. persistence). Section 10 provides a detailed discussion of both approaches to data binding and validation and their associated advantages and disadvantages.

7. Loose Versus Strong Data Types

A 'loosely typed' Web service means the WSDL file does not contain an XML schema in the type system to fully define the format of the messages. Instead the WSDL defines generic types to encapsulate data within messages (i.e. encoded Strings, CDATA, SOAP attachments, xsd:anyType, xsd:any). The main advantage of loose types is that different data formats can be encapsulated without having to update the WSDL interface (e.g. different document types can be transmitted as SOAP attachments, or different markup fragments such as XML can be encoded by strings). This allows Web service components to be replaced in a 'plug-and-play' fashion. Conversely, a 'strongly typed' Web service means the WSDL type system strictly dictates the format of the message data from the outset. Strongly typed Web services are less flexible, but fully describe the required data in a single contract (i.e. the WSDL interface). Each style influences the chosen approach to data binding and validation and each has its own advantages and disadvantages as summarized in Table 4.

Table 4 – Advantages and Disadvantages of Loose versus Strong Data Typing

Modeling Approach	Advantages	Disadvantages
Loose Type	• Easy to develop. • Easy to implement. • Stable WSDL interface (can change the data that is encapsulated by the loose type without updating the WSDL interface). • Flexibility in implementation. • Single Web service may handle multiple types of message. • Can be used as a gateway service which routes to actual services based on contents of message.	• Requires additional/ manual negotiation between client and service to establish the format of data wrapped in a generic type • This may cause problems regarding maintaining consistent implementations and for client/ service relations • No control on the messages • Prone to message related exceptions due to inconsistencies in the format of data sent and accepted (requires WS to be tolerant in what it accepts – this adds extra coding complexity).
Strong Type	• Properly defined interfaces. • Tight control on the data with value constraints. • Message validation. • Often more robust (only highly constrained data enters Web service business logic). • Minimized network overhead. • Benefits from richness of XML.	• Can be more difficult to develop (requires a working knowledge of XML and WSDL). • Resistive to change in data model.

8. Loosely Typed Web services

A loosely typed WSDL interface specifies generic data types for an operation's input and output messages (either string, CDATA, Base64-Encoded, xsd:any, xsd:anyType or Attachment types). The loosely typed approach requires extra negotiation between providers and consumers in order to agree on the format of the data that is encapsulated by the generic type. Consequently, an understanding of the WSDL alone is usually not sufficient to consume the service. Often, loosely typed Web services require a low level understanding of different APIs like SAAJ, JAX-RPC and DOM coupled with a thorough understanding of XML and namespaces. This is because different tools have non-consistent support for some of the loose types.

8.1. ‘String’ Loose Data Type

A string variable can be used to encode markup and complex data allowing the WSDL interface to define simple string input/ output parameters for operations (i.e. akin to simple ‘helloworld’ style services). The string input could be an XML fragment or multiple name-value pairs (similar to a query string). In doing this, the implementation has to parse and extract the data from the string before invoking the business logic. An XML document formatted as a string requires extra coding and decoding in order to escape the markup (e.g. XML special characters). This can drastically increase the message size. This approach is memory inefficient as the whole string is read directly into memory, and is generally only suitable only for encoding simple markup fragments that have limited use of namespaces. Listing 2 shows an example of an escaped XML fragment. The SOAPBodyElement class of the SAAJ API can also be used to create XML fragments from strings (Listing 3). The SOAPBodyElement can be used directly to invoke the service. In most cases, escaping the markup is performed automatically by the SOAP engine.

Listing 2 – String representation of XML and escaping XML special characters with the \ character.

```
public String request = “<admin>” +
“<email xmlns=\"\"> </email>” +
“<PN xmlns=\"\"> </PN>” +
“</admin>” +
“<URL xmlns=\"\"> </URL>” ;
```

Listing 3 –Using strings to create XML fragments using SAAJ API

```
SOAPBodyElement[] input = new SOAPBodyElement[2];
input[0] = new SOAPBodyElement(XMLUtils.StringToElement(
"urn:foo0",  "e0", "This will represent a body element "));
input[1] = new SOAPBodyElement(XMLUtils.StringToElement(
"urn:foo2", "e2", " string loose data type here, e.g. name / value pairs "));
Vector elems = (Vector) call.invoke( input );
```

8.2. The ‘CDATA Section’ Loose Data Type

A CDATA section can be used to embed markup and data directly within a SOAP message. The content defined within the CDATA section may contain any special characters because it is not parsed by a SOAP engine. This prevents the need to escape data. Listing 4 shows how the DOM API can be used to create a CDATA section and Listing 5 shows the resulting CDATA section that is embedded within a SOAP message. Traditionally, few systems wrap encoded data with a CDATA section and this approach introduces limitations related to interoperability (for example, .NET does not handle CDATA correctly).

Listing 4 – Wrapping XML markup within a CDATA section.

```
DocumentBuilder builder = cumentBuilderFactory.newInstance().newDocumentBuilder();
Document doc = builder.newDocument();
Element cdataElem = doc.createElementNS("urn:foo", "e3");
CDATASection cdata = doc.createCDATASection("Text
with\n\tImportant  <b>  whitespace </b> and tags – no escaping of XML special chars! ");
cdataElem.appendChild(cdata);
```

Listing 5 – A CDATA section embedded within a SOAP message fragment.

```
<ns3:e3 xmlns:ns3="urn:foo">
  <![CDATA[Text with    Important
    <b>  whitespace </b> and tags – no escaping of XML special chars ! ]]>
 </ns3:e3>
```

It is possible to mix the DOM and SAAJ APIs to combine CDATA and string loose types where necessary. This provides maximum flexibility. In Listing 6, a CDATA section is created using DOM and an element is created from a string using SAAJ.

Listing 6 – Mixing DOM and SAAJ APIs to create CDATA sections using DOM, and elements from strings using SAAJ

```
DocumentBuilder builder =
DocumentBuilderFactory.newInstance().newDocumentBuilder();
Document doc           = builder.newDocument();
Element cdataElem      = doc.createElementNS("urn:foo", "e3");
CDATASection cdata     = doc.createCDATASection("Text
with\n\tImportant  <b>  whitespace </b> and tags! ");
cdataElem.appendChild(cdata);
SOAPBodyElement[] input = new SOAPBodyElement[2];
input[0] = new SOAPBodyElement(cdataElem);
input[1] = new SOAPBodyElement(XMLUtils.StringToElement(
"urn:foo1", "e1", "string loose data type here, e.g. name / value pairs"));
```

8.3. ‘xsd:any’ and ‘xsd:anyType’ Loose Data Types

An XML schema defined in the WSDL <types> section may define <xsd:any> or <xsd:anyType> elements. These elements make XML documents extensible, as they allow instance documents to contain additional elements that are not declared in the schema. Consequently, arbitrary XML can be embedded directly into the SOAP message. Partners receive the actual XML, but no contract is specified regarding what the XML data actually represents. The <xsd:any> element is a 'placeholder' that can be directly replaced by an element, whereas <xsd:anyType> can be used to declare elements where the type is unknown. This is shown in Listing 7. Extraction of information requires raw XML manipulation, using the Java SAAJ, DOM and SAX APIs for example. The use of these elements is suitable for dynamic environments requiring raw XML handling, and enables schema versioning and flexible polymorphism. It must be noted however, that extensive use of <xsd:any> and <xsd:anyType> deviates from the intent of a WSDL file in providing a full service interface description.

Listing 7 – The <xsd:any> element can be directly replaced by any XML element. The <xsd:anyType> element is used to declare elements where the type is unknown.

```
<types>
  <schema ….>
    <element name="getIndexRequest">
      <complexType> <sequence>
           <!-- replace with any element -->
           <any maxOccurs="1"/>
       </sequence> </complexType>
    </element>
    <!-- declare element where type is unknown -->
   <element name="dummy" type="xsd:anyType" />
  </schema>
</types>
```

In the programming environment, <xsd:any> and <xsd:anyType> elements should be mapped to language specific types. Unfortunately, different SOAP Frameworks map these types to different native objects and are often non-consistent. An example of this is shown in Table 5 which shows how the popular Java2WSDL and WSDL2Java tools of the Axis framework map between these schema types and Java objects. As can be seen, the mappings are non-consistent when traversing between the schema types and Java objects.

Table 5 – Non consistent mapping by the SOAP engine toolkit between schema types <xsd:any> and <xsd:anyType> and Java objects

Schema Type	Tool	Java Mapping
xsd:any	WSDL2Java	MessageElement
xsd:anyType	WSDL2Java	java.lang.Object

Java Mapping	Tool	Schema Type
MessageElement	Java2WSDL	Error
SOAPElement	Java2WSDL	xsd:any

8.4. 'Base64 encoding' Loose Data Type

Base64 Content-Transfer-Encoding is a two way encoding scheme defined by RFC 1521 that is designed to represent data as arbitrary sequences of octets. An XML document can be transmitted as a Base64 encoded string or as raw bytes in the body of a SOAP message. This approach is WS-I compliant, meaning every SOAP engine handles this data in a compatible fashion. This may be useful when the XML contains characters that are not supported by the SOAP message info-set or by the runtime. The encoding and decoding algorithms are simple, but encoded data are consistently about 33% larger than non-encoded data. Base64 encoding is also used by email applications for sending binary attachments, by Web page file uploads, and by browsers when sending Basic Authentication credentials in HTTP headers. There are several open source implementations in Java of this encoding scheme. A typical WSDL file defining Base64 encoding is shown in Listing 8. Listing 9 shows a corresponding template of the Web service implementation, and Listing 10 shows a typical client that uses Base64 encoding to encode a small file and a string. Listing 11 shows an example of the encoded data that is embedded within a SOAP message.

Listing 8 – WSDL Document defining the xsd:base64Binary type.

```
  …
 <element name="getIndexRequest" type="xsd:base64Binary"/>
 <element name="getIndexReply" type="xsd:base64Binary"/>
 <element name="getIndexFault">
   <complexType>
     <sequence>
       <any maxOccurs="1"/>
     </sequence>
   </complexType>
 </element>
  .....
<message name="getIndex_Request">
  <part name="getIndex_Request" element="ns1:getIndexRequest"/>
</message>
<message name="getIndex_Response">
  <part name="getIndex_Response" element="ns1:getIndexReply"/>
</message>
<message name="getIndexProblem">
  <part name="getIndexProblem" element="ns1:getIndexFault"/>
</message>
```

Listing 9– A service endpoint implementation that maps a byte array to the xsd:base64Binary type.

```
public byte[] getIndex(byte[]getIndex_Request)throws java.rmi.RemoteException{}
```

Listing 10 – Client code that invokes a service twice using xsd:base64Binary loose data type to encode a file and a string.

```
File file = new File("Base64WS.wsdl");
InputStream inputFileStream = new FileInputStream(file);
long length = file.length();           // Get the size of the file
byte[] bytes = new byte[(int) length];  // Create the byte array to hold the data
```

```
// Read in the bytes
int offset = 0;
int numRead = 0;
while (offset < bytes.length &&
      (numRead = inputFileStream.read(bytes, offset,  bytes.length - offset)) >= 0) {
   offset += numRead;
}
// Ensure all the bytes have been read in
if (offset < bytes.length) {
   throw new IOException("Could not completely read file " + file.getName());
}
inputFileStream.close(); // Close the input stream and return bytes
base64WS. getIndex (bytes);  // Invoke the Web service

String dummy = "This is the Dummy String";
String dummy = “It is the dummy value”
base64WS. getIndex (dummy.getBytes()); // Invoking the Web service
```

Listing 11– SOAP message with Base64 encoded data.

```
 <soapenv:Body>
   <getIndex xmlns=" urn:ehtpx-process">
      <getIndex_Request>
PD94bWwgdmVyc2lvbj0iMS4wIiBlbmNvZGluZz0iVVRGLTgiP
z4NCjx3c2RsO………Dwvd3NkbDpzZXJ2aWNlPg0KCjwvd3NkbDpkZWZpbml
     aW9ucz4NCg==
      </getIndex_Request >
   </ns1: getIndex >
</soapenv:Body>
```

Embedding encoded data directly within the SOAP message is convenient. However, because the raw data is nested in the SOAP body, it must be scanned unnecessarily by every XML-parser involved in routing the message (e.g. by intermediaries and gateway services). An alternative approach is to wrap encoded data within CDATA sections so that it is not parsed (see section 8.2). In practical terms, the Base64 loose data type is suitable only for encoding relatively small amounts of data and small files.

8.5. ‘SOAP Attachment’ Loose Data Type

SOAP attachments can be used to send data of any format, especially when it is not practical to embed or encode data directly within the SOAP body, such as large binary files, images or any other arbitrary file format. Sending data in an attachment is efficient because the size of the SOAP body is minimized which enables faster message processing (the SOAP message contains a reference to the data but not the data itself). Attachment support is provided by most of the SOAP frameworks. Additional advantages over other techniques include the ability to handle large documents, multiple attachments can be sent in a single Web service invocation, and attachments can be compressed for efficient network transport. SOAP attachments can be implemented using a number of different methods, but none of these are universally accepted. The two most commonly used mechanisms are MIME [15] and DIME [16]. In addition, the WS-I organisation has recently introduced an XML type for attachments ‘wsi:swaRef [17].’

8.5.1. MIME (Multipurpose Internet Mail Extensions)

As the name implies, MIME [15] was originally developed to standardize the format of email attachments. MIME messages can contain text, images, audio, video, or other application-specific data. The standardized MIME type system has been adopted by many other applications besides email. SOAP with Attachments (SwA) extends SOAP 1.1 to add support for MIME attachments.

8.5.1.1. WSDL Document with a MIME Attachment

The MIME information is defined in the <wsdl:binding> section of the WSDL, not in the abstract section of the service interface (i.e. not in <types>, <messages>, <portType>). WSDL parsing tools ignore the MIME information and use the WSDL to generate stubs and proxies only. Listing 12 shows an example WSDL fragment that contains information for three MIME attachments in the <wsdl:binding>.

Listing 12 – WSDL binding that references 3 MIME attachments.

```
<wsdl:binding name="SoapMimeBinding" type="svc:PortType">
   <soap:binding transport="http://schemas.xmlsoap.org/soap/http" style="document"/>
      <wsdl:operation name="GetImageFile">
        <soap:operation  soapAction="http://example.com/GetImageFile"/>
        <wsdl:input>
      <mime:multipartRelated>
        <mime:part> <soap:body use="literal"/> </mime:part>
        <mime:part> <mime:content part="...." type="image/jpeg"/>  </mime:part>
        <mime:part>
           <mime:content part="..." type="application/octet-stream"/>
        </mime:part>
      </mime:multipartRelated>
        </wsdl:input>
      </wsdl:operation>
 </wsdl:binding>
```

8.5.1.2. SOAP Message with a MIME Attachment

The MIME multipart/related type is used to construct a SOAP message that is combined with the attachments as a single logical unit. The original or ‘primary’ message is carried in the root body part of the multipart/related structure. The primary message may reference additional ‘attachment’ or ‘MIME’ parts, which contain the data. The MIME parts are referenced using either a ‘Content-ID MIME header’ or a ‘Content-Location MIME header.’ Listing 13 shows a sample SOAP message with an attachment.

Listing 13 – SOAP message with MIME attachment. The message as a whole is a multipart/related structure where the primary message references the attachment using the Content-Id header.

```
------=_Part_2_17975110.1152232352868
Content-Type: text/xml; charset=UTF-8
Content-Transfer-Encoding: binary
Content-Id: <7B4A1C2A5D383AD0BBFB06D460EA926A>
  <soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
       xmlns:xsd="http://www.w3.org/2001/XMLSchema"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    <soapenv:Body>
      <sendDH xmlns="http://testproject">
        <myAttachmentElem href="cid:1E019A6B976FD5A8AD772F60A1E510D7"/>
      </sendDH>
    </soapenv:Body>
  </soapenv:Envelope>
------=_Part_2_17975110.1152232352868
Content-Type: image/jpeg
Content-Transfer-Encoding: binary
Content-Id:
  <1E019A6B976FD5A8AD772F60A1E510D7>
    ...........
------=_Part_2_17975110.1152232352868—
```

8.5.1.3. Service Implementation with a MIME Attachment

For Java, well-known MIME types can be mapped to corresponding Java objects through the use of Java 'DataHandlers' and 'DataContentHandlers' from the JavaBeans Activation framework. This provides direct application access to the operations that are specific to each content type. Unknown MIME types are returned as raw bytes in a 'java.io.InputStream.' Listing 14 shows the most common MIME types and their corresponding Java classes.

Listing 14 – Mapping of basic MIME types and Java classes.

MIME Type	Java Type
image/gif, image/jpeg	java.awt.Image
text/plain	java.lang.String
multipart/*	javax.mail.internet.MimeMultipart
text/xml, application/xml	javax.xml.transform.Source

8.5.2. DIME (Direct Internet Message Encapsulation)

IBM and Microsoft released a competing specification called WS-Attachments, which uses DIME [16] rather than MIME. In DIME, data is transmitted in small, consumable 'chunks.' The receiver only reads a fixed amount of data into memory at a time. This differs from MIME, which requires the allocation of large contiguous chunks of memory to read entire attachments (this can be problematic when considering large attachments of unknown size). If the receiver stops reading the data, the sender will stop generating data until a timeout period is reached, upon which the SOAP operation fails. This prevents the sender and receiver from running out of memory but imposes 'tighter coupling' between the sender and receiver which deviates from the loosely coupled principles of SOAs. The IETF draft [16] for DIME expired in December 2002. The current status and future direction of DIME and WS-Attachments are 'uncertain.'

8.5.2.1. WSDL with a DIME Attachment

DIME (like MIME) also necessitates extensions to the WSDL interface for binding attachments to the transport protocol. This involves the addition of <dime:message> elements to the <wsdl:input> and/or <wsdl:output> elements of a <wsdl:operation> binding. The <dime:message> element must also set the 'wsdl:required' attribute to 'true' and the 'layout' attribute to one of the following two values:

1) 'http://schemas.xmlsoap.org/ws/2002/04/dime/closed-layout' specifies that all parts of a DIME message should be referenced from the primary SOAP message and in the specified order.
2) 'http://schemas.xmlsoap.org/ws/2002/04/dime/open-layout' specifies that additional attachments can be included in a DIME message which need not be referenced by the SOAP message (provided they come after referenced attachments).

In addition to these new element definitions, the WSDL extensions for DIME also add the following three elements to further describe DIME attachments: <content:type> specifies the value type of an attachment, <content:mediaType> specifies the MIME media-type of an attachment, and <content:documentType> specifies the value type of an XML document. A WSDL consuming tool that does not understand these DIME extensions may fail when parsing the WSDL document. Listing 15 shows an example WSDL fragment that contains information for a DIME attachment.

Listing 15 – WSDL binding that specifies a DIME attachment.

```
<wsdl:binding name="SoapDimeBinding" type="svc:PortType">
 <soap:binding transport="http://schemas.xmlsoap.org/soap/http" style="document"/>
 <wsdl:operation name="GetMediaFile">
  <soap:operation  soapAction="http://example.com/GetMediaFile"/>
  <wsdl:input>
   <dime:message
        layout="http://schemas.xmlsoap.org/ws/2002/04/dime/closed-layout"
    wsdl:required="true"/>
   <soap:body parts="body" use="literal"/>
  </wsdl:input>
 </wsdl:operation>
</wsdl:binding>
```

8.5.2.2. SOAP Message with a DIME Attachment

DIME messages are created by a DIME generator and consumed by a DIME parser. A DIME message contains a series of one or more DIME records at the start of the message. See Table 6 for the description of DIME message records and Listing 16 for the sample SOAP message). Each record contains binary information used by a parser to interpret the message. The most significant fields in a DIME record are 'MB' and

'ME,' which specify whether this record is the first or the last of the message. This feature is an important difference between MIME and DIME. When parsing a MIME message, all of the data in the message must be read into memory and interpreted to determine the number of attachments. However, when using DIME, a parser can simply use the data in the record headers to quickly walk through and count the number of records in the message and obtain their size. This feature enables DIME to process messages faster and more efficiently.

Table 6 – Fields specified by a DIME record. Each field is read by a DIME parser to gain information about the attachments. This means that DIME does not have to read the attachments into memory to obtain their size and quantity (like MIME).

Field	Description
VERSION (5 bit)	Specifies the version of the DIME message
MB (1 bit)	Specifies that this record is the first record of the message
ME (1 bit)	Specifies that this record is the last record of the message
CF (1 bit)	Specifies that the contents of the message have been chunked
TYPE_T (4 bit)	Specifies the structure and format of the TYPE field
RESERVED (4 bit)	Reserved for future use
OPTIONS_LENGTH (16 bit)	Specifies the length (in bytes) of the OPTIONS field, excluding any necessary padding (up to 3 bytes)
ID_LENGTH (16 bit)	Specifies the length (in bytes) of the ID field, excluding any necessary padding (up to 3 bytes)
TYPE_LENGTH (16 bit)	Specifies the length (in bytes) of the TYPE field, excluding any necessary padding (up to 3 bytes)
DATA_LENGTH (32 bit)	Specifies the length (in bytes) of the DATA field, excluding any necessary padding (up to 3 bytes)
OPTIONS	Contains any optional information used by a DIME parser
ID	Contains a URI for uniquely identifying a DIME payload with any additional padding; the length of this field is specified by ID_LENGTH
TYPE	Specifies the encoding for the record based on a type reference URI or a MIME media-type; reference type is specified by TYPE_T, and the length of this field is specified by TYPE_LENGTH
DATA	Contains the actual data payload for the record; format of the data depends on the TYPE specified for the record; length of this field is specified by DATA_LENGTH

Listing 16 – SOAP message with a DIME attachment. The binary data at the top of the message represent the DIME record fields. The binary data beneath the SOAP envelope represent 'chunks' of attachment data.

```
00001 1 0 0 0010 00000000000000000000
0000000000000000 0000000000101000
00000000000000000000000110110101
http://schemas.xmlsoap.org/soap/envelope
<soap-env:Envelope
  xmlns:soap-env="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:msg="http://example.com/DimeExample/Messages/"
  xmlns:ref= "http://schemas.xmlsoap.org/ws/2002/04/reference/">
  <soap-env:Body>
    <msg:GetMediaFile>
      <msg:fileName>myMediaFile.mpg</msg:fileName>
      <msg:file ref:location="uuid:F2DA3C9C-74D3-4A46-B925-B150D62D9483" />
    </msg:GetMediaFile>
  </soap-env:Body>
</soap-env:Envelope>
-----------------------------------------------------------------------
00001 0 0 1 0001 0000000000000000000
0000000000101001 0000000000001010
00000000000101011010101011100000
uuid:F2DA3C9C-74D3-4A46-B925-B150D62D9483
video/mpeg
<<First 1.42 MB of binary data for myMediaFile.mpg>>
-----------------------------------------------------------------------
00001 0 1 0 0000 0000000000000000000
0000000000000000 0000000000000000
00000000000010000110110001000000
<<Remaining 552 KB of binary data for myMediaFile.mpg>>
```

8.5.2.3. Service Implementation with a DIME Attachment

DIME is configured for both the server and client according to the SOAP framework (MIME is usually the default). For example, in Axis DIME may be configured by setting properties on both the server and client as shown in Listing 17.

Listing 17 – Axis specific configuration of service and client to use DIME for attachments.

```
Server:
      org.apache.axis.Message rspmsg = org.apache.axis.AxisEngine.
                  getCurrentMessageContext().getResponseMessage();
       rspmsg.getAttachmentsImpl().setSendType(
                  org.apache.axis.attachments.Attachments.SEND_TYPE_DIME);
Client:
       org.apache.axis.client.Call call;   …
       call.setProperty(call.ATTACHMENT_ENCAPSULATION_FORMAT,
                call.ATTACHMENT_ENCAPSULATION_FORMAT_DIME);
```

8.5.3. DIME Versus MIME

The correct choice of MIME or DIME depends upon an understanding of the issues related to each technology, and choosing the most suitable approach for the service in question. The MIME multipart approach is designed to be efficient for the sender of a MIME message. The sender does not have to know the quantity of data they are sending because they simply stream the data and append the MIME separator string. However, the receiver has no indication of the quantity of incoming data. The receiver may guess an appropriate size for buffer allocation, but undoubtedly will have to deal with situations when there is more data than allocated in the buffer. In addition, determination of the data record boundaries is an expensive process when compared to DIME, where the receiver can efficiently step from record to record based on the lengths of the data given in the data record header. Despite the advantages offered by DIME, creating and reading a DIME record requires parsing the bit-order of the integers in the data record header. It must be noted that not all SOAP frameworks are currently able to parse the bit-order proposed by the DIME specification and, consequently, do not benefit from DIME. Service providers should therefore be aware of the level of DIME compatibility of their chosen SOAP framework before implementing DIME.

8.5.4. WS-I SOAP with Attachment Reference Type

The WS-I organisation recently introduced an XML type for attachments (ATTACHMENTS PROFILE 1.0 2nd Edition – April 2006) [17]. The WS-I 'wsi:swaRef' reference type addresses the limitations associated with declaring MIME and DIME attachments within the concrete part of the WSDL interface (i.e. within the <wsdl:binding>). Declaring attachments within the concrete WSDL means that the attachments are not visible from the abstract part of the WSDL interface. In contrast to MIME and DIME, 'wsi:swaRef' can be referenced from within the abstract WSDL interface. Furthermore, its location is not limited to message parts as it can be used to define an element or complexType in the schema(s) contained in the WSDL <types> section. This is shown in Listing 18.

Listing 18 – XML Schema document using the 'wsi:swaRef' reference type for specifying attachments.

```
<element name="myAttachmentElem">
  <complexType>
    <sequence>
      <element name="octet" type="wsi:swaRef"/>
    </sequence>
  </complexType>
</element>
```

8.5.4.1. SOAP message with WS-I SOAP Attachment Reference Type

The resulting SOAP message shown in Listing 19 is very similar to the MIME based SOAP message shown in Listing 12. The main difference is that the wsi:swaRef content ID is the value of the <myAttachmentElem> element, whereas in the MIME based SOAP message, the content ID is a value of attribute "*href*" in the <myAttachmentElem>.

Listing 19 – SOAP message with an element of type 'wsi:swaRef.'

```
<soapenv:Body>
  <sendDH xmlns="http://testproject">
    <myAttachmentElem>
      <octet>
          cid:1E019A6B976FD5AAD772F60A1E510D7
      </octet>
    </myAttachmentElem>
  </sendDH>
</soapenv:Body>
```

WS-I has defined only a single reference type so it automatically maps to the native objects of the service implementation. For example, for Java ‘wsi:swaRef’ maps to the ‘DataHandler’ class from the ‘java.activation’ package.

9. Strongly typed Web services

A purely strongly typed WSDL interface defines a complete definition of an operation's input and output messages with XML Schema, with additional constraints on the actual permitted values where necessary. Strong typing applies to both Document and RPC literal styles (although strong typing is most common in Document style due to the limitations associated with RPC validation as described in Section 5). The XML schema used to constrain the <wsdl:messages> can be either embedded directly into the WSDL document or imported using <xsd:import> (refer to section 12 for details regarding WSDL modularization). It is important to understand, that strongly typed services do not have to be solely strongly typed, as they may combine both strong typing with loose/ generic types where necessary. Strong typing is especially relevant for scientific applications which often require a tight control on message values, such as length of string values, range of numerical values, permitted sequences of values (e.g. organic compounds must have Carbon and Hydrogen in the chemical formula and rotation angle should be between 0 – 360 degrees). From our experiences related to the e-HTPX project [10], the best approach involved mixing the different styles where necessary. For mature Web services, where the required data is established and stable, the use of strong data typing was preferable. For immature Web services where the required data is often subject to negotiation and revision, loose typing was preferable. We often used the loose typing approach during initial developments and prototyping.

9.1. Strongly Typed WSDL Example

Listing 20 shows a complete example of a strongly typed, Document-literal/ wrapped WSDL file that describes a service developed for the e-HTPX project (specifically, the Bulk Molecular Replacement service) [10]. The WSDL file imports a separately defined XML schema that is shown in Listing 21. The WSDL file defines two service operations ‘describeBulkMR’ and ‘executeBulkMR’. The input and output messages for each operation reference single request and response elements that are globally defined in the imported schema (<describeBulkMR>, <describeBulkMRResponse>, <executeBulkMR>, <executeBulkMRResponse>). The use of single, global elements for request and response messages enables each message to be fully validated according to the rules defined by the schema (Listing 21). The schema makes use of a number of XSD value constraint techniques. As shown in Listing 21, XSD restrictions, patterns, and enumerations are all used where necessary to tightly constrain the permitted values for a number of elements (e.g. the ‘sequence’, ‘pdbIgnoreCodes’ and ‘mrprogram’ elements). In the following sections, the WSDL and schema files shown in Listings 20 and 21 are used to demonstrate a number of different approaches to data binding and validation.

Listing 20 – Strongly typed WSDL file that imports a XML schema file with value constraints (BulkMRFinal.xsd - shown in Listing 21). The schema is imported within the <wsdl:types> element. The schema is used to constrain the input and output messages of the two Web service operations (describeBulkMR and executeBulkMR). Note the use of '*style=document*' and '*use=literal*' attributes in the schema binding element.

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions name="bulkMR"
   targetNamespace="http://e-htpx.ac.uk/bulkmr"
   xmlns:import1="http://e-htpx.ac.uk/bulkMR"
   xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
   xmlns:tns="http://e-htpx.ac.uk/bulkmr"
   xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema"
   xmlns:xsd1="http://e-htpx.ac.uk/bulkMR.xsd1">

   <!-- Types element used to import or directly declare the schema types  -->
   <wsdl:types>
      <xsd:schema
         targetNamespace="http://e-htpx.ac.uk/bulkMR.xsd1"
         xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
         xmlns:xsd="http://www.w3.org/2001/XMLSchema"
         xmlns:xsd1="http://e-htpx.ac.uk/bulkMR.xsd1">
         <xsd:import
         namespace="http://e-htpx.ac.uk/bulkMR"
         schemaLocation="BulkMRFinal.xsd"/>
      </xsd:schema>
   </wsdl:types>

   <!-- Message elements describe the parameters and return values -->
   <wsdl:message name="Msg_executeRequest">
      <wsdl:part element="import1:executeBulkMR" name="parameters"/>
   </wsdl:message>
   <wsdl:message name="Msg_executeResponse">
      <wsdl:part element="import1:executeBulkMRResponse" name="result"/>
   </wsdl:message>
   <wsdl:message name="Msg_describeReqeust">
      <wsdl:part element="import1:describeBulkMR" name="parameters"/>
   </wsdl:message>
   <wsdl:message name="Msg_describeResponse">
      <wsdl:part element="import1:describeBulkMRResponse" name="result"/>
   </wsdl:message>

   <!-- portType element describes the abstract interface of a Web service -->
   <wsdl:portType name="bulkMR_PortType">
      <wsdl:operation name="describeBulkMR">
         <wsdl:input message="tns:Msg_describeReqeust"/>
         <wsdl:output message="tns:Msg_describeResponse"/>
      </wsdl:operation>
      <wsdl:operation name="executeBulkMR">
         <wsdl:input message="tns:Msg_executeRequest"/>
         <wsdl:output message="tns:Msg_executeResponse"/>
      </wsdl:operation>
   </wsdl:portType>

<!-- binding element tells us which protocols and binding styles to use. -->
   <wsdl:binding
```

```
         name="bulkMR_PortTypeBinding"
         type="tns:bulkMR_PortType">
        <soap:binding
        style="document"
        transport="http://schemas.xmlsoap.org/soap/http"/>
        <wsdl:operation name="describeBulkMR">
            <soap:operation soapAction=""/>
            <wsdl:input>
                <soap:body use="literal"/>
            </wsdl:input>
            <wsdl:output>
                <soap:body use="literal"/>
            </wsdl:output>
        </wsdl:operation>
        <wsdl:operation name="executeBulkMR">
            <soap:operation soapAction=""/>
            <wsdl:input>
                <soap:body use="literal"/>
            </wsdl:input>
            <wsdl:output>
                <soap:body use="literal"/>
            </wsdl:output>
        </wsdl:operation>
    </wsdl:binding>

    <!—service gives the endpoint URL of the service. -->
    <wsdl:service name="bulkMRService">
        <wsdl:port binding="tns:bulkMR_PortTypeBinding" name="Port1">
            <soap:address location="http://localhost:8000/ccx/bulkMRService"/>
        </wsdl:port>
    </wsdl:service>
</wsdl:definitions>
```

Listing 21 – XML Schema file (BulkMRFinal.xsd) that is imported into the WSDL file shown in Listing 20. The schema defines four global elements and seven global complex types. The complex types are used to compose the elements. The schema uses a number of XSD restriction techniques to tightly constrain the permitted values of certain elements (e.g. restrictions, patterns, enumerations).

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- edited with XMLSpy v2006 sp2 U (http://www.altova.com)
by david meredith (C18, cclrc, Daresbury Laboratory) -->
<xs:schema xmlns="http://e-htpx.ac.uk/bulkMR"
xmlns:xs="http://www.w3.org/2001/XMLSchema"
targetNamespace="http://e-htpx.ac.uk/bulkMR"
elementFormDefault="qualified"
attributeFormDefault="unqualified">
    <xs:complexType name="execute_BulkMR_Request">
        <xs:sequence>
            <xs:element name="email" type="xs:string"/>
            <xs:element name="jobid" type="xs:string"/>
            <xs:element name="sequence">
                <xs:simpleType>
                    <xs:restriction base="xs:string">
                        <xs:minLength value="1"/>
                        <xs:pattern value="[A-Ia-iK-Nk-nP-Tp-tV-Zv-z\s\*]*"/>
                    </xs:restriction>
                </xs:simpleType>
            </xs:element>
            <xs:element name="mrnum" minOccurs="0" default="20">
                <xs:simpleType>
                    <xs:restriction base="xs:int">
                        <xs:maxInclusive value="100"/> <xs:minInclusive value="0"/>
                    </xs:restriction>
                </xs:simpleType>
            </xs:element>
            <xs:element name="ensemnum" minOccurs="0" default="5">
                <xs:simpleType>
                    <xs:restriction base="xs:int">
                        <xs:maxInclusive value="20"/> <xs:minInclusive value="0"/>
                    </xs:restriction>
                </xs:simpleType>
            </xs:element>
            <xs:element name="mrprogram" default="molrep" minOccurs="0">
                <xs:simpleType>
                    <xs:restriction base="xs:string">
                        <xs:enumeration value="molrep"/> <xs:enumeration value="phaser"/>
                    </xs:restriction>
                </xs:simpleType>
            </xs:element>
            <xs:element name="evalue" minOccurs="0">
                <xs:simpleType>
                    <xs:restriction base="xs:float">
                        <xs:minInclusive value="0.00000000001"/><xs:maxInclusive value="1000.0"/>
                    </xs:restriction>
                </xs:simpleType>
            </xs:element>
            <xs:element name="ignore" type="pdbIgnoreCodes" maxOccurs="1" minOccurs="0"/>
            <xs:element name="nmasu" minOccurs="0">
                <xs:simpleType>
                    <xs:restriction base="xs:int">
                        <xs:minInclusive value="1"/>
                    </xs:restriction>
                </xs:simpleType>
            </xs:element>
            <xs:element name="include" type="includeCodes" minOccurs="0"/>
            <xs:element name="maprogram" default="mafft" minOccurs="0">
                <xs:simpleType>
                    <xs:restriction base="xs:string">
                        <xs:enumeration value="mafft"/>
                        <xs:enumeration value="clustalw"/>
                    </xs:restriction>
                </xs:simpleType>
            </xs:element>
            <xs:element name="pack" default="5" minOccurs="0" maxOccurs="1">
                <xs:simpleType>
                    <xs:restriction base="xs:int">
                        <xs:minInclusive value="0"/>
                    </xs:restriction>
                </xs:simpleType>
            </xs:element>
            <xs:element name="debug" type="xs:boolean" default="false" minOccurs="0"/>
            <xs:element name="ssmsearch" type="xs:boolean" default="false"
```

```
            minOccurs="0"/>
            <xs:element name="scopsearch" type="xs:boolean" default="false"
            minOccurs="0"/>
            <xs:element name="pqssearch" type="xs:boolean" default="false"
            minOccurs="0"/>
            <xs:element name="fastalocal" type="xs:boolean" default="true"
            minOccurs="0"/>
            <xs:element name="labin">
                <xs:complexType>
                    <xs:sequence>
                        <xs:element name="F">
                            <xs:simpleType>
                                <xs:restriction base="xs:string">
                                    <xs:pattern value="[a-zA-Z0-9_\-]*"/>
                                </xs:restriction>
                            </xs:simpleType>
                        </xs:element>
                        <xs:element name="SIGF">
                            <xs:simpleType>
                                <xs:restriction base="xs:string">
                                    <xs:pattern value="[a-zA-Z0-9_\-]*"/>
                                </xs:restriction>
                            </xs:simpleType>
                        </xs:element>
                        <xs:element name="FreeR_flag">
                            <xs:simpleType>
                                <xs:restriction base="xs:string">
                                    <xs:pattern value="[a-zA-Z0-9_\-]*"/>
                                </xs:restriction>
                            </xs:simpleType>
                        </xs:element>
                    </xs:sequence>
                </xs:complexType>
            </xs:element>
            <xs:element name="hklin">
                <xs:simpleType>
                    <xs:restriction base="xs:string">
                        <xs:pattern value="[a-zA-Z0-9_\-]*.mtz"/>
                    </xs:restriction>
                </xs:simpleType>
            </xs:element>
            <xs:element name="hklout" type="xs:string"/>
            <xs:element name="xyzout" type="xs:string"/>
        </xs:sequence>
    </xs:complexType>
    <xs:complexType name="pdbIgnoreCodes">
        <xs:sequence>
            <xs:element name="pdbCode" minOccurs="0" maxOccurs="unbounded">
                <xs:simpleType>
                    <xs:restriction base="xs:string">
                        <xs:pattern value="[0-9][a-zA-Z0-9][a-zA-Z0-9][a-zA-Z0-9]"/>
                    </xs:restriction>
                </xs:simpleType>
            </xs:element>
        </xs:sequence>
</xs:complexType>
    <xs:complexType name="execute_BulkMR_Response">
```

```
        <xs:sequence>
            <xs:element name="stringArrayResponse" type="ArrayOf_xsd_string"/>
        </xs:sequence>
    </xs:complexType>
    <xs:complexType name="ArrayOf_xsd_string">
        <xs:sequence>
            <xs:element name="item" type="xs:string" maxOccurs="unbounded"/>
        </xs:sequence>
    </xs:complexType>
    <xs:complexType name="describeRequest">
        <xs:sequence/>
    </xs:complexType>
    <xs:complexType name="describeResponse">
        <xs:sequence>
            <xs:element name="description" type="xs:string"/>
        </xs:sequence>
    </xs:complexType>
    <xs:complexType name="includeCodes">
        <xs:sequence>
            <xs:element name="chainID" maxOccurs="unbounded" minOccurs="0">
                <xs:simpleType>
                    <xs:restriction base="xs:string">
                        <xs:pattern value="[0-9][a-zA-Z0-9][a-zA-Z0-9][a-zA-Z0-9]_[a-zA-Z]"/>
                    </xs:restriction>
                </xs:simpleType>
            </xs:element>
        </xs:sequence>
    </xs:complexType>
    <xs:element name="executeBulkMR" type="execute_BulkMR_Request"/>
    <xs:element name="executeBulkMRResponse" type="execute_BulkMR_Response"/>
    <xs:element name="describeBulkMR" type="describeRequest"/>
    <xs:element name="describeBulkMRResponse" type="describeResponse"/>
</xs:schema>
```

10. Data Binding and Validation

Data binding and validation applies to both loosely typed and strongly typed services.

- Data binding refers to parsing the message data into a format that can be used internally by the Web service implementation. For example, with regard to loosely typed services, this may involve converting a loosely typed RPC string parameter that represents an XML fragment into an object, or marshalling an XML document sent as an attachment into a DOM tree. With regard to tightly bound services, this generally includes marshalling XML Documents into object graphs. In the context of XML, a document that is said to be 'well-formed' can be successfully bound, whereas a mal-formed document will fail data binding (due to malformed elements and missing characters for example).

- Data validation occurs after data binding and is the process of validating the data values for correctness. In the context of XML, a well-formed document may fail validation because an element's value does not conform to its schema.

According to the service implementation and configuration of the SOAP engine, data binding and validation can be:

- Performed by the SOAP engine, so the service operation is presented with an object representation of the payload, or,
- Separated from the SOAP engine and performed by the business logic implemented by the developer, so the Web service receives the raw XML payload (a 'message' style implementation).

10.1. Data-Binding and Validation Performed by the SOAP Engine

Data binding and validation can be performed by the internal binding-framework of the SOAP engine so that the Web service implementation/ operation is presented with an object representation of the message parameters.

Advantages:

- *Simplicity*: The developer is hidden from the complexity of performing data binding and validation, and is presented with an object model representation of the Web service parameters that can be used directly.

Disadvantages:

- *Restricted choice of binding / validation framework*: For many SOAP engine implementations, you cannot use custom or preferred data binding and validation frameworks.
- *Semi-standardized data bindings which are often not 100% schema compliant:* In many SOAP engine implementations (e.g. JAX-PRC, Axis 1.4 [9]), semi-standardized data binding frameworks are used for the generation of client and server stub/ skeleton classes. In our experience, SOAP engine data binding frameworks are often not 100% Schema complaint, and often do not support the more advanced features of XML Schema (e.g. xsd:patterns). We believe that this has been a major source of ambiguity and in our experience, this has often been a source of error that is beyond immediate control of the developer. However, it must be noted that most of the second-generation SOAP engine frameworks adopt 100% schema compliant data binding frameworks, (e.g. the newly released JAX-WS reference implementation that binds and validates with JAX-B [7], and Axis2 that binds and validates with XMLBeans [8]).
- *May need to manually un-marshal objects back into XML if the raw XML is required for additional processes*: If you require the original XML that was sent to the service for additional 'out-of-band' processes (for storage in a database for example), you may have to reconstruct the XML from the object representation of the payload (i.e. un-marshal the objects back into XML).

10.2. Examples of Data-Binding and Validation Performed by the SOAP Engine

10.2.1. Axis 1.4 (JAX-RPC 1.1 Implementation)

The Axis 1.4 is a popular implementation of the JAX-RPC 1.1 specification. The framework provides the WSDL2Java command line tool for building a Web service (or client) implementation from a WSDL file. This is known as the 'WSDL first' or 'contract driven' approach and is discussed further in Section 11.2. After executing WSDL2Java on the tightly bound WSDL file shown in Listing 20, an implementation of the Web service is generated and is shown in Listing 22 (additional artefacts that are used for binding and validation are also generated but not shown).

The following important points must be emphasised:

- In the example provided, the Web service operation parameter list and return types are 'un-wrapped' object model representations of the XML request and response messages. The request parameters correspond to the data contained within the <executeBulkMR> and <describeBulkMR> schema elements (shown in Listing 22). The same applies to the return types where 'un-wrapped' objects are returned which correspond to the data contained within the <executeBulkMRResponse> and <describeBulkMRResponse> elements. Using Axis customisations, it is also possible to receive and return single 'document' objects that correspond to the global request and response elements.
- The Axis 1.4 binding and validation framework ***is not*** 100% XML schema compliant. Consequently, not all of the schema restrictions that are defined in the XML schema shown in Listing 21 are enforced. For example, the <xs:pattern> restrictions (i.e. regular expressions) that are used to constrain a number of key element values are not imposed (e.g. the string values of the <sequence>, <pdbCode> and <chainID> could pass validation with any possible value). This has very serious implications to the Web service as invalid data can go un-detected and penetrate the business logic of the service. It must be noted however, that the later version of Axis (Axis2) binds and validates with a 100% XML schema compliant toolkit (XMLBeans) and, consequently, these validation failures would not occur with Axis2. However, Axis 1.x is currently a very popular JAX RPC implementation framework and this problem must be recognised by developers.

Listing 22 – An Axis 1.4 auto-generated Web service implementation class that delegates data binding and validation to the SOAP engine. The service implementation has been built from the WSDL file shown in Listing 20. A crucial limitation of this auto-generated implementation, is that not all object parameters will be valid according to the rules defined by the schema shown in listing 21 (most notably, the xsd:pattern restrictions are ignored). This is because the binding and validation toolkit used is not 100% XML schema compliant.

```
package uk.ac.e_htpx.bulkmr;
import uk.ac.e_htpx.bulkMR.*;   // import auto-generated helper classes and artefacts

public class BulkMR_PortTypeBindingImpl implements
uk.ac.e_htpx.bulkmr.BulkMR_PortType{

  /** Web service operation */
  public java.lang.String describeBulkMR() throws java.rmi.RemoteException {
    // TODO add business logic here
    return null;
  }

  /** Web service operation */
  public String[] executeBulkMR(String email, String jobid, String sequence, Integer
                    mrnum, Integer ensemnum, Execute_BulkMR_RequestMrprogram
                    mrprogram, Float evalue, String[] ignore, Integer nmasu, String[]
```

```
                include, Execute_BulkMR_RequestMaprogram maprogram, Integer
                pack, Boolean debug, Boolean ssmsearch, Boolean scopsearch,
                Boolean pqssearch, Boolean fastalocal,
                Execute_BulkMR_RequestLabin labin, String hklin, String hklout,
               String xyzout) throws java.rmi.RemoteException {

        /**  TODO implement business logic here */
        /**
         * WARNING not all parameters will be valid according to schema,
         *  e.g. ‘sequence’ param
         */
        return null;
    }
}
```

10.2.2. JAX-WS

JAX-WS 2.0 (JSR 224 [14]) is the next generation Web service API for Java and replaces JAX-RPC 1.1. The framework was renamed to JAX-WS to help mitigate the common misunderstanding that Web services are just another way of doing RPC. JAX-WS, like Axis, can be deployed to any servlet container and is not dependent on deployment to a J2EE Application Server. The most important feature of the JAX-WS framework is that the data binding and validation is delegated to the JAXB 2.0 framework, which supports all of the W3C XML schema features (100% schema compliant). Consequently, the Web service message data can be fully validated according to the rules defined in the schema(s) imported within the <wsdl:types> element. The JAX-WS RI (reference implementation) provides the ‘wsimport’ (and ‘wsgen’) command line tool to generate the Web service implementation from a WSDL file. Many customisations are also supported to configure and optimise data binding and validation. After executing ‘wsimport’ on the tightly bound WSDL file shown in Listing 21, an implementation of the Web service is generated and is shown in Listing 23 (additional JAXB artefacts that are used for binding and validation are also generated but not shown).

The following important points must be emphasised:

- In the example provided, the Web service operation parameter list and return types are ‘un-wrapped’ object model representations of the XML request and response messages. The request parameters correspond to the data contained within the <executeBulkMR> and <describeBulkMR> schema elements (shown in Listing 20). The same applies to the return types where ‘un-wrapped’ objects are returned which correspond to the data contained within the <executeBulkMRResponse> and <describeBulkMRResponse> elements. Using JAX-WS customisations, it is also possible to receive and return single ‘document’ objects that correspond to the global request and response elements.
- The XML schema restrictions are (optionally) applied during validation. Consequently, the <xs:pattern> restrictions (i.e. regular expressions) can be enforced when binding and validating the payload. This means that only valid data will enter the business logic of the Web service.
- JAX-WS relies on Java 1.5 annotations, such as the ‘@WebService’ and ‘@WebMethod’ annotations.

Listing 23 – A JAX-WS auto-generated Web service implementation class that delegates data binding and validation to the SOAP engine (via JAXB). The service implementation has been built from the WSDL file shown in Listing 20. The key advantage of the JAX-WS implementation over the JAX-RPC implementation shown in Listing 21 is that the parameter values can be fully validated according to the rules defined in the schema shown in Listing 21.

```
import javax.jws.WebService;
import uk.ac.e_htpx.bulkmr.* ; // import auto-generated binding classes and artifacts

@WebService(
serviceName = "bulkMRService",
portName = "Port1",
endpointInterface = "uk.ac.e_htpx.bulkmr.BulkMRPortType",
targetNamespace = "http://e-htpx.ac.uk/bulkmr",
wsdlLocation = "WEB-INF/wsdl/BulkMRService2/bulkMRService.wsdl")
public class BulkMRService2 implements uk.ac.e_htpx.bulkmr.BulkMRPortType {

   @WebMethod
   public String describeBulkMR() {
      // TODO implement business logic
      return null;
   }

   @WebMethod
   public ArrayOfXsdString executeBulkMR(String email,
         String jobid, String sequence, Integer mrnum, Integer ensemnum,
         String mrprogram, Float evalue, PdbIgnoreCodes ignore, Integer nmasu,
         IncludeCodes include, String maprogram, Integer pack, Boolean debug,
         Boolean ssmsearch, Boolean scopsearch, Boolean pqssearch, Boolean
         fastalocal, Labin labin, String hklin, String hklout, String xyzout) {

        /**  TODO implement business logic here */
        /** TODO implement and return valid response (ArrayOfXsdString)*/
        return null;
    }
}
```

10.3. Data-Binding and Validation Separated from the SOAP Engine

Data binding and validation can be separated from the SOAP engine and left to the responsibility of the business logic implemented by the developer. This approach requires a ‘message style’ service endpoint implementation that is passed a raw, un-bound XML payload (i.e. the service and SOAP engine are configured so that data binding and validation is ‘turned off’). In doing this, the business logic binds and validates the XML payload within the Web service operation using APIs and tools of choice. This can be performed using direct XML manipulation API’s, such as DOM or SAX for example. However, it is more efficient and functional to use a dedicated XML schema binding and validation framework such as JAXB[7] or XMLBeans[8]. Developers may still manipulate XML in the familiar format of objects (courtesy of the binding framework), but there is no dependency upon the SOAP engine. In doing this, the XML schema(s) that are imported within the <types> element of the WSDL file are manually compiled to produce an object-binding library. When using

XMLBeans, you can compile the schema using the ‘scomp’ command line tool, which generates a .jar file (library) containing the required binding and validation artifacts. Similarly, JAXB provides the ‘xjc’ command line compiler. The compiled object library is imported directly within the Web service implementation to bind and validate the unbound XML after invocation of the service. In doing this, it is easy to both parse and traverse the XML payload, and validate the XML according the rules defined by the schema.

Advantages:

- *Choice of preferred data binding / validation framework*: This could lever the preferred, or more powerful features of a chosen binding framework. For example, we have found the XML cursor functionality of the XMLBeans toolkit especially useful for isolating and parsing arbitrary XML that is defined in place of xsd:any and xsd:anyType elements.
- *Clear separation of roles*: The SOAP engine and the data binding/validation framework become clearly separated into 'communication-specific' and 'data-specific' roles.
- *On-Demand XML Document Construction and Validation for ‘Out-of-Band’ Processes:* This clear separation of roles means that XML messages/ documents can be constructed and validated for additional ‘out-of-band’ processes, for example, when constructing messages over an extended period of time (e.g. graphically through a GUI) and especially for the purposes of persistence (e.g. saving validated XML to a database). The separation of the data binding from the SOAP engine is gaining popularity in the next generation of SOAP engines that are now beginning to implement 'pluggable' data binding frameworks.

Disadvantages:

- *Manual Binding/ Validation - Increased Complexity:* The developer is responsible for manually parsing, traversing and extracting data from the unbound XML payload that is received by the service. Furthermore, if opting to use manual XML traversing API’s such as SAX or DOM, then a good understanding of XML and XML namespaces is essential.
- *Manually Construct Sensible Exceptions:* The developer is also responsible for throwing sensible exceptions with the occurrence of payload errors such as mal-formed XML and erroneous values.

10.4. Examples of Data-Binding and Validation Separated from the SOAP Engine

10.4.1. Axis 1.4 and Separated Data-Binding and Validation Example

The Axis framework provides ‘Message’ style service endpoint methods. Here, the endpoint implementations access the XML message payloads directly, and do not perform any data binding or validation leaving the developer to create logic to parse the request and response messages directly within the methods. The method signatures shown in Listing 24 are provided by Axis (refer to the Axis documentation for further details):

Listing 24 – Axis 'Message' style endpoint methods that receive the raw, unbound XML payload of the SOAP message.

```
   public Element [] method(Element [] bodies);
   public SOAPBodyElement [] method (SOAPBodyElement [] bodies);
   public Document method(Document body);
  public void method(SOAPEnvelope req, SOAPEnvelope resp);
```

Any of these service endpoint method signatures may be used in conjunction with the WSDL file that is shown in listing 20. However, if a WSDL interface defines multiple/ overloaded operations as shown in Listing 25 (e.g. 'describeBulkMR' and 'executeBulkMR'), then the Web service endpoint methods should also be named after the <wsdl:operation> elements that are defined in the WSDL binding. In our experience, we found that the 'soapAction' attribute of the <soap:operation> element was ignored by Axis, which instead relied on method naming to distinguish between overloaded operations. Furthermore, the 'soapAction' parameter value may be empty. Consequently, we recommend that the endpoint methods be named after the <wsdl:operation> elements. Listing 25 shows an implementation example which uses the XMLBeans toolkit to parse the SOAPEnvelope request, and generate a validated SOAPEnvelope for the response. The code example assumes that the schema shown in Listing 21 has been compiled using the 'scomp' command line tool (e.g. '<XMLBeans_HOME>/bin/scomp BulkMRSchema.xsd –out ehtpxXSDlib.jar'), and that the generated .jar file has been included in the CLASSPATH of the service implementation. It would be equally valid to parse, bind and validate the request and response SOAPEnvelopes using SAX, DOM or JAXB.

Listing 25 – Axis Message style service implementation class that delegates data binding and validation to the business logic of the service. The service implementation is used in conjunction with the WSDL file shown in Listing 20.

```
 package uk.ac.ehtpx.bulkMR.message;
 import uk.ac.eHtpx.bulkMR.*;
 import ….

   public class bulkMRMessageService {

      /** Web service method */
      public Element[] describeBulkMR(Element [] elems) {
         // TODO add business logic here – here we simply return the input elements
         return elems;
      }

      /** Web service Method  */
      public void executeBulkMR(SOAPEnvelope req, SOAPEnvelope resp) throws
                  javax.xml.soap.SOAPException {

         // Get the (last) SOAPBodyElement (this service style is doc/literal-wrapped
         // and therefore, there should only be one "executeBulkMR" soapbody element
         // having same name as this operation !).
```

```
    SOAPBody reqBody = req.getBody();
    Iterator it = reqBody.getChildElements();
    SOAPBodyElement be;
    String content = null;
    while(it.hasNext()){
       be = (SOAPBodyElement)it.next();
       content = be.toString();
    }

    // Use the XMLBeans compiled schema library to parse the document (parsing doesn't
    // do validation – it only checks for well-formedness of XML docs).
    ExecuteBulkMRDocument doc = null;
    try {
       doc = ExecuteBulkMRDocument.Factory.parse(content);
    } catch(XmlException e){
       e.printStackTrace();
       throw new javax.xml.soap.SOAPException( "ERROR - document is not well
             formed "+e.getMessage());
    }

    // Use the compiled schema library (xmlbeans toolkit) to do validation
    try {
       this.validateExecuteBulkMRDocument(doc);
    }catch(SOAPException ex){
       throw ex;  // rethrow
    }

    // get the validated request information for processing.
    ExecuteBulkMRRequest reqdoc = doc.getExecuteBulkMR();
    String email = reqdoc.getEmail();
    String sequence = reqdoc.getSequence(); // sequence is validated !

    /**  TODO implement business logic here */
    /**  TODO can use XML Document to save directly to a DB or for other processes */

    // create a valid response document using xmlbeans
    ExecuteBulkMRResponseDocument respDoc =
            ExecuteBulkMRResponseDocument.Factory.newInstance();
    ExecuteBulkMRResponse r = respDoc.addNewExecuteBulkMRResponse();
    ArrayOfXsdString sr = r.addNewStringArrayResponse();
    sr.addItem("this is the response document");
    sr.addItem("Bulkmr job submitted ok - download results from....");

    // create Dom document from xml beans object and add it to the SOAPEnvelope resp
    Document respDocument = this.createDomDocument(respDoc.toString());
    SOAPBody bod = resp.getBody();
    SOAPElement elem = bod.addDocument(respDocument);

    // no return object required, axis automatically sends back the SOAPEnvelope
    // response argument
}
```

```
  /**
    * Non Web service function used to validate ExecuteBulkMRDocuments
    * The ExecuteBulkMRDocument object is compiled from the schema
    * using xmlbeans toolkit.
    */
   private boolean validateExecuteBulkMRDocument(ExecuteBulkMRDocument doc)
   throws SOAPException{
      ArrayList validationErrors = new ArrayList();
      XmlOptions validationOptions = new XmlOptions();
      validationOptions.setErrorListener(validationErrors);
      boolean valid = doc.validate(validationOptions);
      if(valid){
         // TODO success custom actions – if any
      } else {
         // collate the validation error messages and wrap as SOAPException
         Iterator iter = validationErrors.iterator();
         String errors  = "";
         while (iter.hasNext()) {
            errors += (">> " + iter.next() + "\n");
         }
         throw new SOAPException( "ERROR - Document Is Not Valid \n "+errors);
      }
      return true;
   }
}
```

10.4.2. JAX-WS and Separated Data-Binding and Validation Example

The JAX-WS specification also provides 'message' style Web service implementation endpoints that facilitate direct access to the XML payload by implementing the 'Provider' interface and 'javax.xml.ws.WebServiceProvider' annotation. Provider endpoints must declare the '@WebServiceProvider' annotation rather than the '@WebService' annotation as shown in Listing 26. The @WebServiceProvider annotation also supports properties for declaring the location of the WSDL file, the target namespace of the WSDL file, the WSDL port name and the service name. The '@ServiceMode' annotation may also be used to convey whether the endpoint requires access to the whole SOAP message (Service.Mode.MESSAGE) or just to the payload of a particular request (Service.Mode.PAYLOAD). The JAX-WS documentation provides further details. Listing 26 shows an example of a 'message' style service implemented with JAX-WS using the 'Provider' interface and '@WebServiceProvider' annotation. The data-binding and business logic of the Web service could be identical to that shown in Listing 25.

Listing 26 – JAX-WS Message style service implementation class that implements the 'Provider' interface and '@WebServiceProvider' annotation and provides direct access to the XML payload. Data binding and validation is implemented by the business logic rather than by the SOAP engine. The service implementation is used in conjunction with the WSDL file shown in Listing 20. The data-binding and business logic of the Web service could be identical to that shown in Listing 25.

```
@ServiceMode(value=Service.Mode.PAYLOAD)
@WebServiceProvider(wsdlLocation="WEB-INF/wsdl/bulkMRService.wsdl",
portName="Port1",
targetNamespace="http://e-htpx.ac.uk/bulkmr",
serviceName="ehtpxProviderService")
public class bulkMRImpl implements Provider<Source> {

   /** Web service method – implements invoke of interface javax.xml.ws.Provider */
   public Source invoke(Source source) {
      try {
         /**
          * TODO implement request processing,
          * here shown with DOM but could equally use XMLBeans as shown  in Listing 25
          */
         DOMResult dom = new DOMResult();
         Transformer trans = TransformerFactory.newInstance().newTransformer();
         trans.transform(source, dom) ;
         Node node = dom.getNode();
         Node root = node.getFirstChild();

         /**  TODO implement business logic here – see previous code listing */
         /**  TODO construct a valid response document and return */
         Source response = ....
         return response;
      } catch(Exception e) {
         throw new RuntimeException("Error in provider endpoint", e);
      }
   }
```

11. Code First or WSDL First

For platform independence and Web service interoperability, the WSDL interface should not reference or have dependencies upon any technical API other than XML Schema. However, the complexity of XML schema and over-verbosity of WSDL is a major concern in practical development of Web services. As a result, two divergent practices for developing Web services and WSDL have emerged, the ‘code first’ approach (also known as ‘bottom up’) and 'WSDL first' approach (also know as ‘top down’ or ‘contract driven’). The code first approach, which is often implemented in JAX-RPC environments, involves auto-generation of the WSDL file from service implementation classes using tools that leverage reflection and introspection. In doing this, it could be stated, “*the WSDL describes the underlying implementation*.”
Alternatively, the WSDL first approach involves writing the original WSDL and XML Schema, and generating/ writing service implementation classes from the WSDL file. In doing this, it could be stated, *“The underlying code implements the interface defined by a WSDL.”* The advantages and disadvantages of each approach are summarized below;

11.1. Code First

Advantages

- The 'code first' approach is often appealing because of its simplicity. Developers are hidden from the technical details of writing XML and WSDL, which often encourages developers who are unfamiliar with XML to participate.

Disadvantages

- Generating WSDL files from source code may introduce dependencies upon the implementation language. This is especially relevant when considering older, JAX-RPC implementations, which can lead to interoperability issues across different platforms (e.g. differences in how Java and .NET serialize types that may be value types in one language but are reference/ nillable objects in the other).
- Not all language specific data types can be mapped into interoperable XML. This issue is apparent when non-standard data types are used in the service implementation, or when there are different possible XML data mappings for that non-standard type. Listing 27 shows a few examples that highlight these issues:

Listing 27 – Examples of non-interoperable language specific types that cannot be mapped into interoperable XML. The examples show method endpoint declarations and the resulting XML SOAP fragments. The Java2WSDL tool has introduced dependencies upon the implementation language (in this case Axis and Java).

```
public void sendImage(java.awt.Image sendImage) {}

 <element name="sendImage">
  <complexType>
   <sequence>
    <element name="image" type="apachesoap:Image"/>
   </sequence>
  </complexType>
 </element>

public void sendOctet(org.apache.axis.attachments.OctetStream sendOctet) {}

 <element name="sendOctet">
  <complexType>
   <sequence>
    <element name="octet" type="apachesoap:octet-stream"/>
   </sequence>
  </complexType>
 </element>

public DataHandler sendDH(DataHandler sendDHDataHandler) {}

<element name="sendDH">
   <complexType>
    <sequence>
     <element name="dataHandler" type="apachesoap:DataHandler"/>
    </sequence>
   </complexType>
</element>
```

- WSDL created from source code is less strongly typed than WSDL that is created from the original XML Schema. Indeed, many of the more powerful features of XML Schema are often lost when using automatic WSDL generators (e.g. xsd:patterns). This is particularly important for any Web service handling attachments as the MIME/DIME related information in the binding section of the WSDL cannot be added. The data type in the generated implementation can map to different content-types and thus loses precise control. Similarly in the case of arrays, WSDL tools often ignore the length of an array while generating the WSDL and the client can embed an array of arbitrary length into the SOAP message.

11.2. WSDL First

Advantages

- Platform and language interoperability issues are prevented, because both the client and server are working from a common set of interoperable XML Schema types.
- Defining a common platform-independent type system facilitates separation of roles, whereby client side developers can work in isolation from server side developers. In our experience, this greatly increases productivity and simplifies development, especially for large distributed applications where developers may be geographically separated.
- Tools for the auto-generation of client and server stubs and for creating properly formatted WSDL can be used to minimize coding. For example, the Netbeans 5.5 IDE supports very functional WSDL and XML schema authoring and editing.
- The data types available in different programming languages are only subset of those available in XML Schema. Consequently, developers can utilize the more expansive XML schema data types within the WSDL document, which may enhance processing and efficiency. Examples include "xsd:hexBinary", "xsd: base64Binary", octet-stream and multimedia related content-types.

Disadvantages

- The developer requires at least a reasonable knowledge of XML Schema and of WSDL.
- This approach is initially more time consuming.

In our experience, the WSDL first approach is certainly the most suitable for developing robust, interoperable Web services that make tight constraints on the data. However, we certainly found the code first approach convenient for rapid prototyping, especially when combined with loose data typing. Nevertheless, when opting for the code-first approach, great efforts should be made to ensure interoperability across different platforms.

12. WSDL Modularisation

The WSDL specification and the WS-I basic profile recommend the separation of WSDL files into distinct modular components in order to improve re-usability and manageability. These modular components include;

1. XML schema files, for modeling data and type information. Schema files can be incorporated into the abstract WSDL file through the use of <xsd:include> and <xsd:import> as children of the <wsdl:types> element.

2. An Abstract WSDL file, for defining <wsdl:type>, <wsdl:message> and <wsdl:portType> (business operation) elements.
3. A Concrete WSDL file, for defining <wsdl:service> and <wsdl:binding> elements. The concrete WSDL file includes/imports the abstract WSDL file via <wsdl:import>/ <wsdl:include> elements.

12.1. Separate XML Schema Files

Moving the type declarations of a Web service into their own documents is recommended as data can be modeled in different documents according to namespace requirements. XML Schema declares two elements; <xsd:include> and <xsd:import> which are both valid children of the <wsdl:types> element (<xsd:include> is used when two schema files have the same namespace and <xsd:import> is used to combine schema files from different namespaces). In doing this, complex data types can be created by combining existing documents. Listing 28 demonstrates the practical mixing of schema with <xsd:include> and <xsd:import> using schema fragments taken from the e-HTPX project [].

- **AdminException.xsd** describes some common data types used in different schemas of the e-HTPX project. All elements in this schema have the prefixed namespace (xmlns:tns="urn:ehtpx-process").
- **StrategyImageArray.xsd** has a different prefixed namespace underlined in blue (http://clyde.dl.ac.uk:8080/process). This schema uses elements defined in the "adminException.xsd and therefore uses <xsd:import> to import this schema prefixing its elements with "ns1" (italic-red).
- **Index.xsd** has the same namespace as AdminException.xsd and therefore uses <xsd:include> to include AdminException.xsd, prefixing local elements and those included from "AdminException.xsd with the "tns" prefix (underlined-blue). <xsd:import> is used to import StrategyImageArray.xsd using the prefix "ns1" to fully qualify the imported elements (italic-red). By combining the data model schemas, an abstract WSDL file need only import Index.xsd, which keeps the WSDL simple, less verbose and easy to manage. Most of XML-Object mapping tools are more efficient in parsing separate schema files rather than data types embedded in the WSDL.

Listing 28 – Practical Mixing of Separate XML Schema Files.

AdminException.xsd
```
<schema targetNamespace="urn:ehtpx-process"
         xmlns:tns="urn:ehtpx-process" ……………….>
    <element name="CrystalLatticeElement"
       type="tns:CrystalLattice" nillable="true" />
    <element name="StrategyImageElement"
       type="tns:StrategyImage" nillable="true" />
    <element name="StrategySweep"
       type="tns:StrategySweep" nillable="true" />
    ………………………………..
</schema>
```

StrategyImageArray.xsd
```
<schema
  targetNamespace="http://clyde.dl.ac.uk:8080/process"
    xmlns:tns="http://clyde.dl.ac.uk:8080/process"
   xmlns:ns1="urn:ehtpx-process" .................>
   <import namespace="urn:ehtpx-process" schemaLocation="AdminException.xsd" />
     <complexType name="ArrayOfStrategyImage">
       <sequence>
         <element name="array" type="ns1:StrategyImage" minOccurs="0"
         maxOccurs="unbounded" />
       </sequence>
    </complexType>
    <element name="ArrayOfStrategyImageElement" type="tns:ArrayOfStrategyImage" />
</schema>
```

Index.xsd
```
<schema targetNamespace="urn:ehtpx-process"
   xmlns:tns="urn:ehtpx-process"
   xmlns:ns1="http://clyde.dl.ac.uk:8080/process">
  <include schemaLocation="AdminException.xsd" />
  <import  namespace="http://clyde.dl.ac.uk:8080/process"
     schemaLocation="StrategyImageArray.xsd" />
  <complexType name="IndexOutput">
       <sequence>
           <element name="crystal_lattice"
            type="tns:CrystalLattice" nillable="true" />
           <element name="resolution" type="double" />
           <element name="status" type="string"  nillable="true" />
           <element name="strategy_image_list"  type="ns1:ArrayOfStrategyImage"
            nillable="true" />
       </sequence>
   </complexType>
</schema>
```

12.2. Separate Abstract WSDL File

The abstract.wsdl file defines *what* the Web service does by defining the data types and business operations of the Web service. The abstract file imports XML schema(s) as immediate children of the <wsdl:types> element, and defines different <wsdl:message> and <wsdl:portType> elements. An application can have different abstract WSDL files with similar business logic to address the interface requirements of various clients. The WSDL specification discourages the use of overloaded methods (polymorphism) in the WSDL, which is the building block of Object Oriented programming. This limitation can be overcome by declaring different Abstract WSDL documents.

12.3. Separate Concrete WSDL File

The concrete.wsdl file defines *how* and *where* to invoke a service by defining network protocol and service endpoint information with the <wsdl:binding> and <wsdl:service> elements. The concrete.wsdl file incorporates the abstract.wsdl file through the use of <wsdl:import> or <wsdl:include> elements. These elements should be the first immediate children of the <wsdl:definitions> element (<wsdl:include> is used when two WSDL files have the same namespace and <wsdl:import> is used to combine WSDL files from different namespaces). This approach greatly improves component re-usability as the same abstract WSDL file can have multiple service bindings and URL locations to support different transport protocols. A good example is a Web service that supports both DIME and MIME attachments. MIME and DIME related information is referenced in the <wsdl:binding> section of two separate concrete WSDL documents that re-use the same abstract WSDL document. Listing 29 illustrates the correct syntax of a concrete WSDL file.

Listing 29 – Concrete WSDL Example.

```
<wsdl:definitions  name="Index"
  targetNamespace="http://clyde.dl.ac.uk:8080/
  process/services/IndexService"
  xmlns:import1="http://clyde.dl.ac.uk:8080/process/Index "
  xmlns:import2="http://sax.xml.org"
  xmlns:xsd1="http://clyde.dl.ac.uk:8080/
  process/services/Index.xsd1">
   <wsdl:import namespace=
     “http://clyde.dl.ac.uk:8080/process/Index”
     location=“Index.wsdl”/>
  <wsdl:binding> ….</ wsdl:binding>
  <wsdl:service> ….</ wsdl:service>
< /wsdl:definitions>
```

13. Tools

A combination of good tools can dramatically speed up creation of XML schema and WSDL. XML Spy [11] is currently one of the best visual tools for XML data modeling but the free personal version is limited and has no support for xsd:import/ includes. Netbeans 5.5 [12] is a free to use Java IDE with complete support for visual XML Schema and WSDL authoring. The WS-I organization also provide a range of testing tools for testing WSDL for WS-I compliance [6].

14. Conclusions

Data binding and validation in Web services is intrinsically linked to Web service style and the strength of data typing. However, developments in the field of Web services have largely focused on the RPC style services rather than on the Document style which is more powerful for the binding and validation of complex data. This is apparent in the tools provided by vendors of JAX-RPC/ SOAP engine implementations. For the most part, RPC style services are limited with regard to the description and validation of data, and can lead to serious interoperability issues. Real applications involving complex domain data require powerful data modelling and validation support. For these applications, the RPC/ encoded Web service style involving simple types and SOAP encoded complex types is wholly inadequate. The RPC/literal style improves upon the RPC/encoded style and is WS-I compliant. However, the RPC/literal style also has limitations for the validation of data. This is because the RPC method binding rules introduce additional complexities when integrating nested sub-elements with custom schema defined namespaces; i.e. data binding and validation must also account for the RPC element naming conventions and WSDL namespaces, rather than solely binding and validating plain XML instance documents (as in the Document orientated approach). The RPC style can also produce interoperability issues as many automatic WSDL generation tools introduce technical dependencies upon implementation languages. As a result, the RPC style is increasingly being referred to as 'CORBA with brackets' in the Web service community. Loosely typed services provide an alternative approach by encapsulating data within generic types. Loosely typed services are easy and convenient to develop and are suitable in a number of scenarios. However, loose typing introduces a different set of limitations, mainly associated with the additional manual negotiation that is required between consumer and provider to establish the format of encapsulated data. In contrast to loose typing, strongly typed services fully define all necessary information in the WSDL interface with value constraints where necessary. Strongly typed services use 100% standard XML schema as the type system and conform to the Document orientated Web service style. The exchange of XML instance documents facilitates complex data modelling, loose and tight data typing where necessary, and full document validation support. Since XML is a platform independent type system, Document style services are also platform and transport agnostic, where abstract WSDL definitions can be bound to different transport protocols defined in separate concrete WSDL files. Binding and validation can be performed either by the SOAP engine, or separately when implementing message style service endpoint implementations that access the raw XML payload. When the binding and validation framework is separated from the SOAP engine, the SOAP engine and binding/validation frameworks have clearly separated communication and data-centric roles. This levers the binding/validation framework for use in additional 'out-of-band' processes such as persistence. We also recommend the use of dedicated data binding/ validation frameworks that are 100% XML schema compliant for the construction of XML documents and messages. In doing this, the more powerful features of XML Schema can be levered. The main disadvantage of the Document style is the potential for increased complexity over RPC; developers require at least a reasonable understanding of XML and WSDL and are required to take the 'WSDL first' approach to Web service design.

References

[1] Web Services Description Language (WSDL) 1.1, http://www.w3.org/TR/2001/NOTE-wsdl-20010315
[2] Universal Description, Discovery and Integration, UDDI Version 3.0.2, http://uddi.org/pubs/uddi_v3.htm
[3] Simple Object Access Protocol (SOAP) 1.1, http://www.w3.org/TR/2000/NOTE-SOAP-20000508/
[4] Web Services Base Faults 1.2, http://docs.oasis-open.org/wsrf/wsrf-ws_base_faults-1.2-spec-os.pdf
[5] Sriram Krishnan, Patrick Wagstrom and Gregor von Laszewski (2002). GSFL: A Workflow Framework for Grid Services.
[6] Web Service Interoperability Organization (WS-I); http://www.ws-i.org/
[7] JAXB; http://java.sun.com/webservices/jaxb/
[8] Apache XML Beans; http://xmlbeans.apache.org/
[9] Apache Axis; http://ws.apache.org/axis/
[10] The e-HTPX project; http://www.e-htpx.ac.uk
[11] Altova XML Spy; http://www.altova.com/
[12] Netbeans IDE 5.5; http://www.netbeans.org/
[13] Cape Clear SOA Editor; http://www.capeclear.com
[14] JSR-224; http://jcp.org/aboutJava/communityprocess/final/jsr224/index.html
[15] Multipurpose Internet Mail Extensions (MIME), RFC 1521, http://www.faqs.org/ftp/rfc/rfc1521.pdf

[16] Direct Internet Message Encapsulation (DIME), http://msdn.microsoft.com/library/en-us/dnglobspec/html/draft-nielsen-dime-02.txt
[17] Attachments Profile 1.0 Second Edition, http://www.ws-i.org/Profiles/AttachmentsProfile-1.0.html