Web Services Description Language (WSDL) Version 2.0 Part 0: Primer

W3C Candidate Recommendation 6 January 2006

This version:
    http://www.w3.org/TR/2006/CR-wsdl20-primer-20060106
Latest version:
    http://www.w3.org/TR/wsdl20-primer
Previous versions:
    http://www.w3.org/TR/2005/WD-wsdl20-primer-20050803
Editors:
    David Booth, W3C Fellow / Hewlett-Packard
    Canyang Kevin Liu, SAP Labs

This document is also available in these non-normative formats: PDF, PostScript, XML, and plain text.

Abstract

This document is a companion to the WSDL 2.0 specification (*Web Services Description Language (WSDL) Version 2.0 Part 1: Core Language* [*WSDL 2.0 Core [p.83]*], *Web Services Description Language (WSDL) Version 2.0 Part 2: Adjuncts* [*WSDL 2.0 Adjuncts [p.83]*]). It is intended for readers who wish to have an easier, less technical introduction to the main features of the language.

This primer is only intended to be a starting point toward use of WSDL 2.0, and hence does not describe every feature of the language. Users are expected to consult the WSDL 2.0 specification if they wish to make use of more sophisticated features or techniques.

Finally, this primer is *non-normative*. Any specific questions of what WSDL 2.0 requires or forbids should be referred to the WSDL 2.0 specification.

Status of this Document

This section describes the status of this document at the time of its publication. Other documents may supersede this document. A list of current W3C publications and the latest revision of this technical report can be found in the W3C technical reports index at http://www.w3.org/TR/.

This is the W3C Candidate Recommendation of Web Services Description Language (WSDL) Version 2.0 Part 0: Primer for review by W3C Members and other interested parties. It has been produced by the Web Services Description Working Group, which is part of the W3C Web Services Activity. This specification will remain a Candidate Recommendation at least until 15 March 2006.

This Working Draft addresses all the comments received during the second Last Call review period on the WSDL 2.0 drafts. The detailed disposition of the comments received can be found in the Last Call issues list. A diff-marked version against the previous version of this document is available.

If the feedback is positive, the Working Group plans to submit this specification for consideration as a W3C Proposed Recommendation along with the rest of the WSDL 2.0 documents for which an implementation report is available.

Implementers are encouraged to provide feedback by 15 March 2006. Comments on this document are to be sent to the public public-ws-desc-comments@w3.org mailing list (public archive).

Issues about this document are recorded in the Candidate Recommendation issues list maintained by the Working Group. A list of formal objections against the set of WSDL 2.0 Working Drafts is also available.

Publication as a Candidate Recommendation does not imply endorsement by the W3C Membership. This is a draft document and may be updated, replaced or obsoleted by other documents at any time. It is inappropriate to cite this document as other than work in progress.

This document has been produced under the 24 January 2002 Current Patent Practice as amended by the W3C Patent Policy Transition Procedure. Patent disclosures relevant to this specification may be found on the Working Group's patent disclosure page. An individual who has actual knowledge of a patent which the individual believes contains Essential Claim(s) with respect to this specification should disclose the information in accordance with section 6 of the W3C Patent Policy.

Short Table of Contents

Table of Contents

Appendix

---

1. Introduction

1.1 Prerequisites

This primer assumes that the reader has the following prerequisite knowledge:

- familiarity with XML (*Extensible Markup Language (XML) 1.0 (Second Edition)* [*XML 1.0 [p.82]*], *XML Information Set* [*XML Information Set [p.82]*]) and XML Namespaces (*Namespaces in XML* [*XML Namespaces [p.82]*]);

- some familiarity with XML Schema (*XML Schema Part 1: Structures* [*XML Schema: Structures [p.82]*] *XML Schema Part 2: Datatypes* [*XML Schema: Datatypes [p.82]*]);

- familiarity with basic Web services concepts such as Web service, client, and the purpose and function of a Web service description. (For an explanation of basic Web services concepts, see *Web Services Architecture* [*WS Architecture [p.83]*] Section 1.4 and *Web Services Glossary* [*WS Glossary [p.83]*] glossary. However, note the *Web Services Architecture* document uses the slightly more precise terms "requester agent" and "provider agent" instead of the terms "client" and "Web service" used in this primer.)

No previous experience with WSDL is assumed.

1.2 Structure of this Primer

Section 2 starts with a hypothetical use case involving a hotel reservation service. It proceeds step-by-step through the development of a simple example WSDL 2.0 document that describes this service:

- The `types` element describes the kinds of messages that the service will send and receive.

- The `interface` element describes *what* abstract functionality the Web service provides.
- The `binding` element describes *how* to access the service.
- The `service` element describes *where* to access the service.

After presenting the example, it moves on to introduce the WSDL 2.0 infoset, schema, and component model. Then it provides more detailed coverage on defining message types, interfaces, bindings, and services.

Section 3 explains the WSDL 2.0 importing mechanisms in great details.

Section 4 talks about WSDL 2.0 extensibility and various predefined extensions.

Section 5 covers various topics that may fall outside the scope of WSDL 2.0, but shall provide useful background and best practice guidances that may be useful when authoring a WSDL 2.0 document or implementing the WSDL 2.0 specification.

1.3 Use of URI and IRI

The core specification of WSDL 2.0 supports Internationalized Resource Identifiers or IRIs [*IETF RFC 3987 [p.82]*]. IRIs are a superset of URIs with added support for internationalization. The URI syntax [*IETF RFC 3986 [p.82]*] only allows the use of a small set of characters, including upper and lower case letters of the English alphabet, European numerals and a few symbols. IRIs allow the use of characters from a wider range of language scripts.

For simplicity, examples throughout this primer only use URIs. If you are interested in learning more about the use of IRIs, you might care to read the paper prepared by the W3C Internationalization Activity.

1.4 Notational Conventions

This document uses several XML namespaces, some of which are defined by standards, and some are application-specific. Namespace names of the general form "http://greath.example.com/..." represent application or context-dependent URIs [*IETF RFC 3986 [p.82]*].Note also that the choice of any namespace prefix is arbitrary and not semantically significant (see [*XML Information Set [p.82]*]).

Following the convention for XML syntax summary in [*WSDL 2.0 Core [p.83]*], this primer uses an informal syntax to describe the XML grammar of a WSDL 2.0 document:

- The syntax appears as an XML instance, but the values indicate the data types instead of values.
- Characters are appended to elements and attributes as follows: "?" (0 or 1), "*" (0 or more), "+" (1 or more).
- Elements names ending in "…" indicate that elements/attributes irrelevant to the context are being omitted.

2. WSDL 2.0 Basics

2.1 Getting Started: The GreatH Hotel Example

This section introduces the basic concepts used in WSDL 2.0 through the description of a hypothetical hotel reservation service. We start with a simple scenario, and later add more requirements to illustrate how more advanced WSDL 2.0 features may be used.

2.1.1 Example Scenario: The GreatH Hotel Reservation Service

Hotel GreatH (a fictional hotel)) is located in a remote island. It has been relying on fax and phone to provide room reservations. Even though the facilities and prices at GreatH are better than what its competitor offers, GreatH notices that its competitor is getting more customers than GreatH. After research, GreatH realizes that this is because the competitor offers a Web service that permits travel agent reservation systems to reserve rooms directly over the Internet. GreatH then hires us to build a reservation Web service with the following functionality:

- *CheckAvailability*. To check availability, the client must specify a check-in date, a check-out date, and room type. The Web service will return a room rate (a floating point number in USD$) if such a room is available, or a zero room rate if not. If any input data is invalid, the service should return an error. Thus, the service will accept a `checkAvailability` message and return a `checkAvailabilityResponse` or `invalidDataFault` message.

- *MakeReservation*. To make a reservation, a client must provide a name, address, and credit card information, and the service will return a confirmation number if the reservation is successful. The service will return an error message if the credit card number or any other data field is invalid. Thus, the service will accept a `makeReservation` message and return a `makeReservationResponse` or `invalidCreditCardFault` message.

We know that we will later need to build a complete system that supports transactions and secured transmission, but initially we will implement only minimal functionality. In fact, to simplify our first example, we will implement only the *CheckAvailability* operation.

The next several sections proceed step-by-step through the process of developing a WSDL 2.0 document that describes the desired Web service. However, for those who can't wait to see a complete example, here is the WSDL 2.0 document that we'll be creating.

Example 2-1. WSDL 2.0 Document for the GreatH Web Service (Initial Example)

```
<?xml version="1.0" encoding="utf-8" ?>
<description
    xmlns="http://www.w3.org/2006/01/wsdl"
    targetNamespace= "http://greath.example.com/2004/wsdl/resSvc"
    xmlns:tns= "http://greath.example.com/2004/wsdl/resSvc"
    xmlns:ghns = "http://greath.example.com/2004/schemas/resSvc"
    xmlns:wsoap= "http://www.w3.org/2006/01/wsdl/soap"
    xmlns:soap="http://www.w3.org/2003/05/soap-envelope"
    xmlns:wsdlx= "http://www.w3.org/2006/01/wsdl-extensions">
```

```
<documentation>
  This document describes the GreatH Web service.  Additional
  application-level requirements for use of this service --
  beyond what WSDL 2.0 is able to describe -- are available
  at http://greath.example.com/2004/reservation-documentation.html
</documentation>

<types>
  <xs:schema
      xmlns:xs="http://www.w3.org/2001/XMLSchema"
      targetNamespace="http://greath.example.com/2004/schemas/resSvc"
      xmlns="http://greath.example.com/2004/schemas/resSvc">

    <xs:element name="checkAvailability" type="tCheckAvailability"/>
    <xs:complexType name="tCheckAvailability">
      <xs:sequence>
        <xs:element  name="checkInDate" type="xs:date"/>
        <xs:element  name="checkOutDate" type="xs:date"/>
        <xs:element  name="roomType" type="xs:string"/>
      </xs:sequence>
    </xs:complexType>

    <xs:element name="checkAvailabilityResponse" type="xs:double"/>

    <xs:element name="invalidDataError" type="xs:string"/>

  </xs:schema>
</types>

<interface  name = "reservationInterface" >

  <fault name = "invalidDataFault"
          element = "ghns:invalidDataError"/>

  <operation name="opCheckAvailability"
          pattern="http://www.w3.org/2006/01/wsdl/in-out"
          style="http://www.w3.org/2006/01/wsdl/style/iri"
          wsdlx:safe = "true">
      <input messageLabel="In"
            element="ghns:checkAvailability" />
      <output messageLabel="Out"
            element="ghns:checkAvailabilityResponse" />
      <outfault ref="tns:invalidDataFault" messageLabel="Out"/>
  </operation>

</interface>

<binding name="reservationSOAPBinding"
        interface="tns:reservationInterface"
        type="http://www.w3.org/2006/01/wsdl/soap"
        wsoap:protocol="http://www.w3.org/2003/05/soap/bindings/HTTP">

  <fault ref="tns:invalidDataFault"
    wsoap:code="soap:Sender"/>

  <operation ref="tns:opCheckAvailability"
    wsoap:mep="http://www.w3.org/2003/05/soap/mep/soap-response"/>
```

```
  </binding>

  <service name="reservationService"
       interface="tns:reservationInterface">

     <endpoint name="reservationEndpoint"
               binding="tns:reservationSOAPBinding"
               address ="http://greath.example.com/2004/reservation"/>

  </service>

</description>
```

2.1.2 Defining a WSDL 2.0 Target Namespace

Before writing our WSDL 2.0 document, we need to decide on a *WSDL 2.0 target namespace* URI for it. The WSDL 2.0 target namespace is analogous to an XML Schema target namespace. Interface, binding and service names that we define in our WSDL 2.0 document will be associated with the WSDL 2.0 target namespace, and thus will be distinguishable from similar names in a different WSDL 2.0 target namespace. (This will become important if using WSDL 2.0's import or interface inheritance mechanisms.)

The value of the WSDL 2.0 target namespace must be an absolute URI. Furthermore, it should be dereferenceable to a WSDL 2.0 document that describes the Web service that the WSDL 2.0 target namespace is used to describe. For example, the GreatH owners should make the WSDL 2.0 document available from this URI. (And if a WSDL 2.0 description is split into multiple documents, then the WSDL 2.0 target namespace should resolve to a master document that includes all the WSDL 2.0 documents needed for that service description.) However, there is no absolute requirement for this URI to be dereferenceable, so a WSDL 2.0 processor must not depend on it being dereferenceable.

This recommendation may sound circular, but bear in mind that the client might have obtained the WSDL 2.0 document from anywhere -- not necessarily an authoritative source. But by dereferencing the WSDL 2.0 target namespace URI, a user should be able to obtain an authoritative version. Since GreatH will be the owner of the service, the WSDL 2.0 target namespace URI should refer to a location on the GreatH Web site or otherwise within its control.

Once we have decided on a WSDL 2.0 target namespace URI, we can begin our WSDL 2.0 document as the following empty shell.

Example 2-2. An Initial Empty WSDL 2.0 Document

```
<?xml version="1.0" encoding="utf-8" ?>
<description
    xmlns="http://www.w3.org/2006/01/wsdl"
    targetNamespace= "http://greath.example.com/2004/wsdl/resSvc"
    xmlns:tns= "http://greath.example.com/2004/wsdl/resSvc"
    . . . >

  . . .
</description>
```

2.1.2.1 Explanation of Example

`<description`

Every WSDL 2.0 document has a `description` element as its top-most element. This merely acts as a container for the rest of the WSDL 2.0 document, and is used to declare namespaces that will be used throughout the document.

`xmlns="http://www.w3.org/2006/01/wsdl"`

This is the XML namespace for WSDL 2.0 itself. We assign it as the default namespace for this example by not defining a prefix for it. In other words, any unprefixed elements in this example are expected to be WSDL 2.0 elements (such as the `description` element).

`targetNamespace= "http://greath.example.com/2004/wsdl/resSvc"`

This defines the WSDL 2.0 target namespace that we have chosen for the GreatH reservation service, as described above. Note that this is not an actual XML namespace declaration. Rather, it is a WSDL 2.0 attribute whose purpose is *analogous* to an XML Schema target namespace.

`xmlns:tns= "http://greath.example.com/2004/wsdl/resSvc"`

This is an actual XML namespace declaration for use in our GreatH service description. Note that this is the same URI that was specified above as the value of the `targetNamespace` attribute. This will allow us later to use the `tns:` prefix in QNames, to refer to the WSDL 2.0 target namespace of the GreatH service. (For more on QNames see [*XML Namespaces [p.82]*] section 3 Qualified Names.)

Now we can start describing the GreatH service.

2.1.3 Defining Message Types

We know that the GreatH service will be sending and receiving messages, so a good starting point in describing the service is to define the message types that the service will use. We'll use XML Schema to do so, because WSDL 2.0 processors are likely to support XML Schema at a minimum. However, WSDL 2.0 does not prohibit the use of some other schema definition language.

WSDL 2.0 allows message types to be defined directly within the WSDL 2.0 document, inside the `types` element, which is a child of the `description` element. (Later we'll see how we can provide the type definitions in a separate document, using XML Schema's `import` mechanism.) The following schema defines `checkAvailability`, `checkAvailabilityResponse` and `invalidDataError` message types that we'll need.

In WSDL 2.0, all normal and fault message types must be defined as single *elements* at the topmost level (though of course each element may have any amount of substructure inside it). Thus, a message type must not directly consist of a sequence of elements or other complex type.

Example 2-3. GreatH Message Types

```
<?xml version="1.0" encoding="utf-8" ?>
<description
    xmlns="http://www.w3.org/2006/01/wsdl"
    targetNamespace= "http://greath.example.com/2004/wsdl/resSvc"
    xmlns:tns= "http://greath.example.com/2004/wsdl/resSvc"
    xmlns:ghns = "http://greath.example.com/2004/schemas/resSvc"
    . . . >

  ...

  <types>
    <xs:schema
        xmlns:xs="http://www.w3.org/2001/XMLSchema"
        targetNamespace="http://greath.example.com/2004/schemas/resSvc"
        xmlns="http://greath.example.com/2004/schemas/resSvc">

      <xs:element name="checkAvailability" type="tCheckAvailability"/>
      <xs:complexType name="tCheckAvailability">
        <xs:sequence>
          <xs:element  name="checkInDate" type="xs:date"/>
          <xs:element  name="checkOutDate" type="xs:date"/>
          <xs:element  name="roomType" type="xs:string"/>
        </xs:sequence>
      </xs:complexType>

      <xs:element name="checkAvailabilityResponse" type="xs:double"/>

      <xs:element name="invalidDataError" type="xs:string"/>

    </xs:schema>
  </types>
  . . .
</description>
```

2.1.3.1 Explanation of Example

`xmlns:ghns = "http://greath.example.com/2004/schemas/resSvc"`

We've added another namespace declaration. The `ghns` namespace prefix will allow us (later, when defining an interface) to reference the XML Schema target namespace that we define for our message types. Thus, the URI we specify must be the same as the URI that we define as the target namespace of our XML Schema types (below) -- *not* the target namespace of the WSDL 2.0 document itself.

`targetNamespace="http://greath.example.com/2004/schemas/resSvc"`

This is the XML Schema target namespace that we've created for use by the GreatH reservation service. The `checkAvailability`, `checkAvailabilityResponse` and `invalid-DataError` element names will be associated with this XML Schema target namespace.

`checkAvailability`, `checkAvailabilityResponse` and `invalidDataError`

These are the message types that we'll use. Note that these are defined to be XML *elements*, as explained above.

Although we have defined several types, we have not yet indicated which ones are to be used as message types for a Web service. We'll do that in the next section.

2.1.4 Defining an Interface

WSDL 2.0 enables one to separate the description of a Web service's abstract functionality from the concrete details of how and where that functionality is offered. This separation facilitates different levels of reusability and distribution of work in the lifecycle of a Web service and the WSDL 2.0 document that describes it.

A WSDL 2.0 `interface` defines the abstract interface of a Web service as a set of abstract *operations*, each operation representing a simple interaction between the client and the service. Each operation specifies the types of messages that the service can send or receive as part of that operation. Each operation also specifies a message exchange *pattern* that indicates the sequence in which the associated messages are to be transmitted between the parties. For example, the *in-out* pattern (see *WSDL 2.0 Predefined Extensions* [*WSDL 2.0 Adjuncts [p.83]*] section 2.2.3 In-Out) indicates that if the client sends a message *in* to the service, the service will either send a reply message back *out* to the client (in the normal case) or it will send a fault message back to the client (in the case of an error). We will explain more about message exchange *pattern*s in **2.4.4.3 Understanding Message Exchange Patterns (MEPs)** [p.33]

For the GreatH service, we will (initially) define an interface containing a single operation, `opCheckAvailability`, using the `checkAvailability` and `checkAvailabilityResponse` message types that we defined in the `types` section. We'll use the in-out pattern for this operation, because this is the most natural way to represent a simple request-response interaction. We could have instead (for example) defined two separate operations using the in-only and out-only patterns (see *WSDL 2.0 Predefined Extensions* [*WSDL 2.0 Adjuncts [p.83]*] section 2.2.1 In-Only and section 2.2.5 Out-Only), but that would just complicate matters for the client, because we would then have to separately indicate to the client developer that the two operations should be used together as a request-response pair.

In addition to the normal input and output messages, we also need to specify the fault message that we wish to use in the event of an error. WSDL 2.0 permits fault messages to be declared within the `interface` element in order to facilitate reuse of faults across operations. If a fault occurs, it terminates whatever message sequence was indicated by the message exchange pattern of the operation.

Let's add these to our WSDL 2.0 document.

Example 2-4. GreatH Interface Definition

```
<?xml version="1.0" encoding="utf-8" ?>
<description
    xmlns="http://www.w3.org/2006/01/wsdl"
    targetNamespace= "http://greath.example.com/2004/wsdl/resSvc"
    xmlns:tns= "http://greath.example.com/2004/wsdl/resSvc"
    xmlns:ghns = "http://greath.example.com/2004/schemas/resSvc"
    . . .
    xmlns:wsdlx="http://www.w3.org/2006/01/wsdl-extensions">
```

```
  . . .
  <types>
      ...
  </types>

  <interface  name = "reservationInterface" >

    <fault name = "invalidDataFault"
            element = "ghns:invalidDataError"/>

    <operation name="opCheckAvailability"
            pattern="http://www.w3.org/2006/01/wsdl/in-out"
            style="http://www.w3.org/2006/01/wsdl/style/iri"
            wsdlx:safe = "true">
        <input messageLabel="In"
              element="ghns:checkAvailability" />
        <output messageLabel="Out"
              element="ghns:checkAvailabilityResponse" />
        <outfault ref="tns:invalidDataFault" messageLabel="Out"/>
    </operation>

  </interface>

  . . .
</description>
```

2.1.4.1 Explanation of Example

`<interface name = "reservationInterface" >`

Interfaces are declared directly inside the `description` element. In this example, we are declaring only one interface, but in general a WSDL 2.0 document may declare more than one interface. Thus, each interface must be given a name that is unique within the set of interfaces defined in this WSDL 2.0 target namespace. Interface names are tokens that must not contain a space or colon (":").

`<fault name = "invalidDataFault"`

The `name` attribute defines a name for this fault. The name is required so that when an operation is defined, it can reference the desired fault by name. Fault names must be unique within an interface.

`element = "ghns:invalidDataError"/>`

The `element` attribute specifies the schema type of the fault message, as previously defined in the `types` section.

`<operation name="opCheckAvailability"`

The `name` attribute defines a name for this operation, so that it can be referenced later when bindings are defined. Operation names must also be unique within an interface. (WSDL 2.0 uses separate symbol spaces for operation and fault names, so operation name "foo" is distinct from fault name "foo".)

`pattern="http://www.w3.org/2006/01/wsdl/in-out"`

This line specifies that this operation will use the in-out pattern as described above. WSDL 2.0 uses URIs to identify message exchange patterns in order to ensure that the identifiers are globally unambiguous, while also permitting future new patterns to be defined by anyone. (However, just because someone defines a new pattern and creates a URI to identify it, that does *not* mean that other WSDL 2.0 processors will automatically recognize or understand that pattern. As with any other extension, it can only be used among processors that *do* recognize and understand it.)

`style="http://www.w3.org/2006/01/wsdl/style/iri"`

This line indicates that the XML schema defining the input message of this operation follows a set of rules as specified in IRI Style that ensures the message can be serialized as an IRI.

`wsdlx:safe="true" >`

This line indicates that this operation will not obligate the client in any way, i.e., the client can safely invoke this operation without fear that it may be incurring an obligation (such as agreeing to buy something). This is further explained in **2.4.4 Interface Operations** [p.31] .

`<input messageLabel="In"`

The `input` element specifies an input message. Even though we have already specified which message exchange pattern the operation will use, a message exchange pattern represents a template for a message sequence, and in theory could consist of multiple input and/or output messages. Thus we must also indicate which potential input message in the pattern this particular input message represents. This is the purpose of the `messageLabel` attribute. Since the in-out pattern that we've chosen to use only has one input message, it is trivial in this case: we simply fill in the message label "In" that was defined in *WSDL 2.0 Predefined Extensions* [*WSDL 2.0 Adjuncts [p.83]*] section 2.2.3 In-Out for the in-out pattern. However, if a new pattern is defined that involve multiple input messages, then the different input messages in the pattern could then be distinguished by using different labels.

`element="ghns:checkAvailability" />`

This specifies the message type for this input message, as defined previously in the `types` section.

`<output messageLabel="Out" . . .`

This is similar to defining an input message.

`<outfault ref="tns:invalidDataFault" messageLabel="Out"/>`

This associates an output fault with this operation. Faults are declared a little differently than normal messages. The `ref` attribute refers to the name of a previously defined fault in this interface -- not a message schema type directly. Since message exchange patterns could in general involve a sequence of several messages, a fault could potentially occur at various points within the message sequence. Because one may wish to associate a different fault with each permitted point in the sequence, the `messageLabel` is used to indicate the desired point for this particular fault. It does so indirectly by specifying the message that will either trigger this fault or that this fault will replace, depending on the pattern. (Some patterns use a message-triggers-fault rule; others use a fault-replaces-message rule. See *WSDL 2.0 Predefined Extensions* [*WSDL 2.0 Adjuncts [p.83]*] section 2.1.2 Message Triggers Fault and section 2.1.1 Fault Replaces Message.)

Now that we've defined the abstract interface for the GreatH service, we're ready to define a binding for it.

2.1.5 Defining a Binding

Although we have specified *what* abstract messages can be exchanged with the GreatH Web service, we have not yet specified *how* those messages can be exchanged. This is the purpose of a *binding*. A binding specifies concrete message format and transmission protocol details for an interface, and must supply such details for every operation and fault in the interface.

In the general case, binding details for each operation and fault are specified using `operation` and `fault` elements inside a `binding` element, as shown in the example below. However, in some cases it is possible to use defaulting rules to supply the information. The WSDL 2.0 SOAP binding extension, for example, defines some defaulting rules for operations. (See *Web Services Description Language (WSDL) Version 2.0 Part 2: Adjuncts* [*WSDL 2.0 Adjuncts [p.83]*], Default Binding Rules.)

In order to accommodate new kinds of message formats and transmission protocols, bindings are defined using extensions to the WSDL 2.0 language, via WSDL 2.0's open content model. (See **4.1 Extensibility** [p.50] for more on extensibility.) WSDL 2.0 Part 2 [*WSDL 2.0 Adjuncts [p.83]*] defines binding extensions for SOAP 1.2 [*SOAP 1.2 Part 1: Messaging Framework [p.84]*] and HTTP 1.1 [*IETF RFC 2616 [p.84]*] as predefined extensions, so that SOAP 1.2 or HTTP 1.1 bindings can be easily defined in WSDL 2.0 documents. However, other specifications could define new binding extensions that could also be used to define bindings. (As with any extension, other WSDL 2.0 processors would have to know about the new constructs in order to make use of them.)

For the GreatH service, we will use SOAP 1.2 as our concrete message format and HTTP as our underlying transmission protocol, as shown below.

Example 2-5. GreatH Binding Definition

```
<?xml version="1.0" encoding="utf-8" ?>
<description
    xmlns="http://www.w3.org/2006/01/wsdl"
    targetNamespace= "http://greath.example.com/2004/wsdl/resSvc"
    xmlns:tns= "http://greath.example.com/2004/wsdl/resSvc"
    xmlns:ghns = "http://greath.example.com/2004/schemas/resSvc"
    xmlns:wsoap= "http://www.w3.org/2006/01/wsdl/soap"
    xmlns:soap="http://www.w3.org/2003/05/soap-envelope">
  . . .

  <types>
    . . .
  </types>

  <interface  name = "reservationInterface" >
        ...
```

```
  </interface>

  <binding name="reservationSOAPBinding"
          interface="tns:reservationInterface"
          type="http://www.w3.org/2006/01/wsdl/soap"
          wsoap:protocol="http://www.w3.org/2003/05/soap/bindings/HTTP">

    <operation ref="tns:opCheckAvailability"
      wsoap:mep="http://www.w3.org/2003/05/soap/mep/soap-response"/>

    <fault ref="tns:invalidDataFault"
      wsoap:code="soap:Sender"/>

  </binding>

  . . .
</description>
```

2.1.5.1 Explanation of Example

`xmlns:wsoap= "http://www.w3.org/2006/01/wsdl/soap"`

We've added two more namespace declarations. This one is the namespace for the SOAP 1.2 binding extension that is defined in WSDL 2.0 Part 3 [*SOAP 1.2 Part 1: Messaging Framework [p.84]*]. Elements and attributes prefixed with `wsoap:` are constructs defined there.

`xmlns:soap="http://www.w3.org/2003/05/soap-envelope"`

This namespace is defined by the SOAP 1.2 specification itself. The SOAP 1.2 specification defines certain terms within this namespace to unambiguously identify particular concepts. Thus, we will use the `soap:` prefix when we need to refer to one of those terms.

`<binding name="reservationSOAPBinding"`

Bindings are declared directly inside the `description` element. The `name` attribute defines a name for this binding. Each name must be unique among all bindings in this WSDL 2.0 target namespace, and will be used later when we define a service endpoint that references this binding. WSDL 2.0 uses separate symbol spaces for interfaces, bindings and services, so interface "foo", binding "foo" and service "foo" are all distinct.

`interface="tns:reservationInterface"`

This is the name of the interface whose message format and transmission protocols we are specifying. As discussed in **2.5 More on Bindings** [p.35] , a reusable binding can be defined by omitting the `interface` attribute. Note also the use of the `tns:` prefix, which refers to the previously defined WSDL 2.0 target namespace for this WSDL 2.0 document. In this case it may seem silly to have to specify the `tns:` prefix, but in **3.1 Importing WSDL** [p.42] we will see how WSDL 2.0's import mechanism can be used to combine components that are defined in different WSDL 2.0 target namespaces.

`type="http://www.w3.org/2006/01/wsdl/soap"`

This specifies what kind of concrete message format to use, in this case SOAP 1.2.

`wsoap:protocol="http://www.w3.org/2003/05/soap/bindings/HTTP"`

This attribute is specific to WSDL 2.0's SOAP binding extension (thus it uses the `wsoap:` prefix). It specifies the underlying transmission protocol that should be used, in this case HTTP.

`<operation ref="tns:opCheckAvailability"`

This is not defining a new operation; rather, it is referencing the previously defined `opCheck-Availability` operation in order to specify binding details for it. This element can be omitted if defaulting rules are instead used to supply the necessary information. (See the SOAP binding extension in WSDL 2.0 Part 2 [*WSDL 2.0 Adjuncts [p.83]*] section 4.3 Default Binding Rules .)

`wsoap:mep="http://www.w3.org/2003/05/soap/mep/soap-response">`

This attribute is also specific to WSDL 2.0's SOAP binding extension. It specifies the SOAP message exchange pattern (MEP) that will be used to implement the abstract WSDL 2.0 message exchange pattern (in-out) that was specified when the `opCheckAvailability` operation was defined.

When HTTP is used as the underlying transport protocol (as in this example) the `wsoap:mep` attribute also controls whether GET or POST will be used as the underlying HTTP method. In this case, the use of `wsoap:mep="http://www.w3.org/2003/05/soap/mep/soap-response"` causes GET to be used by default. See also **2.5.7 HTTP GET Versus POST: Which to Use?** [p.41] .

`<fault ref="tns:invalidDataFault"`

As with a binding operation, this is not declaring a new fault; rather, it is referencing a fault (`invalidDataFault`) that was previously defined in the `opCheckAvailability` interface, in order to specify binding details for it.

`wsoap:code="soap:Sender"/>`

This attribute is also specific to WSDL 2.0's SOAP binding extension. This specifies the SOAP 1.2 fault code that will cause this fault message to be sent. If desired, a list of subcodes can also be specified using the optional `wsoap:subcodes` attribute.

2.1.6 Defining a Service

Now that our binding has specified *how* messages will be transmitted, we are ready to specify *where* the service can be accessed, by use of the `service` element.

A WSDL 2.0 *service* specifies a single interface that the service will support, and a list of *endpoint* locations where that service can be accessed. Each endpoint must also reference a previously defined binding to indicate what protocols and transmission formats are to be used at that endpoint. A service is only permitted to have one interface. (See **5.4 Multiple Interfaces for the Same Service** [p.79] for further discussion of this limitation.)

Here is a definition for our GreatH service.

Example 2-6. GreatH Service Definition

```
<?xml version="1.0" encoding="utf-8" ?>
<description
    xmlns="http://www.w3.org/2006/01/wsdl"
    targetNamespace= "http://greath.example.com/2004/wsdl/resSvc"
    xmlns:tns= "http://greath.example.com/2004/wsdl/resSvc"
    xmlns:ghns = "http://greath.example.com/2004/schemas/resSvc"
    xmlns:wsoap= "http://www.w3.org/2006/01/wsdl/soap"
    xmlns:soap="http://www.w3.org/2003/05/soap-envelope">
  . . .

  <types>
    . . .
  </types>

  <interface  name = "reservationInterface" >
    . . .
  </interface>

  <binding name="reservationSOAPBinding"
          interface="tns:reservationInterface"
        . . . >
    . . .
  </binding>

  <service name="reservationService"
       interface="tns:reservationInterface">

     <endpoint name="reservationEndpoint"
               binding="tns:reservationSOAPBinding"
               address ="http://greath.example.com/2004/reservation"/>

  </service>

</description>
```

2.1.6.1 Explanation of Example

`<service name="reservationService"`

> This defines a name for this service, which must be unique among service names in the WSDL 2.0 target namespace. The name attribute is required. It allows URIs to be created that identify components in WSDL 2.0 description. (See *WSDL 2.0 Core Language* [*WSDL 2.0 Core [p.83]*] appendix C URI References for WSDL 2.0 constructs.)

`interface="tns:reservationInterface">`

This specifies the name of the previously defined interface that these service endpoints will support.

```
<endpoint name="reservationEndpoint"
```

This defines an endpoint for the service, and a name for this endpoint, which must be unique within this service.

```
binding="tns:reservationSOAPBinding"
```

This specifies the name of the previously defined binding to be used by this endpoint.

```
address ="http://greath.example.com/2004/reservation"/>
```

This specifies the physical address at which this service can be accessed using the binding specified by the `binding` attribute.

That's it! Well, almost.

2.1.7 Documenting the Service

As we have seen, a WSDL 2.0 document is inherently only a *partial* description of a service. Although it captures the basic mechanics of interacting with the service -- the message types, transmission protocols, service location, etc. -- in general, additional documentation will need to explain other application-level requirements for its use. For example, such documentation should explain the purpose and use of the service, the meanings of all messages, constraints on their use, and the sequence in which operations should be invoked.

The `documentation` element allows the WSDL 2.0 author to include some human-readable documentation inside a WSDL 2.0 document. It is also a convenient place to reference any additional external documentation that a client developer may need in order to use the service. It can appear in a number of places in a WSDL 2.0 document (see **2.2.1 WSDL 2.0 Infoset** [p.20]), though in this example we have only demonstrated its use at the beginning.

Example 2-7. Documenting the GreatH Service

```
<?xml version="1.0" encoding="utf-8" ?>
<description
    . . . >

  <documentation>
    This document describes the GreatH Web service.  Additional
    application-level requirements for use of this service --
    beyond what WSDL 2.0 is able to describe -- are available
    at http://greath.example.com/2004/reservation-documentation.html
  </documentation>
  . . .
</description>
```

2.1.7.1 Explanation of Example

`<documentation>`

> This element is optional, but a good idea to include. It can contain arbitrary mixed content.

`at http://greath.example.com/2004/reservation-documentation.html`

> The most important thing to include is a pointer to any additional documentation that a client developer would need in order to use the service.

This completes our presentation of the GreatH example. In the following sections, we will move on to look into more details of various aspects of WSDL 2.0 specification.

2.2 WSDL 2.0 Infoset, Schema and Component Model

In computer science theory, a language consists of a (possibly infinite) set of sentences, and each sentence is a finite string of literal symbols or characters. A language specification must therefore define the set sentences in that language, and, to be useful, it should also indicate the meaning of each sentence. Indeed, this is the purpose of the WSDL 2.0 specification.

However, instead of defining WSDL 2.0 in terms of literal symbols or characters, to avoid dependency on any particular character encoding, WSDL 2.0 is defined in terms of the *XML Infoset* [*XML Information Set [p.82]*]. Specifically, a *WSDL 2.0 document* consists of a `description` element information item (in the XML Infoset) that conforms to the WSDL 2.0 specification. In other words, a sentence in the WSDL 2.0 language is a `description` element information item that obeys the additional constraints spelled out in the WSDL 2.0 specification.

Since an XML Infoset can be created from more than one physical document, a WSDL 2.0 document does not necessarily correspond to a single *physical* document: the word "document" is used figuratively, for convenience. Furthermore, since WSDL 2.0 provides `import` and `include` mechanisms, a WSDL 2.0 document may reference other WSDL 2.0 documents to facilitate convenient organization or reuse. In such cases, the meaning of the including or importing document as a whole will depend (in part) on the meaning of the included or imported document.

The XML Infoset uses terms like "element information item" and "attribute information item". Unfortunately, those terms are rather lengthy to repeat often. Thus, for convenience, this primer often uses the terms "element" and "attribute" instead, as a shorthand. It should be understood, however, that since WSDL 2.0 is based on the XML Infoset, we really mean "element information item" and "attribute information item", respectively.

2.2.1 WSDL 2.0 Infoset

The following diagram gives an overview of the XML Infoset for a WSDL 2.0 document.

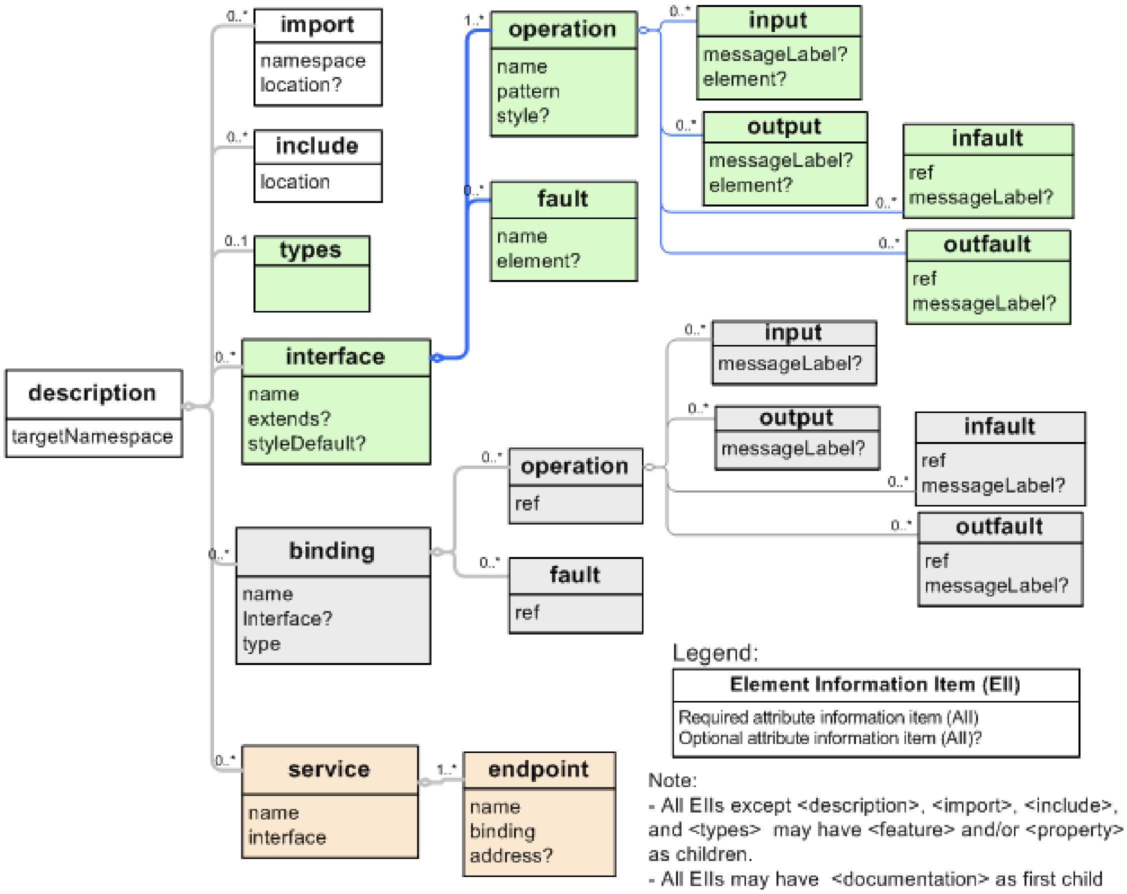

Figure 2-1. WSDL 2.0 Infoset Diagram

2.2.2 WSDL 2.0 Schema

The WSDL 2.0 specification supplies a normative WSDL 2.0 schema, defined in [*XML Schema: Structures [p.82]*], which can be used as an aid in validating WSDL 2.0 documents. We say "as an aid" here because WSDL 2.0 specification [*WSDL 2.0 Core [p.83]*] often provides further constraints to the WSDL 2.0 schema. In addition to being valid with the normative schema, a WSDL 2.0 document must also follow all the constraints defined by the WSDL 2.0 specification.

2.2.2.1 WSDL 2.0 Element Ordering

This section gives an example of how WSDL 2.0 specification constrains the WSDL 2.0 schema about the ordering of top WSDL 2.0 elements.

Although the WSDL 2.0 schema does not indicate the required ordering of elements, the WSDL 2.0 specification (WSDL 2.0 Part 1 [*WSDL 2.0 Core [p.83]*] section "XML Representation of Description Component") clearly states a set of constraints about how the children elements of the `description` element should be ordered. Thus, the order of the WSDL 2.0 elements matters, in spite of what the WSDL 2.0 schema says.

The following is a pseudo-content model of `description`.

```
<description>
  <documentation />?
  [ <import /> | <include /> ]*
  <types />?
  [ <interface /> | <binding /> | <service /> ]*
</description>
```

In other words, the children elements of the `description` element should be ordered as follows:

- An optional `documentation` comes first, if present.
- then comes zero or more elements from among the following, in any order:
  - `include`
  - `import`
  - extensions
- An optional `types` follows
- Zero or more elements from among the following, in any order:
  - `interface`
  - `binding`
  - `service`
  - extensions.

Note the term "extension" is used above as a convenient way to refer to namespace-qualified extension elements. The namespace name of such extension elements must not be"http://www.w3.org/2006/01/wsdl".

2.2.3 WSDL 2.0 Component Model

The WSDL 2.0 Infoset model above illustrates the required structure of a WSDL 2.0 document, using the XML Infoset. However, the WSDL 2.0 language also imposes many semantic constraints over and above structural conformance to this XML Infoset. In order to precisely describe these constraints, and as an aid in precisely defining the meaning of each WSDL 2.0 document, the WSDL 2.0 specification defines a *component model* as an additional layer of abstraction above the XML Infoset. Constraints and meaning are defined in terms of this component model, and the definition of each component includes a mapping that specifies how values in the component model are derived from corresponding items in the XML Infoset. The following diagram gives an overview of the WSDL 2.0 components and their containment hierarchy.

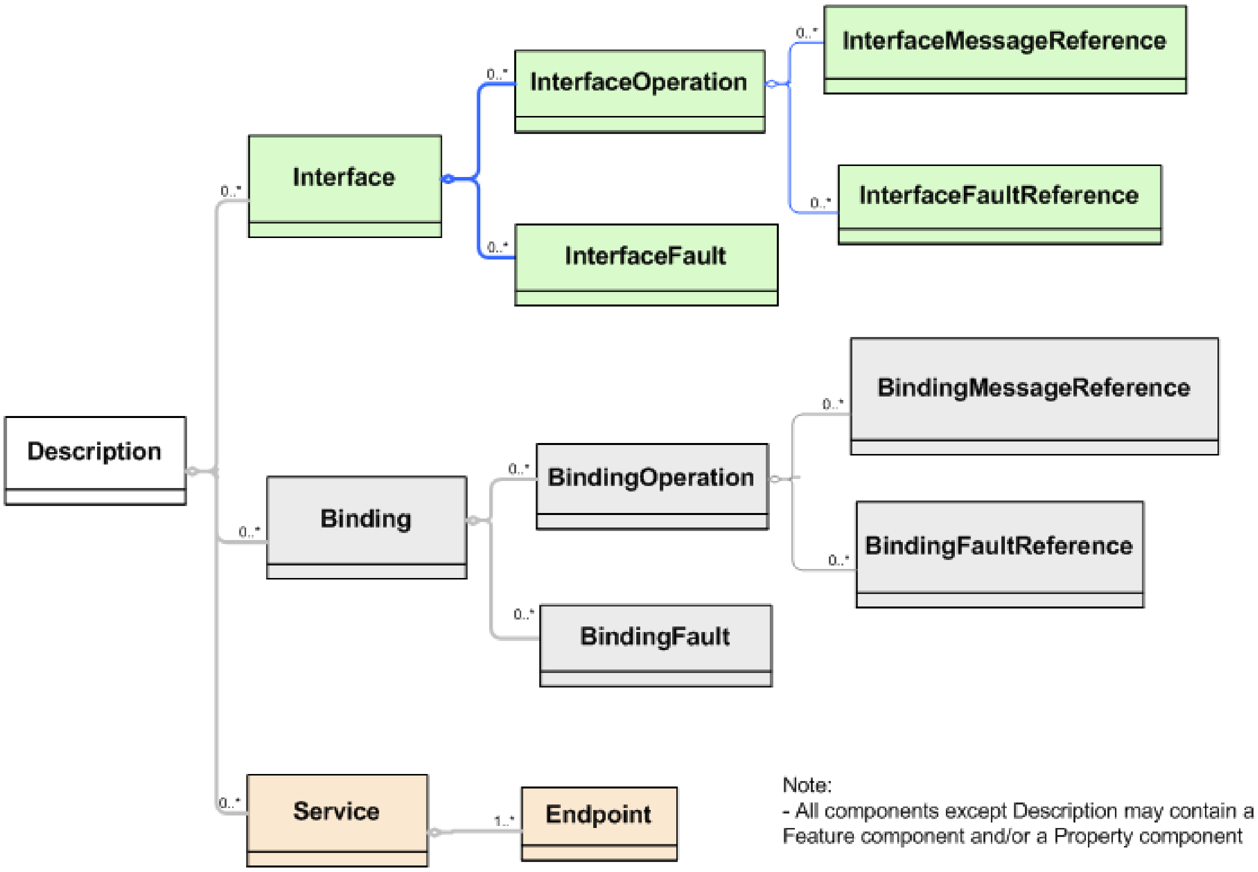

Figure 2-2. WSDL 2.0 Components Containment hierarchy

In general, the WSDL 2.0 component model parallels the structure of the required XML Infoset illustrated above. For example, the *Description*, *Interface*, *Binding*, *Service* and *Endpoint components* correspond to the `description`, `interface`, `binding`, `service`, and `endpoint` element information items, respectively. Since WSDL 2.0 relies heavily on the component model to convey the meaning of the constructs in the WSDL 2.0 language, you can think of the Description component as representing the meaning of the `description` element information item, and hence, it represents the meaning of the WSDL 2.0 document as a whole.

Furthermore, each of these components has *properties* whose values are (usually) derived from the element and attribute information item children of those element information items. For example, the Service component corresponds to the `service` element information item, so the Service component has an {endpoints} property whose value is a set of Endpoint components corresponding to the `endpoint` element information item children of that `service` element information item. (Whew!).

2.2.3.1 WSDL 2.0 Import and Include

The WSDL 2.0 component model is particularly helpful in defining the meaning of `import` and `include` elements. The `include` element allows you to assemble the contents of a given WSDL 2.0 namespace from several WSDL 2.0 documents that define components for that namespace. The components defined by a given WSDL 2.0 document consist of those whose definitions are contained in the document and those that are defined by any WSDL 2.0 documents that are included in it via the `include` element. The effect of the `include` element is cumulative so that if document A includes document B and document B includes document C, then the components defined by document A consist of those whose definitions are contained in documents A, B, and C.

In contrast, the `import` element does not define any components. Instead, the `import` element declares that the components whose definitions are contained in a WSDL 2.0 document for a given WSDL 2.0 namespace refer to components that belong to a different WSDL 2.0 namespace. If a WSDL 2.0 document contains definitions of components that refer to other namespaces, then those namespaces must be declared via an `import` element. The `import` element also has an optional `location` attribute that is a hint to the processor where the definitions of the imported namespace can be found. However, the processor may find the definitions by other means, for example, by using a catalog.

After processing any `include` elements and locating the components that belong to any imported namespaces, the WSDL 2.0 component model for a WSDL 2.0 document will contain a set of components that belong to the document's WSDL 2.0 namespace and any imported namespaces. These components will refer to each other, usually via QName references. A WSDL 2.0 document is invalid if any component reference cannot be resolved, whether or not the referenced component belongs to the same or a different namespace.

We will cover a lot more about how to use WSDL 2.0 import and include in **3.1 Importing WSDL** [p.42]

2.3 More on Message Types

Message types may be defined in various schema languages. In this primer, we will only focus on the use of XML Schema [*XML Schema: Structures [p.82]*] since it's natively supported by WSDL 2.0. Message types defined in other languages may be introduced into a WSDL 2.0 `description` via extensions, see the W3C notes [*Alternative Schema Languages Support [p.85]*] for more details.

The following is the XML syntax for the `wsdl:types` element:

```
<description>
  <types>
    <documentation />*
    [ <xs:import namespace="xs:anyURI" schemaLocation="xs:anyURI"? /> |
      <xs:schema targetNamespace="xs:anyURI" /> |
      other extension elements ]*
  </types>
</description>
```

There are two ways to make XML Schema message definitions visible, or in other words, available for reference by QName (see WSDL 2.0 Part 1 [*WSDL 2.0 Core [p.83]*] "QName Resolution") in a WSDL 2.0 document: inlining or importing. Inlining is to put the schema definitions directly within an `xs:schema` element under `types`. Importing is to have the schema defined in a separate document and then bring it into the WSDL definition by using `xs:import` directly under `types`.

In the following sections, we will provide examples for the different mechanisms.

2.3.1 Inlining XML Schema

We have already seen an example of using inlined schema definitions in section **2.1.3 Defining Message Types** [p.10] . When XML Schema is inlined directly in a WSDL 2.0 document, it uses the existing top-level `xs:schema` element defined by XML Schema to do so, as though a schema file had been copied and pasted into the `types` element. The schema components defined in the inlined schema are then available to the containing WSDL 2.0 `description` for reference by QName. For instance, in Example 2-1 [p.7] , the input message of the interface operation "opCheckAvailability" is defined by the "ghns:checkAvailability" element in the inlined schema.

2.3.2 Importing XML Schema

XML Schema components can be defined in separate schema files and be made available to a WSDL2.0 `description` by using `xs:import` directly under `types`.

There are many cases where one would prefer having schema definitions in separate schema files. One reason is the reusability of the schema definitions. Inlined schema definitions are only available to the containing WSDL 2.0 `description`. Although WSDL 2.0 provides a `wsdl:import` mechanism for importing other WSDL files, schema definitions inlined in an imported WSDL document are NOT automatically made available to the importing WSDL 2.0 document, even though other WSDL 2.0 components (such as Interfaces, Bindings, etc.) do become available. Therefore, if one wishes to share schema definitions across several WSDL 2.0 `descriptions`, these schema definitions should instead be placed in separate XML Schema documents and imported into each WSDL 2.0 `description` using `xs:import` directly under `types`.

Let's see an example. Assuming the message types in Example 2-3 [p.11] are defined in a separate schema file named "http://greath.example.com/2004/schemas/resSvc.xsd" with a target namespace "http://greath.example.com/2004/schemas/resSvc", the schema definition can then be brought into the WSDL 2.0 `description` using `xs:import`. Note that only components in the imported namespace "http://greath.example.com/2004/schemas/resSvc" are available for reference in the WSDL 2.0 document.

Example 2-8. `xs:imported` Message Definitions that Are Visible to the Containing WSDL 2.0 Description

```
<?xml version="1.0" encoding="utf-8" ?>
<description xmlns="http://www.w3.org/2006/01/wsdl"
targetNamespace= "http://greath.example.com/2004/wsdl/resSvc"
xmlns:tns= "http://greath.example.com/2004/wsdl/resSvc"
xmlns:ghns = "http://greath.example.com/2004/schemas/resSvc"
. . . >
. . .

<types>
        <xs:import namespace="http://greath.example.com/2004/schemas/resSvc"
               schemaLocation= "http://greath.example.com/2004/schemas/resSvc.xsd"/>
```

```
</types>

. . .
</description>
```

It's important to note that `xs:import` used directly under `wsdl:types` has been given a different visibility than `xs:import` used inside an inlined schema. An inlined schema may use native XML schema `xs:import` to bring in external schema definitions that are in different namespaces; However, though this is the schema importing mechanism recommended for WSDL 1.1 in WS-I Basic Profile, according to XML Schema specification, such enclosed message definitions are only visible to the importing schema (in this case, the inlined schema). They are not visible to the containing WSDL 2.0 `description`.

If we change Example 2-8 [p.25] to use XML Schema's native `xs:import` element in an inlined schema, the schema components defined in the namespace http://greath.example.com/2004/schemas/resSvc are not available to our example WSDL 2.0 definition any more.

Example 2-9. `xs:imported` Message Definitions in Inlined Schema Are Not Visible to the Containing WSDL 2.0 Description

```
<?xml version="1.0" encoding="utf-8" ?>
<description xmlns="http://www.w3.org/2006/01/wsdl"
targetNamespace= "http://greath.example.com/2004/wsdl/resSvc"
xmlns:tns= "http://greath.example.com/2004/wsdl/resSvc"
xmlns:ghns = "http://greath.example.com/2004/schemas/resSvc"
. . . >
. . .

<types>
        <xs:schema targetNamespace="http://greath.example.com/2004/schemas/resSvcWrapper">
                <xs:import namespace="http://greath.example.com/2004/schemas/resSvc"
                schemaLocation= "http://greath.example.com/2004/schemas/resSvc.xsd"/>
        </xs:schema>
</types>

. . .
</description>
```

Of course, an inlined XML schema may also use XML Schema's native `xs:include` element to refer to schemas defined in separate files when the included schema has no namespace or has the same namespace as the including schema. In this case, according to XML Schema, the included schema components become a part of the including schema as though they had been copied and pasted into the including schema. Hence, the included schema components are also available to the containing WSDL 2.0 `description` for reference by QName.

The following example has the same effect as Example 2-3 [p.11] :

Example 2-10. `xs:included` Message Definitions in Inlined Schema Are Visible to the Containing WSDL 2.0 Description

```
<?xml version="1.0" encoding="utf-8" ?>
<description xmlns="http://www.w3.org/2006/01/wsdl"
targetNamespace= "http://greath.example.com/2004/wsdl/resSvc"
xmlns:tns= "http://greath.example.com/2004/wsdl/resSvc"
xmlns:ghns = "http://greath.example.com/2004/schemas/resSvc"
. . . >
. . .

<types>
        <xs:schema targetNamespace="http://greath.example.com/2004/schemas/resSvc">
                <xs:include schemaLocation= "http://greath.example.com/2004/schemas/resSvc.xsd"/>
        </xs:schema>
</types>

. . .
</description>
```

2.3.3 Summary of Import and Include Mechanisms

So far we have briefly covered both WSDL import/include and schema import/include. The following table summarizes the similarities and differences between the WSDL 2.0 and XML Schema `include` and `import` mechanisms. We will talk a lot more about importing mechanisms in **3.1 Importing WSDL** [p.42] and **3.2 Importing Schemas** [p.45]

Table 2-1. Summary of Import and Include Mechanisms

Mechanism	Object	Meaning	Visibility of Schema Components
wsdl:import	WSDL 2.0 Namespace	Declare that WSDL 2.0 components refer to WSDL 2.0 components from a DIFFERENT targetNamespace.	XML Schema Components in the imported Description component are NOT visible to the containing `description`.
wsdl:include	WSDL 2.0 Document	Merge Interface, Binding and Service components from another WSDL 2.0 document that has the SAME targetNamespace.	XML Schema components in the included Description component's {element declarations} and {type definitions} properties are visible to the containing `description`.
wsdl:types/ xs:import	XML Schema Namespace	Declare that XML Schema components refer to XML Schema components from a DIFFERENT targetNamespace.	XML Schema components in the imported namespace are visible to the containing `description`.
wsdl:types/ xs:schema/xs:import	XML Schema Namespace	Declare that XML Schema components refer to XML Schema components from a DIFFERENT targetNamespace.	XML Schema components in the imported namespace are NOT visible to the containing `description`.
wsdl:types/ xs:schema/xs:include	XML Schema Document	Merge XML Schema components from another XML Schema document that has the SAME or NO targetNamespace.	XML Schema components in the included document are visible to the containing `description`.

2.4 More on Interfaces

We previously mentioned that a WSDL 2.0 interface is basically a set of operations. However, there are some additional capabilities that we have not yet covered. First, let's review the syntax for the `interface` element.

2.4.1 Interface Syntax

Below is the XML syntax summary of the `interface` element, simplified by omitting optional `<documentation>` elements and `<feature>` and `<property>` extension elements:

```
<description targetNamespace="xs:anyURI" >

  . . .
  <interface name="xs:NCName"
          extends="list of xs:QName"?
          styleDefault="list of xs:anyURI"? >

    <fault name="xs:NCName"
            element="xs:QName"? >
    </fault>*

    <operation name="xs:NCName"
            pattern="xs:anyURI"
            style="list of xs:anyURI"?
            wsdlx:safe="xs:boolean"? >

      <input messageLabel="xs:NCName"?
            element="union of xs:QName, xs:Token"? >
      </input>*

      <output messageLabel="xs:NCName"?
            element="union of xs:QName, xs:Token"? >
      </output>*

      <infault ref="xs:QName" messageLabel="xs:NCName"? > </infault>*

      <outfault ref="xs:QName" messageLabel="xs:NCName"? > </outfault>*

    </operation>*

  </interface>*
  . . .

</description>
```

The `interface` element has two optional attributes: `styleDefault` and `extends` . The `styleDefault` attribute can be used to define a default value for the `style` attributes of all operations under this interface (see WSDL 2.0 Part 1 "styleDefault attribute information item"). The `extends` attribute is for inheritance, and is explained next.

2.4.2 Interface Inheritance

The optional `extends` attribute allows an interface to extend or inherit from one or more other interfaces. In such cases the interface contains the operations of the interfaces it extends, along with any operations it defines directly. Two things about extending interfaces deserve some attention.

First, an inheritance loop (or infinite recursion) is prohibited: the interfaces that a given interface extends must NOT themselves extend that interface either directly or indirectly.

Second, we must explain what happens when operations from two different interfaces have the same target namespace and operation name. There are two cases: either the component models of the operations are the same, or they are different. If the component models are the same (per the component comparison algorithm defined in WSDL 2.0 Part 1 [*WSDL 2.0 Core [p.83]*] " Equivalence of Components ") then they are considered to be the same operation, i.e., they are collapsed into a single operation, and the fact that they were included more than once is not considered an error. (For operations, component equivalence basically means that the two operations have the same set of attributes and descendants.) In the second case, if two operations have the same name in the same WSDL 2.0 target namespace but are not equivalent, then it is an error. For the above reason, it is considered good practice to ensure that all operations within the same target namespace are named uniquely.

Finally, since faults can also be defined as children of the `interface` element (as described in the following sections), the same name-collision rules apply to those constructs.

Let’s say the GreatH hotel wants to maintain a standard message log operation for all received messages. It wants this operation to be reusable across the whole reservation system, so each service will send out, for potential use of a logging service, the content of each message it receives together with a timestamp and the originator of the message. One way to meet such requirement is to define the log operation in an interface which can be inherited by other interfaces. Assuming a `messageLog` element is already defined in the ghns namespace with the required content, the inheritance use case is illustrated in the following example. As a result of the inheritance, the `reservationInterface` now contains two operations: `opCheckAvailability` and `opLogMessage`

Example 2-11. Interface Inheritance

```
<description ...>
        ...
        <interface  name = "messageLogInterface" >

                <operation name="opLogMessage"
                                pattern="http://www.w3.org/2006/01/wsdl/out-only">
                        <output messageLabel="out"
                                element="ghns:messageLog" />
                </operation>

        </interface>

        <interface  name="reservationInterface" extends="tns:messageLogInterface" >

                <operation name="opCheckAvailability"
                                pattern="http://www.w3.org/2006/01/wsdl/in-out"
                                style="http://www.w3.org/2006/01/wsdl/style/iri"
                                wsdlx:safe = "true">
                        <input messageLabel="In"
                                element="ghns:checkAvailability" />
                        <output messageLabel="Out"
                                element="ghns:checkAvailabilityResponse" />
                        <outfault ref="tns:invalidDataFault" messageLabel="Out"/>

                </operation>
        </interface>
        ...
</description>
```

Now let's have a look at the element children of `interface`, beginning with `fault`.

2.4.3 Interface Faults

The `fault` element is used to declare faults that may occur during execution of operations of an interface. They are declared directly under `interface`, and referenced from operations where they apply, in order to permit reuse across multiple operations.

Faults are very similar to messages and can be viewed as a special kind of message. Both faults and messages may carry a payload that is normally described by an element declaration. However, WSDL 2.0 treats faults and messages slightly differently. The messages of an operation directly refer to their element declaration, however the faults of an operation indirectly refer to their element declaration via a fault element that is defined on the interface.

The reason for defining faults at the interface level is to allow their reuse across multiple operations. This design is especially beneficial when bindings are defined, since in binding extensions like SOAP there is additional information that is associated with faults. In the case of SOAP, faults have codes and subcodes in addition to a payload. By defining faults at the interface level, common codes and subcodes can be associated with them, thereby ensuring consistency across all operations that use the faults

The `fault` element has a required `name` attribute that must be unique within the parent `interface` element, and permits it to be referenced from operation declarations. The optional `element` attribute can be used to indicate a schema for the content or payload of the fault message. Its value should be the QName of a global element defined in the `types` section. Please note that when other type systems are used to define the schema for a fault message, additional attributes may need to be defined via WSDL 2.0's attribute extension mechanism to allow the schema to be associated with the fault.

2.4.4 Interface Operations

As shown earlier, the `operation` element is used to indicate an operation supported by the containing interface. It associates message schemas with a message exchange pattern (MEP), in order to abstractly describe a simple interaction with a Web service.

2.4.4.1 Operation Attributes

An `operation` has two required attributes and one optional attribute:

- A required `name` attribute, as seen already, which must be unique within the interface.
- A required `pattern` attribute whose value must be an absolute URI that identifies the desired MEP for the `operation`. MEPs are further explained in **2.4.4.3 Understanding Message Exchange Patterns (MEPs)** [p.33] .
- An optional `style` attribute whose value is a list of absolute URIs. Each URI identifies a certain set of rules that were followed in defining this `operation`. It is an error if a particular style is indicated, but the associated rules are not followed. [*WSDL 2.0 Adjuncts [p.83]*] defines a set of styles, including

- RPC Style. The RPC style is selected when the `style` is assigned the value http://www.w3.org/2006/01/wsdl/rpc. It places restrictions for Remote Procedure Call-types of interactions.
- IRI Style. The IRI style is selected when the `style` is assigned the value http://www.w3.org/2006/01/wsdl/style/iri. It places restrictions on message definitions so they may be serialized into something like HTTP URL encoded.
- The Multipart style. The Multipart style is selected when the `style` is assigned the value http://www.w3.org/2006/01/wsdl/style/multipart. In the HTTP binding, for XForms clients, a message must be defined following the Multipart style and serialized as "Multipart/form-data".

You can find more details of these WSDL 2.0 predefined styles. Section **4.4 RPC Style** [p.58] provides an example of using the RPC `style`. [*WSDL 2.0 Adjuncts [p.83]*] provides examples for the IRI style and Multipart style.

Note that [*WSDL 2.0 Adjuncts [p.83]*] provides a predefined extension for indicating operation safety. The `wsdlx:safe` global attribute whose value is a boolean can be used with an operation to indicate whether the operation is asserted to be "safe" (as defined in Section 3.5 of the Web Architecture [*Web Architecture [p.83]*]) for clients to invoke. In essence, a safe operation is any operation that does not give the client any new obligations. For example, an operation that permits the client to check prices on products typically would not obligate the client to buy those products, and thus would be safe, whereas an operation for purchasing products would obligate the client to pay for the products that were ordered, and thus would not be safe.

An operation should be marked safe (by using the `wsdlx:safe` and by setting its value to "true") if it meets the criteria for a safe interaction defined in Section 3.5 of the Web Architecture [*Web Architecture [p.83]*], because this permits the infrastructure to perform efficiency optimizations, such as pre-fetch, re-fetch and caching.

The default value of this attribute is false. If it is false or is not set, then no assertion is made about the safety of the operation; thus the operation may or may not be safe.

2.4.4.2 Operation Message References

An `operation` will also have `input`, `output`,`infault`, and/or `outfault` element children that specify the ordinary and fault message types to be used by that operation. The MEP specified by the `pattern` attribute determines which of these elements should be included, since each MEP has placeholders for the message types involved in its pattern.

Since operations were already discussed in **2.1.4 Defining an Interface** [p.12] , this section will merely comment on additional capabilities that were not previously explained.

2.4.4.2.1 The messageLabel Attribute

The `messageLabel` attribute of the `input` and `output` elements is optional. It is not necessary to explicitly set the `messageLabel` when the MEP in use is one of the eight MEPs predefined in WSDL 2.0 Part 2 [*WSDL 2.0 Adjuncts [p.83]*] and it has only one message with a given direction.

2.4.4.2.2 The element Attribute

The `element` attribute of the `input` and `output` elements is used to specify the message content schema (aka payload schema) when the content model is defined using XML Schema. As we have seen already, it can specify the QName of an element schema that was defined in the `types` section. However, alternatively it can specify one of the following tokens:

`#any`

The message content is any single element.

`#none`

There is no message content, i.e., the message payload is empty.

`#other`

The message content is described by a non-XML type system. Extension attributes specify the type.

The `element` attribute is also optional. If it is not specified, then the message content is described by a non-XML type system.

Note that there are situations that the information conveyed in the `element` attribute is not sufficient for a service implementation to uniquely identify an incoming message and dispatch it to an appropriate operation. In such situations, additional means may be required to aid identifying an incoming message. See **5.1 Enabling Easy Message Dispatch** [p.62] for more detail.

2.4.4.2.3 Multiple infault or outfault Elements

When `infault` and/or `outfault` occur multiple times within an `operation`, they define alternative fault messages.

2.4.4.3 Understanding Message Exchange Patterns (MEPs)

WSDL 2.0 message exchange patterns (MEPs) are used to define the sequence and cardinality of the abstract messages in an operation. By design, WSDL 2.0 MEPs are abstract. First of all, they abstract out specific message types. MEPs identify placeholders for messages, and placeholders are associated with specific message types when an operation is defined, which includes specifying which MEP to use for that operation. Secondly, unless explicitly stated otherwise, MEPs also abstract out binding-specific information like timing between messages, whether the pattern is synchronous or asynchronous, and whether the messages are sent over a single or multiple channels.

It's worth pointing out that WSDL 2.0 MEPs do not exhaustively describe the set of messages that may be exchanged between a service and other nodes. By some prior agreement, another node and/or the service may send other messages (to each other or to other nodes) that are not described by the MEP. For instance, even though an MEP may define a single message sent from a service to one other node, a service defined by that MEP may multicast that message to other nodes. To maximize reuse, WSDL 2.0 message exchange patterns identify a minimal contract between other parties and Web Services, and contain only information that is relevant to both the Web service and the client that engages that service.

A total of eight MEPs are defined in [*WSDL 2.0 Adjuncts [p.83]*]. These MEPs should cover the most common use cases, but they are not meant to be an exhaustive list of MEPs that can ever be used by operations. More MEPs can be defined for particular application needs by interested parties. (See **2.4.4.3 Understanding Message Exchange Patterns (MEPs)** [p.33])

For the eight MEPs defined by WSDL 2.0, some of them are variations of others based on how faults may be generated. For example, the In-Only pattern ("http://www.w3.org/2006/01/wsdl/in-only") consists of exactly one message received by a service from some other node. No fault can be generated. As a variation of In-Only, Robust In-Only pattern ("http://www.w3.org/2006/01/wsdl/robust-in-only") also consists of exactly one message received by a service, but in this case faults can be triggered by the message and must be delivered to the originator of the message. If there is no path to this node, the fault must be discarded. For details about the common fault generation models used by the eight WSDL 2.0 MEPs, see [*WSDL 2.0 Adjuncts [p.83]*].

Depending on how the first message in the MEP is initiated, the eight WSDL 2.0 MEPs may be grouped into two groups: in-bound MEPs, for which the service receives the first message in the exchange, and out-bound MEPs, for which the service sends out the first message in the exchange. (Such grouping is not provided in the WSDL 2.0 specification and is presented here only for the purpose of easy reference in this primer).

A frequently asked question about out-bound MEPs is how a service knows where to send the message. Services using out-bound MEPs are typically part of large scale integration systems that rely on mapping and routing facilities. In such systems, out-bound MEPs are useful for specifying the functionality of a service abstractly, including its requirements for potential customers, while endpoint address information can be provided at deployment or runtime by the underlying integration infrastructure. For example, the GreatH hotel reservation system may require that every time a customer interacts with the system to check availability, data about the customer must be logged by a CRM system. At design time, it's unknown which particular CRM system would be used together with the reservation system. To address this requirement, we may change the "reservationInterface" in Example 2-1 [p.7] to include an out-bound logInquiry operation. This `logInquiry` operation advertises to potential service clients that customer data will be made available by the reservation service at run time. When the reservation service is deployed to GreatH's IT landscape, appropriate configuration time and run time infrastructure will help determine which CRM system will get the customer data and log it appropriately. It's worth noting that in addition to being used by a CRM system for customer management purpose, the same data may also be used by a system performance analysis tool for different purpose. Providing an out-bound operation in the reservation service enables loose coupling and so improves the overall GreatH IT landscape's flexibility and scalability.

Example 2-12. Use of outbound MEPs

```
<description ...>
        ...
        <interface  name="reservationInterface">
                ...
                <operation name="opCheckAvailability" ... >

                <operation name="opLogInquiry"
                                pattern="http://www.w3.org/2006/01/wsdl/out-only">
                        <output messageLabel="Out" element="ghns:customerData" />
                </operation>

        </interface>
        ...
</description>
```

Although the eight MEPs defined in WSDL 2.0 Part 2 [*WSDL 2.0 Adjuncts [p.83]*] are intended to cover most use cases, WSDL 2.0 has designed this set to be extensible. This is why MEPs are identified by URIs rather than a fixed set of tokens.

For more about defining new MEPs, see **4.3 Defining New MEPs** [p.55] .

2.5 More on Bindings

Bindings are used to supply protocol and encoding details that specify *how* messages are to be sent or received. Each `binding` element uses a particular *binding extension* to specify such information. WSDL 2.0 Part 2 [*WSDL 2.0 Adjuncts [p.83]*] defines several binding extensions that are typically used. However, binding extensions that are not defined in WSDL 2.0 Part 2 can also be used, provided that client and service toolkits support them.

Binding information must be supplied for every operation in the interface that is used in an endpoint. However, if the desired binding extension provides suitable defaulting rules, then the information will only need to be explicitly supplied at the interface level, and the defaulting rules will implicitly propagate the information to the operations of the interface. For example, see the Default Binding Rules of SOAP binding extension in WSDL 2.0 Part 2 [*WSDL 2.0 Adjuncts [p.83]*].

2.5.1 Syntax Summary for Bindings

Since bindings are specified using extensions to the WSDL 2.0 language (i.e., binding extensions are not in the WSDL 2.0 namespace), the XML for expressing a binding will consist of a mixture of elements and attributes from WSDL 2.0 namespace and from the binding extension's namespace, using WSDL 2.0's open content model.

Here is a syntax summary for `binding`, simplified by omitting optional `documentation`, `feature` and `property` elements. Bear in mind that this syntax summary only shows the elements and attributes defined within the WSDL 2.0 namespace. When an actual binding is defined, elements and attributes from the namespace of the desired binding extension will also be intermingled as required by that particular binding extension.

```
<description targetNamespace="xs:anyURI" >
  . . .
  <binding name="xs:NCName" interface="xs:QName"? >

    <fault ref="xs:QName" >  </fault>*

    <operation ref="xs:QName" >
      <input messageLabel="xs:NCName"? > </input>*
      <output messageLabel="xs:NCName"? > </output>*
      <infault ref="xs:QName" messageLabel="xs:NCName"? > </infault>*
      <outfault ref="xs:QName" messageLabel="xs:NCName"? > </outfault>*
    </operation>*

  </binding>*
  . . .
</description>
```

The `binding` syntax parallels the syntax of `interface`: each interface construct has a binding counterpart. Despite this syntactic similarity, they are indeed different constructs, since they are in different symbol spaces and are designed for different purposes.

2.5.2 Reusable Bindings

A binding can either be reusable (applicable to any interface) or non-reusable (specified for a particular interface). Non-reusable bindings may be specified at the granularity of the interface (assuming the binding extension provides suitable defaulting rules), or on a per-operation basis if needed. A non-reusable binding was demonstrated in **2.1.5 Defining a Binding** [p.15] .

To define a reusable binding, the `binding` element simply omits the `interface` attribute and omits specifying any operation-specific and fault-specific binding details. Endpoints can later refer to a reusable binding in the same manner as for a non-reusable binding. Thus, a reusable binding becomes associated with a particular interface when it is referenced from an endpoint, because an endpoint is part of a service, and the service specifies a particular interface that it implements. Since a reusable binding does not specify an interface, reusable bindings cannot specify operation-specific details. Therefore, reusable bindings can only be defined using binding extensions that have suitable defaulting rules, such that the binding information only needs to be explicitly supplied at the interface level.

2.5.3 Binding Faults

A binding `fault` associates a concrete message format with an abstract fault of an interface. It describes how faults that occur within a message exchange of an operation will be formatted, since the fault does not occur by itself. Rather, a fault occurs as part of a message exchange specified by an interface `operation` and its binding counterpart, the binding `operation`.

A binding `fault` has one required `ref` attribute which is a reference, by QName, to an `interface fault` . It identifies the abstract interface `fault` for which binding information is being specified. Be aware that the value of `ref` attribute of all the `faults` under a `binding` must be unique. That is, one cannot define multiple bindings for the same interface fault within a given `binding`.

2.5.4 Binding Operations

A binding `operation` describes a concrete binding of an interface operation to a concrete message format. An interface operation is uniquely identified by the WSDL 2.0 target namespace of the interface and the name of the operation within that interface, via the required `ref` attribute of binding `operation`. As with faults, for each `operation` within a `binding`, the value of the `ref` attribute must be unique.

2.5.5 The SOAP Binding Extension

The WSDL 2.0 SOAP Binding Extension (see WSDL 2.0 Part 2 [*WSDL 2.0 Adjuncts [p.83]*]) was primarily designed to support the features of SOAP 1.2 [*SOAP 1.2 Part 1: Messaging Framework [p.84]*]. However, for backwards compatibility, it also provides some support for SOAP 1.1 [*SOAP 1.1 [p.84]*].

An example using the WSDL 2.0 SOAP binding extension was already presented in **2.1.5 Defining a Binding** [p.15] , but some additional points are worth mentioning:

- Because the same binding extension is used for both SOAP 1.2 and SOAP 1.1, a `wsoap:version` attribute is provided to allow you to indicate which version of SOAP you want. If this attribute is not specified, it defaults to SOAP 1.2.
- The WSDL 2.0 SOAP binding extension defines a set of default rules, so that bindings can be specified at the interface level or at the operation level (or both), with the operation level taking precedence. However, it does not define default binding rules for faults. Thus, if a given interface defines any faults, then corresponding binding information must be explicitly provided for each such fault.
- If HTTP is used as the underlying protocol, then the binding can (and should) control whether each operation will use HTTP GET or POST. (See **2.5.7 HTTP GET Versus POST: Which to Use?** [p.41] .)

Here is an example that illustrates both a SOAP 1.2 binding (as seen before) and a SOAP 1.1 binding.

Example 2-13. SOAP 1.2 and SOAP 1.1 Bindings

```
<?xml version="1.0" encoding="utf-8" ?>
<description
  xmlns="http://www.w3.org/2006/01/wsdl"
  targetNamespace="http://greath.example.com/2004/wsdl/resSvc"
  xmlns:tns="http://greath.example.com/2004/wsdl/resSvc"
  xmlns:ghns="http://greath.example.com/2004/schemas/resSvc"
  xmlns:wsoap="http://www.w3.org/2006/01/wsdl/soap"
  xmlns:soap="http://www.w3.org/2003/05/soap-envelope"
  xmlns:soap11="http://schemas.xmlsoap.org/soap/envelope/">

 ....

  <!-- SOAP 1.2 Binding -->
  <binding name="reservationSOAPBinding"
    interface="tns:reservationInterface"
    type="http://www.w3.org/2006/01/wsdl/soap"
    wsoap:protocol="http://www.w3.org/2003/05/soap/bindings/HTTP">
```

```
    <operation ref="tns:opCheckAvailability"
      wsoap:mep="http://www.w3.org/2003/05/soap/mep/request-response"/>

    <fault ref="tns:invalidDataFault"
      wsoap:code="soap:Sender"/>

  </binding>

  <!-- SOAP 1.1 Binding -->
  <binding name="reservationSOAP11Binding"
    interface="tns:reservationInterface"
    type="http://www.w3.org/2006/01/wsdl/soap"
    wsoap:version="1.1"
    wsoap:protocol="http://www.w3.org/2006/01/soap11/bindings/HTTP">

    <operation ref="tns:opCheckAvailability"/>

    <fault ref="tns:invalidDataFault"
      wsoap:code="soap11:Client"/>

  </binding>


  <service name="reservationService"
    interface="tns:reservationInterface">

    <!-- SOAP 1.2 End Point -->
    <endpoint name="reservationEndpoint"
      binding="tns:reservationSOAPBinding"
      address="http://greath.example.com/2004/reservation"/>

    <!-- SOAP 1.1 End Point -->
    <endpoint name="reservationEndpoint2"
      binding="tns:reservationSOAP11Binding"
      address="http://greath.example.com/2004/reservation"/>

  </service>
</description>
```

2.5.5.1 Explanation of Example

Most lines in this example is the same as previously explained in **2.1.5 Defining a Binding** [p.15] , so we’ll only point out lines that are demonstrating something new for SOAP 1.1 binding.

```
<description ... xmlns:soap11="http://schemas.xmlsoap.org/soap/enve-
lope/">
```

This is the namespace for terms defined within the SOAP 1.1 specification [*SOAP 1.1 [p.84]*].

```
<binding...wsoap:version="1.1"
```

This line indicates that this binding uses SOAP 1.1 [*WSDL 2.0 SOAP 1.1 Binding [p.83]*], rather than SOAP 1.2.

```
wsoap:protocol="http://www.w3.org/2005/05/soap11/bindings/HTTP">
```

This line specifies that HTTP should be used as the underlying transmission protocol. See also **2.5.7 HTTP GET Versus POST: Which to Use?** [p.41] .

```
<operation ref="tns:opCheckAvailability"/>
```

Note that `wsoap:mep` is not applicable to SOAP 1.1 binding.

```
<fault...wsoap:code="soap11:Client"/>
```

This line specifies the SOAP 1.1 fault code that will be used in transmitting invalidDataFault.

2.5.6 The HTTP Binding Extension

In addition to the WSDL 2.0 SOAP binding extension described above, WSDL 2.0 Part 2 [*WSDL 2.0 Adjuncts [p.83]*] defines a binding extension for HTTP 1.1 [*IETF RFC 2616 [p.84]*] and HTTPS [*IETF RFC 2818 [p.84]*], so that these protocols can be used natively to send and receive messages, without first encoding them in SOAP.

The HTTP binding extension provides many features to control:

- Which HTTP operation will be used. (GET, PUT, POST, DELETE, and other HTTP operations are supported.)
- Input, output and fault serialization
- Transfer codings
- Authentication requirements
- Cookies
- HTTP over TLS (https)

As with the WSDL 2.0 SOAP binding extension, the HTTP binding extension also provides defaulting rules to permit binding information to be specified at the interface level and used by default for each operation in the affected interface, however, defaulting rules are not provided for binding faults.

Here is an example of using the HTTP binding extension to check hotel room availability at GreatH.

Example 2-14. HTTP Binding Extension

```
<?xml version="1.0" encoding="utf-8" ?>
<description xmlns="http://www.w3.org/2006/01/wsdl"
      . . .
      xmlns:whttp="http://www.w3.org/2006/01/wsdl/http" >
```

```
  . . .
  <binding name="reservationHTTPBinding"
      interface="tns:reservationInterface"
      type="http://www.w3.org/2006/01/wsdl/http"
      whttp:methodDefault="GET">

    <operation ref="tns:opCheckAvailability"
      whttp:location="{checkInDate}"  />
  </binding>

  <service name="reservationService"
    interface="tns:reservationInterface">

    <!-- HTTP 1.1 GET End Point -->
    <endpoint name="reservationEndpoint"
      binding="tns:reservationHTTPBinding"
      address="http://greath.example.com/2004/checkAvailability/"/>

  </service>
  . . .
</description>
```

2.5.6.1 Explanation of Example

Most of this example is the same as previously explained in **2.1.5 Defining a Binding** [p.15] , so we'll only point out lines that are demonstrating something new for HTTP binding extension.

`<description...xmlns:whttp="http://www.w3.org/2006/01/wsdl/http" >`

This defines the namespace prefix for elements and attributes defined by the WSDL 2.0 HTTP binding extension.

`<binding...type="http://www.w3.org/2006/01/wsdl/http"`

This declares the binding as being an HTTP binding.

`whttp:methodDefault="GET">`

The default method for operations in this interface will be HTTP GET.

`whttp:location="{checkInDate}" >`

The `whttp:location` attribute specifies a pattern for serializing input message instance data into the path component of the request URI. The default binding rules for HTTP specify that the default input serialization for GET is `application/x-www-form-urlencoded`. Curly braces are used to specify the name of a schema type in the input message schema, which determines what input instance data will be inserted into the path component of the request URI. The curly brace-enclosed name will be replaced with instance data in constructing the path component. Remaining input instance data (not specified by `whttp:location`) will either be serialized into the query string portion of the URI or into the message body, as follows: if a "/" is appended to a curly brace-enclosed type name, then any remaining input message instance data will be serialized into the message body.

Otherwise it will be serialized into query parameters.

Thus, in this example, each of the elements in the `tCheckAvailability` type will be serialized into the query parameters. A sample resulting URI would therefore be `http://greath.example.com/2004/checkAvailability/5-5-5?checkOut-Date=6-6-5&roomType=foo`.

Here is an alternate example that appends "/" to the type name in order to serialize the remaining instance data into the message body:

Example 2-15. Serializing a Subset of Types in the Path

```
. . .
<operation ref="tns:opCheckAvailability"
    whttp:location="bycheckInDate/{checkInDate/}"  >
. . .
```

This would instead serialize to a request URI such as: `http://greath.example.com/2004/checkAvailability/bycheckInDate/5-5-5`. The rest of the message content would go to the HTTP message body.

2.5.7 HTTP GET Versus POST: Which to Use?

When a binding using HTTP is specified for an operation, the WSDL 2.0 author must decide which HTTP method is appropriate to use -- usually a choice between GET and POST. In the context of the Web as a whole (rather than specifically Web services), the W3C Technical Architecture Group (TAG) has addressed the question of when it is appropriate to use GET, versus when to use POST, in a finding entitled *URIs, Addressability, and the use of HTTP GET and POST* ([*W3C TAG Finding: Use of HTTP GET [p.85]*]). From the abstract:

". . . *designers should adopt [GET] for safe operations such as simple queries. POST is appropriate for other types of applications where a user request has the potential to change the state of the resource (or of related resources). The finding explains how to choose between HTTP GET and POST for an application taking into account architectural, security, and practical considerations.*"

Recall that the concept of a safe operation was discussed in **2.4.4.1 Operation Attributes** [p.31] . (Briefly, a safe operation is one that does not cause the invoker to incur new obligations.) Although the `wsdlx:safe` attribute of an interface operation indicates that the abstract operation is safe, it does not automatically cause GET to be used at the HTTP level when the binding is specified. The choice of GET or POST is determined at the binding level:

- If the WSDL 2.0 SOAP binding extension is used (**2.5.5 The SOAP Binding Extension** [p.37]), with HTTP as the underlying transport protocol, then GET may be specified by setting:

  `wsoap:protocol="http://www.w3.org/2003/05/soap/bindings/HTTP"`

  on the `binding` element (to indicate the use of HTTP as the underlying protocol); and

```
wsoap:mep="http://www.w3.org/2003/05/soap/mep/soap-response/"
```

on the binding `operation` element, which causes GET to be used by default.

- If the WSDL 2.0 HTTP binding extension is used directly (**2.5.6 The HTTP Binding Extension** [p.39]), GET may be specified by setting either:

  ```
  whttp:methodDefault="GET"
  ```

  on the `binding` element; or

  ```
  whttp:method="GET"
  ```

  on the binding `operation` element, which overrides `whttp:methodDefault` if set on the `binding` element; or

  ```
  wsdlx:safe="true"
  ```

  on the bound `interface operation` . When the above two items are not explicitly set, and when the bound interface operation is marked safe, the HTTP Binding will by default set the method to GET.

For example, in the GreatH interface definition shown in Example 2-4 [p.12] , the wsdlx:safe attribute is set to "true". The HTTP binding definition in Example 2-14 [p.39] may take advantage of that and be simplified as below and still have the http method set to GET by default:

Example 2-16. Safety and HTTP Binding

```
<?xml version="1.0" encoding="utf-8" ?>

<binding name="reservationHTTPBinding"

      interface="tns:reservationInterface"

      type="http://www.w3.org/2006/01/wsdl/http" >

    <operation ref="tns:opCheckAvailability"

        whttp:location="{checkInDate}"/>

  </binding>
```

3. Advanced Topics I: Importing Mechanisms

3.1 Importing WSDL

In some circumstances WSDL authors may want to split up a Web service description into two or more documents. For example, if a description is getting long or is being developed by several authors, then it is convenient to divide it into several parts. Another very important case is when you expect parts of the description to be reused in several contexts. Clearly it is undesirable to cut and paste sections of one document into another, since that is error prone and leads to maintenance problems. More importantly, you may need to reuse components that belong to a wsdl:targetNamespace that is different than that of the document you are writing, in which case the rules of WSDL 2.0 prevent you from simply cutting and pasting them into your document.

To solve these problems, WSDL 2.0 provides two mechanisms for modularizing Web service description documents: `import` and `include`. This section discusses the import mechanism and describes some typical cases where it may be used.

The `import` mechanism lets one refer to the definitions of Web service components that belong to other namespaces. To illustrate this, consider the GreatH hotel reservation service. Suppose that the reservation service uses a standard credit card validation service that is provided by a financial services company. Furthermore, suppose that companies in the financial services industry decided that it would be useful to report errors in credit card validation using a common set of faults, and have defined these faults in the following Web service description:

Example 3-1. Standard Credit Card Validation Faults (credit-card-faults.wsdl)

```
<?xml version="1.0" encoding="utf-8" ?>
<description xmlns="http://www.w3.org/2006/01/wsdl"
        targetNamespace="http://finance.example.com/CreditCards/wsdl"
        xmlns:tns="http://finance.example.com/CreditCards/wsdl"
        xmlns:cc="http://finance.example.com/CreditCards/xsd">

  <documentation>
        This document describes standard faults for use
      by Web services that process credit cards.
  </documentation>

  <types>
        <xs:import xmlns:xs="http://www.w3.org/2001/XMLSchema"
            namespace="http://finance.example.com/CreditCardFaults/xsd"
            schemaLocation="credit-card-faults.xsd" />
  </types>

  <interface name="creditCardFaults">

        <fault name="cancelledCreditCard" element="cc:CancelledCreditCard">
            <documentation>Thrown when the credit card has been cancelled.</documentation>
        </fault>

        <fault name="expiredCreditCard" element="cc:ExpiredCreditCard">
            <documentation>Thrown when the credit card has expired.</documentation>
        </fault>

        <fault name="invalidCreditCardNumber" element="cc:InvalidCreditCardNumber">
            <documentation>Thrown when the credit card number is invalid.
                This fault will occur if the wrong credit card type is specified.
          </documentation>
        </fault>

        <fault name="invalidExpirationDate" element="cc:InvalidExpirationDate">
            <documentation>Thrown when the expiration date is invalid.</documentation>
        </fault>
```

```
  </interface>

</description>
```

This example defines an interface, `creditCardFaults`, that contains four faults, `cancelledCreditCard`, `expiredCreditCard`, `invalidCreditCardNumber`, and `invalidExpirationDate`. These components belong to the namespace `http://finance.example.com/CreditCards/wsdl`.

Because these faults are defined in a different wsdl:targetNamespace than the one used by the GreatH Web service description, import must be used to make them available within the GreatH Web service description, as shown in the following example:

Example 3-2. Using the Standard Credit Card Validation Faults (use-credit-card-faults.wsdl)

```
<?xml version="1.0"?>
<description
        targetNamespace="http://greath.example.com/2004/wsdl/resSvc"
        xmlns:ghns="http://greath.example.com/2004/schemas/resSvc"
        xmlns:cc="http://finance.example.com/CreditCards/wsdl"
        xmlns="http://www.w3.org/2006/01/wsdl"
        xmlns:xs="http://www.w3.org/2001/XMLSchema">

        <documentation>
                Description: The definition of the reservation Web service of
                GreatH hotel. Author: Joe Somebody Date: 05/17/2004
        </documentation>

        <import namespace="http://finance.example.com/CreditCards/wsdl"
              location="credit-card-faults.wsdl"/>
        . . .
        <interface name="reservation" extends="cc:creditCardFaults">
                . . .
                <operation name="makeReservation"
                        pattern="http://www.w3.org/2006/01/wsdl/in-out">

                        <input messageLabel="In" element="ghns:makeReservation" />

                        <output messageLabel="Out"
                                element="ghns:makeReservationResponse" />

                        <outfault ref="invalidDataFault" messageLabel="Out" />

                        <outfault ref="cc:cancelledCreditCard" messageLabel="Out" />
                        <outfault ref="cc:expiredCreditCard" messageLabel="Out" />
                        <outfault ref="cc:invalidCreditCardNumber" messageLabel="Out" />
                        <outfault ref="cc:invalidExpirationDate" messageLabel="Out" />
                </operation>
        </interface>
</description>
```

The hotel reservation service declares that it is using components from another namespace via the `import>` element. The import element has a required `namespace` attribute that specifies the other namespace, and an optional `location` attribute that gives the processor a hint where to find the description of the other namespace. The `reservation` interface extends the `creditCardFault` interface from the other namespace in order to make the faults available in the reservation interface. Finally, the `makeReservation` operation refers to the standard faults in its `outfault` elements.

Another typical situation for using imports is to define a standard interface that is to be implemented by many services. For example, suppose the hotel industry decided that it was useful to have a standard interface for making reservations. This interface would belong to some industry association namespace, e.g. `http://hotels.example.com/reservations/wsdl`. Each hotel that implemented the standard reservation service would define a service in its own namespace, e.g. `http://greath.example.com/2004/wsdl/resSvc`. The description of each service would import the `http://hotels.example.com/reservations/wsdl` namespace and refer to the standard reservation interface in it.

3.2 Importing Schemas

WSDL 2.0 documents may contain one or more XML schemas defined within the `wsdl:types` element. This section illustrates the correct way to refer to these schemas, both from within the same document and from other documents.

3.2.1 Schemas in Imported Documents

In this example, we consider some GreatH Hotel Web services that retrieve and update reservation details. The retrieval Web service is defined in the `retrieveDetails.wsdl` WSDL 2.0 document, along with a schema for the message format. The updating Web service is defined in the `updateDetails.wsdl` WSDL 2.0 document which imports the first document and refers to both WSDL 2.0 and schema definitions contained in the imported document.

Example 3-3 [p.45] shows the definition of the retrieval Web service in the `http://greath.example.com/2004/services/retrieveDetails` namespace. This WSDL 2.0 document also contains an inline schema that describes the reservation detail in the `http://greath.example.com/2004/schemas/reservationDetails` namespace. This schema is visible to the `retrieveDetailsInterface` interface definition which refers to it in the `retrieve` operation's output message.

Example 3-3. The Retrieve Reservation Details Web Service: retrieveDetails.wsdl

```
                                                        <?xml version="1.0" encoding="utf-8" ?>
<description xmlns="http://www.w3.org/2006/01/wsdl"
        targetNamespace="http://greath.example.com/2004/services/retrieveDetails"
        xmlns:tns="http://greath.example.com/2004/services/retrieveDetails"
        xmlns:wdetails="http://greath.example.com/2004/schemas/reservationDetails"
        xmlns:xs="http://www.w3.org/2001/XMLSchema">

        <documentation>
                This document describes the GreatH Retrieve Reservation Details
                Web service.
        </documentation>
```

```
	<types>
		<xs:schema xmlns="http://www.w3.org/2001/XMLSchema"
			targetNamespace="http://greath.example.com/2004/schemas/reservationDetails">

			<xs:element name="reservationDetails">
				<xs:complexType>
					<xs:sequence>
						<xs:element name="confirmationNumber"
							type="string" />
						<xs:element name="checkInDate" type="date" />
						<xs:element name="checkOutDate" type="date" />
						<xs:element name="roomType" type="string" />
						<xs:element name="smoking" type="boolean" />
					</xs:sequence>
				</xs:complexType>
			</xs:element>
		</xs:schema>
	</types>

	<interface name="retrieveDetailsInterface">

		<operation name="retrieve"
			pattern="http://www.w3.org/2006/01/wsdl/in-out">
			<input messageLabel="In" element="#none" />
			<output messageLabel="Out"
				element="wdetails:reservationDetails" />
		</operation>

	</interface>

</description>
```

Example 3-4 [p.47] shows the definition of the updating Web service in the `http://greath.example.com/2004/services/updateDetails` namespace. The `updateDetailsInterface` interface extends the `retrieveDetailsInterface` interface. However, the `retrieveDetailsInterface` belongs to the `http://greath.example.com/2004/services/retrieveDetails` namespace, so `updateDetails.wsdl` must import `retrieveDetails.wsdl` to make that namespace visible.

The `updateDetailsInterface` interface also uses the `reservationDetails` element definition that is contained in the inline schema of the imported `retrieveDetails.wsdl` document. However, this schema is not automatically visible within the `updateDetails.wsdl` document. To make it visible, the `updateDetails.wsdl` document must import the namespace of the inline schema within the `types` element using the XML schema `import` element.

In this example, the `schemaLocation` attribute of the `import` element has been omitted. The `schemaLocation` attribute is a hint to the WSDL 2.0 processor that tells it where to look for the imported schema namespace. However, the WSDL 2.0 processor has already processed the `retrieveDetails.wsdl` document which contains the imported namespace in an inline schema so it should not need any hints. However, this behavior depends on the implementation of the processor and so cannot be relied on.

Although the WSDL 2.0 document may validly omit the `schemaLocation` attribute, it is a best practice to either provide a reliable value for it or move the inline schema into a separate document, say `reservationDetails.xsd`, and directly import it in the `types` element of both `retrieveDetails.wsdl` and `updateDetails.wsdl`. In general, schemas that are expected to be referenced from more than one WSDL 2.0 document should be defined in a separate schema document rather than be inlined.

Example 3-4. The Update Reservation Details Web Service: updateDetails.wsdl

```
<?xml version="1.0" encoding="utf-8" ?>
<description xmlns="http://www.w3.org/2006/01/wsdl"
        targetNamespace="http://greath.example.com/2004/services/updateDetails"
        xmlns:tns="http://greath.example.com/2004/services/updateetails"
        xmlns:retrieve="http://greath.example.com/2004/services/retrieveDetails"
        xmlns:details="http://greath.example.com/2004/schemas/reservationDetails"
        xmlns:xs="http://www.w3.org/2001/XMLSchema">

        <documentation>
                This document describes the GreatH Update Reservation Details
                Web service.
        </documentation>

        <import
                namespace="http://greath.example.com/2004/services/retrieveDetails"
                location="retrieveDetails.wsdl" />

        <types>
                <xs:import
                        namespace="http://greath.example.com/2004/schemas/reservationDetails" />
        </types>

        <interface name="updateDetailsInterface"
                extends="retrieve:retrieveDetailsInterface">

                <operation name="update"
                        pattern="http://www.w3.org/2006/01/wsdl/in-out">
                        <input messageLabel="In"
                                element="details:reservationDetails" />
                        <output messageLabel="Out"
                                element="details:reservationDetails" />
                </operation>

        </interface>

</description>
```

3.2.2 Multiple Inline Schemas in One Document

A WSDL 2.0 document may define multiple inline schemas in its `types` element. The two or more schemas may have the same target namespace provided that they do not define the same elements or types. It is an error to define the same element or type more than once, even if the definitions are identical.

Each namespace of an inline schema becomes visible to the Web service definitions. However, the namespaces are not automatically visible to the other inline schemas. Each inline schema must explicitly import any other namespace it references. The `schemaLocation` attribute is not required in this case since the WSDL 2.0 processor knows the location of each schema by virtue of having processed the enclosing WSDL 2.0 document.

To illustrate this, consider Example 3-5 [p.48] which contains two inline schemas. The `http://greath.example.com/2004/schemas/reservationItems` namespace contains some elements for items that appear in the reservation details. The `http://greath.example.com/2004/schemas/reservationDetails` namespace contains the `reservationDetails` element which refers to the item elements. The schema for the `http://greath.example.com/2004/schemas/reservationDetails` namespace contains an `import` element that imports the `http://greath.example.com/2004/schemas/reservationItems` namespace. No `schemaLocation` attribute is required for this import since the schema is defined inline in the importing document.

Example 3-5. Multiple Inline Schemas: retrieveItems.wsdl

```
<?xml version="1.0" encoding="utf-8" ?>
<description xmlns="http://www.w3.org/2006/01/wsdl"
        targetNamespace="http://greath.example.com/2004/services/retrieveDetails"
        xmlns:tns="http://greath.example.com/2004/services/retrieveDetails"
        xmlns:wdetails="http://greath.example.com/2004/schemas/reservationDetails"
        xmlns:xs="http://www.w3.org/2001/XMLSchema">

        <documentation>
                This document describes the GreatH Retrieve Reservation Details
                Web service.
        </documentation>

        <types>

                <xs:schema targetNamespace="http://greath.example.com/2004/schemas/reservationItems">

                        <xs:element name="confirmationNumber" type="string" />
                        <xs:element name="checkInDate" type="date" />
                        <xs:element name="checkOutDate" type="date" />
                        <xs:element name="roomType" type="string" />
                        <xs:element name="smoking" type="boolean" />

                </xs:schema>

                <xs:schema targetNamespace="http://greath.example.com/2004/schemas/reservationDetails"
                        xmlns:items="http://greath.example.com/2004/schemas/reservationItems">

                        <xs:import
                                namespace="http://greath.example.com/2004/schemas/reservationItems" />

                        <xs:element name="reservationDetails">
                                <xs:complexType>
                                        <xs:sequence>
                                                <xs:element ref="items:confirmationNumber" />
                                                <xs:element ref="items:checkInDate" />
                                                <xs:element ref="items:checkOutDate" />
                                                <xs:element ref="items:roomType" />
                                                <xs:element ref="items:smoking" />
                                        </xs:sequence>
                                </xs:complexType>
                        </xs:element>
                </xs:schema>

        </types>

        <interface name="retrieveDetailsInterface">

                <operation name="retrieve"
                        pattern="http://www.w3.org/2006/01/wsdl/in-out">
                        <input messageLabel="In" element="#none" />
```

```
                <output messageLabel="Out"
                        element="wdetails:reservationDetails" />
        </operation>

    </interface>

</description>
```

3.2.3 The schemaLocation Attribute

In the preceding examples, schemas were defined inline in WSDL 2.0 documents. This section discusses the correct way to specify a `schemaLocation` attribute on a schema `import` element to provide a processor with a hint for locating these schemas.

Example 3-4 [p.47] shows how one WSDL 2.0 document imports a schema defined in another, i.e. Example 3-3 [p.45] . Similarly, Example 3-5 [p.48] shows how one schema in a WSDL 2.0 document imports another schema defined in the same document. In both of these examples, the `schemaLocation` attribute was omitted since the WSDL 2.0 processor was assumed to know how to locate the imported schemas because they were part of the WSDL 2.0 documents being processed. The `schemaLocation` attribute can be used to give the processor a URI reference that explicitly locates the schemas. A URI reference is a URI plus an optional fragment identifier that indicates part of the resource. For schemas, the fragment should identify the `schema` element. The simplest way to accomplish this is to use the `id` attribute, however XPointer (see [*XPointer Framework [p.85]*]) can also be used.

3.2.3.1 Using the id Attribute to Identify Inline Schemas

Example 3-6 [p.49] shows the use of the `id` attribute. Both of the inline schemas have `id` attributes. The id of the `http://greath.example.com/2004/schemas/reservationItems` schema is `items` and the id of the `http://greath.example.com/2004/schemas/reservationDetails` schema is `details`. The `import` element in the `http://greath.example.com/2004/schemas/reservationDetails` schema uses the id of the `http://greath.example.com/2004/schemas/reservationItems` schema in the `schemaLocation` attribute, i.e. `#items`.

Example 3-6. Using Ids in Inline Schemas: schemaIds.wsdl

```
<?xml version="1.0" encoding="utf-8" ?>
<description xmlns="http://www.w3.org/2006/01/wsdl"
        targetNamespace="http://greath.example.com/2004/services/retrieveDetails"
        xmlns:tns="http://greath.example.com/2004/services/retrieveDetails"
        xmlns:wdetails="http://greath.example.com/2004/schemas/reservationDetails"
        xmlns:xs="http://www.w3.org/2001/XMLSchema">

        <documentation>
                This document describes the GreatH Retrieve Reservation Details
                Web service.
        </documentation>

        <types>

                <xs:schema id="items"
                        targetNamespace="http://greath.example.com/2004/schemas/reservationItems">

                        <xs:element name="confirmationNumber" type="string" />
                        <xs:element name="checkInDate" type="date" />
```

```
                <xs:element name="checkOutDate" type="date" />
                <xs:element name="roomType" type="string" />
                <xs:element name="smoking" type="boolean" />

        </xs:schema>

        <xs:schema id="details"
                targetNamespace="http://greath.example.com/2004/schemas/reservationDetails"
                xmlns:items="http://greath.example.com/2004/schemas/reservationItems">

                <xs:import
                        namespace="http://greath.example.com/2004/schemas/reservationItems"
                        schemaLocation="#items" />

                <xs:element name="reservationDetails">
                        <xs:complexType>
                                <xs:sequence>
                                        <xs:element ref="items:confirmationNumber" />
                                        <xs:element ref="items:checkInDate" />
                                        <xs:element ref="items:checkOutDate" />
                                        <xs:element ref="items:roomType" />
                                        <xs:element ref="items:smoking" />
                                </xs:sequence>
                        </xs:complexType>
                </xs:element>
        </xs:schema>

    </types>

    <interface name="retrieveDetailsInterface">

        <operation name="retrieve"
                pattern="http://www.w3.org/2006/01/wsdl/in-out">
                <input messageLabel="In" element="#none" />
                <output messageLabel="Out"
                        element="wdetails:reservationDetails" />
        </operation>

    </interface>

</description>
```

4. Advanced Topics II: Extensibility and Predefined Extensions

4.1 Extensibility

WSDL 2.0 provides two extensibility mechanisms: an open content model, which allows XML elements and attributes from other (non-WSDL 2.0) XML namespaces to be interspersed in a WSDL 2.0 document; and Features and Properties. Both mechanisms use URIs to identify the semantics of the extensions. For extension XML elements and attributes, the namespace URI of the extension element or attribute acts as an unambiguous name for the semantics of that extension. For Features and Properties, the Feature or Property is named by a URI.

In either case, the URI that identifies the semantics of an extension should be dereferenceable to a document that describes the semantics of that extension. As of this writing, there is no generally accepted standard for what kind of document that should be. However, the W3C TAG has been discussing the issue (see TAG issue namespaceDocument-8) and is likely to provide guidance at some point.

4.1.1 Optional Versus Required Extensions

Extensions can either be required or optional.

An *optional* extension is one that the client may either engage or ignore, entirely at its discretion, and is signaled by attribute `wsdl:required="false"` or the absence of the `wsdl:required` attribute (because it defaults to false). Thus, a WSDL 2.0 processor, acting on behalf of the client, that encounters an unknown optional extension can safely ignore it and continue to process the WSDL 2.0 document. However, it is important to stress that optional extensions are only optional to the *client* -- not the service. A service must support all optional and required extensions that it advertises in its WSDL 2.0 document.

A *required* extension is one that must be supported and engaged by the client in order for the interaction to proceed properly, and is signaled by attribute `wsdl:required="true"`. If a WSDL 2.0 processor, acting on behalf of the client, encounters a required extension that it does not recognize or does not support, then it cannot safely continue to process the WSDL 2.0 document. In most practical cases, this is likely to mean that the processor will require manual intervention to deal with the extension. For example, a client developer might manually provide an implementation for the required extension to the WSDL 2.0 processor.

4.2 Features and Properties

Editorial note: KevinL	20050519
The section is subject to change. Pending on the resolution of the minority opinions filed about Feature and Property.	

After a few successful trials of the reservation service, GreatH decides that it is time to make the makeReservation operation secure, so that sensitive credit-card information is not being sent across the public network in a snoopable fashion. We will do this using the WSDL 2.0 Features and Properties mechanisms [*WSDL 2.0 Core [p.83]*], which is modeled after the Features and Properties mechanism defined in SOAP 1.2 [*SOAP 1.2 Part 1: Messaging Framework [p.84]*].

To facilitate presentation, this section will assume the existence of a hypothetical security feature named "`http://features.example.com/2005/securityFeature`", which defines, in the abstract, the idea of message confidentiality. This feature has an associated property, named "`http://features.example.com/2005/securityFeature/securityLevel`", which defines various safety levels (from 0 meaning clear text, all the way through 10, involving highly complex cryptographic algorithms with keys in the tens of thousands of bits). We also assume that a SOAP module (for more about SOAP module, see SOAP1.2 spec and **4.2.1 SOAP Modules** [p.52]), named "`http://features.example.com/2005/modules/Security`", has been defined, which implements the security feature described above.

GreatH has chosen an abstract security feature which is standard in the fictitious hotels community, and has integrated both a SOAP module and a new secure HTTP binding into its infrastructure – both of which implement the security feature (the SOAP module does this inside the SOAP envelope using headers, and the secure binding does it at the transport layer). Now they’d like to advertise and control the usage of these extensions using WSDL 2.0.

4.2.1 SOAP Modules

The first step GreatH takes is to require the usage of the SOAP module in their normal SOAP/HTTP endpoint, which looks like this:

Example 4-1. Requiring a SOAP Module in an Endpoint

```
. . .
<service name="reservationService"
       interface="tns:reservationInterface">

  <endpoint name="reservationEndpoint"
            binding="tns:reservationSOAPBinding"
            address ="http://greath.example.com/2004/reservation">
    <wsoap:module uri="http://features.example.com/2005/modules/Security"
              required="true"/>
  </endpoint>

</service>
. . .
```

This syntax indicates that a SOAP Module is required by this endpoint. This means that anyone using this endpoint must both understand the specification that the module URI references, and must use that specification when communicating with the endpoint in question, which typically means including appropriate SOAP headers on transmitted messages.

If the "required" attribute was not present, or if it was set to "`false`", then the `<wsoap:module>` syntax would indicate optional the availability of the referenced module, rather than a requirement to engage it, as explained in **4.1.1 Optional Versus Required Extensions** [p.51] .

4.2.2 Abstract Features

Since GreatH began the web service improvements, they have been talking to several travel agents. The possibility of making their simple hotel interface an industry standard amongst a consortium of hotels has come up, and as such they would like to enable specifying the requirement for the "makeReservation" operation to be secure at the interface level – in other words indicating that the operation must be secure, but without specifying exactly how that should concretely be achieved (to enable maximal reuse of the interface). The next example uses the WSDL 2.0 Feature element to indicate this.

Example 4-2. Declaring an Abstract Feature Requirement

```
. . .
<interface name="reservationInterface">
 <operation name="makeReservation">
  <feature uri="http://features.example.com/2005/securityFeature"
           required="true"/>
  . . . [The rest of the operation is unchanged] . . .
 </operation>
</interface>
. . .
```

This declaration indicates that understanding of, and compliance with, the specified security feature is required for all uses of the "makeReservation" operation. The security feature is *abstract*, which means that although it defines semantics and a level of detail about its general operation, it expects a concrete component (like a SOAP module or binding) to actually realize the functionality.

By definition, if you understand a SOAP module, you understand which (if any) abstract features it implements. Therefore, since the security module in this example is defined as an implementation of the abstract security feature, we know that the use of this module satisfies the requirement to implement the feature. Therefore users of the HTTP endpoint shown above (with the required SOAP module) will be able to make use of it. GreatH also defines a new endpoint:

Example 4-3. A SOAP Binding Over a Secure HTTP Protocol

```
. . .
<binding name="reservationSecureSOAPBinding"
    interface="tns:reservationInterface"
    type="http://www.w3.org/2006/01/wsdl/soap"
    wsoap:protocol="http://bindings.example.com/SOAPBindings/secureHTTP">
  . ..
</binding>
. . .
<service name="reservationService">
  . . .
  <endpoint name="secureReservationEndpoint"
            binding="tns:reservationSecureSOAPBinding"
            address="https://greath.example.com/2004/secureReservation"/>
</service>
. . .
```

The user will have a choice as to which of the endpoints, and therefore which binding, is to be used, but they both satisfy the abstract feature requirement specified in the interface.

Note that it is not necessary to declare the abstract feature in order to use/require the SOAP module, or in order to use/require the secure binding. Abstract feature declarations serve purely to indicate requirements which must be fulfilled by more concrete components such as modules or bindings. In other words, the abstract feature declaration allows components such as interfaces to be reused without caring exactly which SOAP modules or bindings satisfy the feature.

4.2.3 Properties

So far we’ve discussed how to indicate the availability or the "requiredness" of features and modules. Often it is not enough to indicate that a particular extension is available/required: you also need some way to control or parameterize aspects of its behavior. This is achieved by the use of WSDL 2.0 *properties*. Each feature, SOAP module, or SOAP binding may express a variety of *properties* in its specification. These properties are very much like variables in a programming language. If GreatH would like to indicate that the `securityLevel` property should be 5 for the "makeReservation" operation, it would look like this:

Example 4-4. Defining a Property

```
. . .
<interface name="reservationInterface">
 <operation name="makeReservation">
  <property
      uri="http://features.example.com/2005/securityFeature/securityLevel">
   <value>5</value>
  </property>
  . . . [rest of operation definition] . . .
 </operation>
</interface>
. . .
```

The `property` element specifies which property is to be set. By setting the `value` element, a toolkit processing this WSDL 2.0 document is informed that the securityLevel property must be set to 5. The particular meanings of any such values are up to the implementations of the modules/bindings that use them. The `property` element can be placed at many different levels in a WSDL 2.0 document (see "Property Composition Model" section in WSDL 2.0 Part 1 [*WSDL 2.0 Core [p.83]*]).

It is also possible to provide a *constraint* on the value space for a given property. This allows the author of the WSDL 2.0 document to indicate that several valid values for the property are possible for a given scope, limiting the value space already described in the specification that defined the property. Let's extend our example to make this clearer.

The security feature specification defines securityLevel as an integer with values between 1 and 10, each of which indicates, according to the spec, a progressively higher level of security. The GreatH service authors, having read the relevant specifications, have decided that any security level between 3 and 7 will be supported by their infrastructure. Levels less than 3 are deemed unsafe for GreatH's purposes, and levels greater than 7 require too much in the way of resources to make it worthwhile. We can express this in WSDL 2.0 as follows:

Example 4-5. Defining Property Constraints

```
. . .
<types>
 <schema>
  <simpleType name="securityLevelConstraint">
   <restriction base="xs:int">
    <min 3, max 7> <!-- check schema for syntax -->
   </restriction>
  </simpleType>
 </schema>
</types>
. . .
<property uri="http://features.example.com/2005/securityFeature/securityLevel">
  <constraint type="tns:securityLevelConstraint">
</property>
. . .
```

First we define, in the `types` section, an XML Schema restriction type over integers with minimum and maximum values, per our discussion above. Then instead of using the `value` element inside `property`, we use `constraint` and refer to the restriction type. This informs the implementation that the property must have the appropriate values. This information might be useful to a deployment user interface, for example, which might allow an administrator to set this value with a slider when deploying the service.

4.3 Defining New MEPs

As we mentioned in **2.4.4.3 Understanding Message Exchange Patterns (MEPs)** [p.33] , even though the 8 MEPs defined by WSDL 2.0 are intended to cover most of the common use cases, there are situations that require new MEPs to be defined. In this section, we will explain how new MEPs can be defined to address special business requirements.

Following the wild success of its reservation service, GreatH discovered that it could radically increase tourist interest by supplying information on weather conditions, both to travel agents and to the general touring public. This produced a challenge for the service implementers: how could this information be supplied to interested parties without requiring knowledge of web service technology specifically, and of computers generally? At issue was the desire to provide asynchronous updates to unsophisticated customers without incurring onerous overheads for technical support.

The solution adopted was to create a standard mailing list, and to make available a small cross-platform web service client (actually, a subscriber) that could be installed on any computer with POP or IMAP access to a mailbox. The mailbox, once signed up for the mailing list, could either be processed as "dedicated" (to the GreatH weather service; travel agents did this) or as "general purpose" (in which case the application would only examine those emails that contained Subject headers associated with the service). This required development of a binding to email, which is out of scope for this example, but the resulting WSDL 2.0 was otherwise quite straightforward.

Note: the email binding in use here supports publish/subscribe, by supporting the robust-out-only MEP as well as the client/server style in-out used for subscribing and unsubscribing. Details of this binding would require a document as long as the primer, so play along.

Example 4-6. Weather Notification Service (Initial)

```
<?xml version="1.0" encoding="utf-8" ?>
<description xmlns="http://www.w3.org/2006/01/wsdl"
      targetNamespace="http://greath.example.com/2004/wsdl/weathSvc.wsdl"
      xmlns:tns="http://greath.example.com/2004/wsdl/weathSvc.wsdl"
      xmlns:wsoap="http://www.w3.org/2006/01/wsdl/soap"
      xmlns:email="http://www.example.com/webservices/email" >

    <types>
        . . .
    </types>

    <interface name="weatherInterface">
        <operation name="opSubscribeWeather"
                   pattern="http://www.w3.org/2006/01/wsdl/in-out">
            <input element=". . ." />
            <output element=". . ." />
```

```
        </operation>
        <operation name="opUnsubscribeWeather"
                   pattern="http://www.w3.org/2006/01/wsdl/in-out">
            <output element=". . ." />
            <input element=". . ." />
        </operation>
        <operation name="opNotifyWeather"
                   pattern="http://www.w3.org/2006/01/wsdl/robust-out-only">
            <output element=". . ." />
        </operation>
    </interface>

    <binding name="weatherMailingListBinding"
             interface="tns:weatherInterface
             type="http://www.w3.org/2006/01/wsdl/soap"
             wsoap:protocol="http://www.example.com/bindings/email">
             . . .
    </binding>

    <service name="weatherService"
             interface="tns:weatherInterface">
        <endpoint name="greatHWeatherList"
                  binding="tns:weatherMailingListBinding"
                  address="mailto:weather-owner@greath.example.com" />
    </service>

</description>
```

Note: in the example, the messageLabels of all input and output elements have been elided, as they are not necessary to disambiguate (but note that the order of input and output elements is not significant).

Unfortunately, the service was soon highjacked for the purpose of annoyment. Repeatedly, hotels in less salubrious climes, and the victims of various natural climactic disasters (hurricanes, tornadoes) found themselves signed up to receive material full of incomprehensible pointy brackets. They complained to GreatH, who complained to their service designers.

Applying public key infrastructure to solving the problem was immediately rejected as too complex and too heavyweight. Analysis showed that the problem was simply to verify that the address requesting information actually wanted that information. Consequently, a new message exchange pattern was defined.

4.3.1 Confirmed Challenge

This pattern consists of two or more messages in order as follows:

1. A message:

   - indicated by a Message Label component whose message label is "Request" and direction is "in"
   - received from some node N1

2. A message:
   - indicated by a Message Label component whose message label is "Challenge" and direction is "out"
   - sent to some node N2 (which *may* be the same node as N1)
3. An optional message:
   - indicated by a Message Label component whose message label is "Confirmation" and direction is "in"
   - received from node N2
4. An optional message:
   - indicated by a Message Label component whose message label is "Response" and direction is "out"
   - sent to node N2

This pattern uses the rule Message Triggers Fault.

An operation using this message exchange pattern has a pattern property with the value "http://www.example.com/webservices/meps/confirmed-challenge".

Once the MEP had been defined (and the email binding specification appropriately modified to indicate that this was a supported MEP), the service was redefined and redeployed. Only the changed operations are shown in the excerpt below.

Example 4-7. Weather Notification Service (Revised)

```
<?xml version="1.0" encoding="utf-8" ?>
<description xmlns="http://www.w3.org/2006/01/wsdl"
      targetNamespace="http://greath.example.com/2004/wsdl/weathSvc.wsdl"
      xmlns:tns="http://greath.example.com/2004/wsdl/weathSvc.wsdl"
      xmlns:wsoap="http://www.w3.org/2006/01/wsdl/soap"
      xmlns:email="http://www.example.com/webservices/email" >

   . . .

    <interface name="weatherInterface">
        <operation name="opSubscribeWeather"
                   pattern="http://www.example.com/webservices/meps/confirmed-challenge">
            <input messageLabel="Request" element=". . ." />
            <output messageLabel="Challenge" element=". . ." />
            <input messageLabel="Confirmation" element=". . ." />
            <output messageLabel="Response" element=". . ." />
        </operation>
        <operation name="opUnsubscribeWeather"
                   pattern="http://www.example.com/webservices/meps/confirmed-challenge">
            <output messageLabel="Challenge" element=". . ." />
            <output messageLabel="Response" element=". . ." />
```

```
            <input messageLabel="Confirmation" element=". . ." />
            <input messageLabel="Request" element=". . ." />
        </operation>
        . . .
    </interface>

    . . .

</description>
```

Note: in the second example, the input and output examples are not in the sequence in which they occur in the pattern; this illustrates that the sequence is not significant. Note, however, that for this pattern, the messageLabel attribute is required on every input and output element.

4.4 RPC Style

Section **2.4.4.1 Operation Attributes** [p.31] mentioned that the (optional) `style` attribute of an interface operation is used to indicate that the operation conforms to a particular pre-defined operation style, or set of constraints. Actually, if desired the `style` attribute can hold a list of URIs, indicating that the operation simultaneously conforms to multiple styles.

Operation styles are named using URIs, in order to be unambiguous while still permitted new styles to be defined without requiring updates to the WSDL 2.0 language. WSDL 2.0 Part 2 [*WSDL 2.0 Adjuncts [p.83]*] defines three such operation styles; one of these is the RPC Style (RPC Style).

The *RPC Style* is designed to facilitate programming language bindings to WSDL 2.0 constructs. It allows a WSDL 2.0 interface operation to be easily mapped to a method or function signature, such as a method signature in Java(TM) or C#. RPC Style is restricted to operations that use the In-Out or In-Only MEPs (see **2.4.4.3 Understanding Message Exchange Patterns (MEPs)** [p.33]).

A WSDL 2.0 document makes use of the RPC Style in an interface operation by first defining the operation in conformance with all of the RPC Style rules, and then setting that operation's `style` attribute to include the URI that identifies the RPC Style, thus asserting that the operation does indeed conform to the RPC Style. These rules permit the input and output message schemas to map conveniently to inputs and outputs of a method signature. Roughly, input elements map to input parameters, output elements map to output parameters, and elements that appear both in the input and output message schemas map to input/output parameters. WSDL 2.0 Part 2 section "RPC Style" provides full details of the mapping rules and requirements.

The RPC Style also permits the full signature of the intended mapping to be indicated explicitly, using the `wrpc:signature` attribute defined in WSDL 2.0 Part 2 section "wrpc:signature Extension". This is an (optional) extension to the WSDL 2.0 language whose value designates how input and output message schema elements map to input and output parameters in the method signature.

The example below illustrates how RPC Style may be used to designate a signature. This example is a modified version of the GreatH reservation service. In particular, the `interface` and `types` sections have been modified to specify and conform to the RPC Style.

Example 4-8. Specifying RPC Style

```
. . .
<types>

  <xs:element name="checkAvailability">
    <xs:complexType>
      <xs:sequence>
          <xs:element  name="checkInDate" type="xs:date"/>
          <xs:element  name="checkOutDate" type="xs:date"/>
          <xs:element  name="roomType" type="xs:string"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>

  <xs:element name="checkAvailabilityResponse">
    <xs:complexType>
      <xs:sequence>
         <xs:element  name="roomType" type="xs:string"/>
         <xs:element  name="rateType" type="xs:string"/>
         <xs:element  name="rate" type="xs:double"/>
     </xs:sequence>
    </xs:complexType>
  </xs:element>
  . . .
</types>

<interface  name = "reservationInterface" >

  <operation name="checkAvailability"
        pattern="http://www.w3.org/2006/01/wsdl/in-out"
        style="http://www.w3.org/2006/01/wsdl/rpc"
        wrpc:signature=
          "checkInDate #in checkOutDate #in roomType #inout rateType #out rate #return">
    <input messageLabel="In"
          element="tns:checkAvailability" />
    <output messageLabel="Out"
          element="tns:checkAvailabilityResponse" />

  </operation>
. . .
</interface>
. . .
```

Note that the interface operation's name "`checkAvailability`", is the same as the localPart of the input element's QName, "`tns:checkAvailability`". This is one of the requirements of the RPC Style. The name of the operation is used as the name of the method in a language binding, subject to further mapping restrictions specific to the target programming language. In this case, the name of the method would be "`checkAvailability`".

The local children elements of the input element and output element designate the parameters and the return type for a method call. Note that the elements `checkInDate`, `checkOutDate` are input parameters, however the element `roomType` is an in-out parameter, as it appears both as a local element child of both input and output elements. This indicates that the reservation system may change the room type requested based on availability.

The reservation service also returns a rate type for the reservation, such as "rack rate". The return value for the method is designated as the "rate" element.

Based on the value of the `wrpc:signature` attribute, the method signature would be obtained following the order of the parameters. A sample mapping is provided below for the Java(TM) language. This example was created using JAX RPC 1.1 [*JAX RPC 1.1 [p.85]*] for mapping simple types to Java types and designated inout and output parameters by using Holder classes.

Example 4-9. Sample Java(TM) Signature for RPC Style

```
public interface reservationInterface extends Remote{

   double checkAvailability(java.util.calendar checkInDate,
              java.util.calendar checkOutDate,
              StringHolder roomType,
              StringHolder rateType) throws RemoteException;
    . . .
}
```

Programming languages may further specify how faults are mapped to language constructs and their scopes, such as Exceptions, but they are not specific to RPC style.

4.5 MTOM and Attachments Support

Unlike WSDL 1.1 which defines a MIME binding for attachments support, WSDL 2.0 supports MIME attachments via the SOAP Message Transmission Optimization Mechanism (MTOM) [*SOAP MTOM [p.84]*]. This section shows how MTOM may be engaged in the WSDL 2.0 SOAP binding extension.

We will modify the `CheckAvailability` operation of the GreatH Hotel Reservation Service (Example 2-1 [p.7]) to return not only the room rate, but images of the room and the floorplan. This will involve modifying the `checkAvailabilityResponse` data structure to include binary data representing these two images, indicated by the `xs:base64Binary` data type. Here is an example:

Example 4-10. XML Schema with Optimizable Elements

```
. . .
<xs:element name="checkAvailabilityResponse">

  <xs:sequence>

    <xs:element name="rate" type="xs:double"/>

    <xs:element name="photo"
        type="xmime:base64Binary"
        xmime:expectedContentType="image/jpeg image/png" />

    <xs:element name="floorplan"
        xmime:expectedContentType="image/svg">
      <xs:simpleContent>
        <xs:restriction base="xs:base64Binary">
          <xs:attribute ref="xmime:contentType"
                fixed="image/svg" />
```

```
        </xs:restriction>
      </xs:simpleContent>
    </xs:element>

  </xs:sequence>

</xs:element>
. . .
```

Note the use of the `xmime:expectedContentType` and `xmime:contentType` attributes to declare the expected media type of the encoded data and to allow the client to indicate the type at runtime, respectively. These attributes are defined in [*Describing Media Content of Binary Data in XML [p.83]*]. Also note that, when using the WSDL HTTP Binding, an implementation MAY use incoming HTTP Accept headers to choose between alternative media types listed in xmime:expectedContentType.

A `checkAvailabilityResponse` message conforming to this schema might look like this:

Example 4-11. Non-optimized SOAP Message with Embedded Binary Data

```
<soap:Envelope
    xmlns:soap='http://www.w3.org/2003/05/soap-envelope'
    xmlns:xmime='http://www.w3.org/2005/05/xmlmime'>

  <soap:Body>
    <g:checkAvailabilityResponse
        xmlns:g="http://greath.example.com/2004/schemas/resSvc">

      <g:rate>129.95</g:rate>
      <g:photo xmime:contentType='image/png'>/aWKKapGGyQ=</g:photo>
      <g:floorplan xmime:contentType="image/svg">Faa7vROi2VQ=</g:floorplan>

    </g:checkAvailabilityResponse>
  </soap:Body>

</soap:Envelope>
```

While this (non-optimized) message satisfies the schema definition, a service may choose to allow or require that the binary data be sent in an optimized format using the Message Transmission and Optimization Mechanism (MTOM). The use of this feature by the WSDL 2.0 SOAP binding extension is indicated as follows:

Example 4-12. Specifying MTOM in a WSDL 2.0 Binding

```
  . . .
<binding name="reservationSOAPBinding"
    interface="tns:reservationInterface"
    type="http://www.w3.org/2006/01/wsdl/soap"
    wsoap:protocol="http://www.w3.org/2003/05/soap/bindings/HTTP">

 <operation ref="tns:opCheckAvailability"
      wsoap:mep="http://www.w3.org/2003/05/soap/mep/request-response">

   <input name="checkAvailability" />
```

```
    <output name="checkAvailabilityResponse">
      <feature
        uri="http://www.w3.org/2004/08/soap/features/http-optimization"
        required="true" />
    </output>

  </operation>
    . . .
</binding>
. . .
```

The HTTP Message Transmission Optimization (MTOM) feature is engaged using the `feature` element. Note that the attribute `required="true"` on the feature declaration indicates that the message must be encoded using the HTTP Optimization feature. If the attribute were `required="false"` (or this attribute were absent), it would indicate that the use of MTOM is optional for this service: the service accepts either MTOM-encoded messages, or the embedded base64Binary data directly in the SOAP Body, and the client is free to send either form of message.

The example above shows MTOM enabled for a specific message within an operation. Placing the feature declaration as a child of `operation` would require (or enable if `required="false"`) MTOM support for all the messages in that operation. Placing the feature declaration as a child of `binding` would require (or enable if `required="false"`) MTOM support for all the operations in that interface.

5. Advanced Topics III: Miscellaneous

This section covers various topics that may fall outside the scope of WSDL 2.0, but shall provide useful background and best practice guidances that may be useful when authoring a WSDL 2.0 document or implementing the WSDL 2.0 specification.

5.1 Enabling Easy Message Dispatch

It is desirable for a message recipient to have the capability to uniquely identify a message in order to handle it correctly. The capability of identifying a message is typically used for dispatching purposes within an implementation of a web service. Therefore, WSDL authors are recommended to take disambiguating of messages that are defined in a description into consideration when they develop descriptions of their services.

The context in which a Web service may be deployed plays an important role in choosing an appropriate way to disambiguate and identify messages. In a typical deployment, an endpoint address may host a single service that is described by a WSDL service element. In this case, when XSD is used, assigning unique qualified names of global element declarations as inputs within the interface that describes the service would be sufficient to disambiguate the messages that are received. However, when endpoint address hosts multiple services, in essence supports several WSDL descriptions, the desire to disambiguate messages should considered within the context of all the deployed services, not only within a single interface.

As explained in **2.4.4.1 Operation Attributes** [p.31] , when XSD is used as the type system, a few special tokens can be used for the `element` attributes. Uniquely identifying a message may become very difficult when:

- any of these input elements within an interface has a value of “#any”; or
- more than one of these input elements (see below) has a value of “#none”; or
- the qualified names of the global element declarations that are specified as input elements are NOT unique when considered together.

If any of the three cases above arise, then one of the following two alternatives can be used within the context of a single WSDL service by WSDL authors:

- *Feature.* The service or the interface element contains a Feature element declaration, having a required attribute with a value of true. The feature unambiguously identifies the mechanism that a message sender is required to support in order to enable the message recipient to unambiguously determine the message received.
- *Extension.* The interface element contains an extension element (i.e., an element that is not in the http://www.w3.org/2006/01/wsdl namespace), having a wsdl:required attribute with a value of "true". The extension element unambiguously identifies the mechanism that a message sender is required to support in order to enable the message recipient to unambiguously determine the message received.

In addition, WS-Addressing [WS-Addressing] specification already provides a disambiguation mechanism. It defines a required [action] property whose value is always present in a message delivery. The value of the action property can be used to disambiguate the message by the receiver and there is a well defined way to associate actions to messages in WS-Addressing specifications. Further, WS-Addressing also provides an appropriate default action value that identifies each message uniquely.

5.2 Web Service Versioning

A WSDL 2.0 document describes a set of messages that a Web service may send and receive. In essence, it describes a language for interacting with that service. However it is possible for a Web service to exchange other messages beyond those described in a particular WSDL 2.0 document. Often this circumstance occurs following an evolution of the client and/or service, and thus an evolution of the interaction language.

How best to manage the evolution (versioning) of Web based systems is, at the time of writing, the subject of a wide ranging debate. However, there are three activities within the W3C that are directly relevant to versioning of Web services description:

- The Technical Architecture Group (TAG) has published guidance on the extensibility and versioning of data formats in its Web Architecture document [*Web Architecture [p.83]*]. There is also a more wide ranging draft finding on Versioning and Extensibility [*W3C TAG Finding: Versioning [p.85]*]. Both of these works build upon the technical note on Web Architecture: Extensible Languages [*WebArch: Extensible Languages [p.85]*].

- The XML Schema Working Group is collecting a series of use cases for schema versioning as a part of the Schema 1.1 activity. See XML Schema Versioning Use Cases [*XML Schema: Versioning Use-Cases [p.85]*].
- The Semantic Web Best Practices and Deployments Working Group is examining how vocabularies may evolve. See [*SW VocabManagementNote [p.85]*]

Editorial note: PaulD	20050706
This section may be subject to change dependent upon the outcome of the WSDL Last Call Issue LC124, which discusses support compatible evolution of messages described using XML Schema 1.0.	

While incomplete, these activities all agree in one important respect: that versioning is difficult, but you should anticipate and plan for change.

The draft finding on Versioning and Extensibility details two key approaches to versioning:

- compatible evolution; and
- big bang.

5.2.1 Compatible Evolution

In *compatible evolution*, designers are expected to limit changes to those that are either backward or forward compatible, or both:

Backward compatible

The receiver behaves correctly if it receives a message in an *older* version of the interaction language.

Forward compatible

The receiver behaves correctly if it receives a message in a *newer* version of the interaction language.

Since Web services and their clients both send and receive messages, these concepts can apply to both parties. However, since WSDL 2.0 is service-centric, we will focus on the case of service evolution.

There are three critical areas in which a service described in WSDL 2.0 my evolve:

- The service now also supports additional binding. In compatible evolution, this should be a safe addition, given that adding a new binding should not impact any existing interactions using another transport.
- An interface supports new operations. Again, in compatible evolution this is usually safe, given that adding an additional operation to an abstract interface should not impact any existing interactions.

- The message bodies may include additional data. How the message contents may change within a description depends to a large extent upon the type system being used to describe the message contents. RelaxNG [*RELAX NG [p.85]*] has good support for describing vocabularies that ignore unknown XML, as does OWL/RDF. XML Schema 1.0 has limited support for extending the description of a message via the `xs:any` and `xs:anyAttribute` constructs. XML Schema 1.1 has been chartered to provide "changes necessary to provide better support for versioning of schemas", and it is anticipated that this may include improved support for more "open content" and therefore better support for compatible evolution of messages.

- The protocol used to exchange messages may provide mechanisms for exchanging data outside of the message body. In the case of SOAP, the WSDL 2.0 binding provides the ability to describe application data to be exchanged as headers. The SOAP processing model has a very good extensibility model with unknown headers being ignored by a receiver by default. There is also a mechanism whereby headers which are required as a part of an incompatible change may be marked with a 'mustUnderstand' flag. Passing additional items as headers may be the only way to compatibly evolve messages with fixed bodies.

5.2.2 Big Bang

The *big bang* approach to versioning is the simplest to currently represent in WSDL 2.0. In this approach, any change to a WSDL 2.0 document implies a change to the document's namespace, a change to the interface implies a new interface namespace and a change to the message contents is communicated using a new message namespace. This approach has particular benefits where an agent may quickly tell if a service has changed by simply comparing the namespace value.

5.2.3 Evolving a Service

Compatible changes are far more easily managed than incompatible ones:

- With a compatible change the service need only support the latest version of a service. A client may continue to use a service adjusting to new version of the interface description at a time of its choosing.

- With an incompatible change, the client receives a new version of the interface description and is expected to adjust to the new interface before old interface is terminated. Either the service will need to continue to support both versions of the interface during the hand over period, or the service and the clients are coordinated to change at the same time. An alternative is for the client to continue until it encounters an error, at which point it uses the new version of the interface.

5.2.4 Combined Approaches

It is feasible to combine the "compatible evolution" and "big bang" approaches in a variety of different ways. For example, the namespace could be changed when message descriptions are changed, but the namespace could stay the same when new operations are added.

While the big bang approach is currently the easiest to implement in WSDL 2.0, it can lead to a large number of cloned interfaces that become difficult to manage, thus making the compatible approach preferable to many for widely distributed systems. In the end, the choice of a versioning strategy for Web services described in WSDL 2.0 is left as an exercise to the reader.

5.2.5 Examples of Versioning and Extending a Service

5.2.5.1 Additional Optional Elements Added in Content

The following example demonstrates how content may be extended with additional content. The reservation service is changed to a newer version that can accept an optional number of guests parameter. The service provider wants existing clients to continue to be able to use the service. The author adds the element into the schema as an optional element.

Example 5-1. XML Schema with Optional Elements

```
<xs:complexType name="tCheckAvailability">
    <xs:sequence>
      <xs:element  name="checkInDate" type="xs:date"/>
      <xs:element  name="checkOutDate" type="xs:date"/>
      <xs:element  name="roomType" type="xs:string"/>
      <xs:element  name="numberOfGuests" type="xs:integer" minOccurs="0"/>
      <xs:any namespace="##other" processContents="lax"/>
    </xs:sequence>
</xs:complexType>
```

The author has the choice of keeping the same namespace or using a different namespace for the additional content and the existing content. In this scenario, it is a compatible change and the author decides to keep the same namespace. This allows existing clients to interact with a new service, and it allows newer clients to interact with older services.

5.2.5.2 Additional Optional Elements Added to a Header

Another option is to add the extension as a header block. This is accomplished by defining an element for the extension and adding a header element that references the element into the binding operation as child of the input.

Example 5-2. Additional optional elements added to a SOAP header

```
<xs:element name="NumberOfGuests" type="tNumberOfGuests"/>
<xs:complexType name="tNumberOfGuests">
    <xs:sequence>
        <xs:element  name="numberOfGuests" type="xs:integer" minOccurs="0"/>
    </xs:sequence>
</xs:complexType>

<binding name="reservationSOAPBinding"
    interface="tns:reservationInterface"
    type="http://www.w3.org/2006/01/wsdl/soap"
    wsoap:protocol="http://www.w3.org/2003/05/soap/bindings/HTTP">

    <operation ref="tns:opCheckAvailability">
```

```
        <input>
          <wsoap:header element="tns:NumberOfGuests"/>
        </input>
    </operation>
...
</binding>
```

It is also possible for the header to be marked with soap:mustUnderstand set to true. The HTTP Binding has similar functionality though without a mustUnderstand attribute.

5.2.5.3 Additional Mandatory Elements in Content

This following example demonstrates an extension with additional content. The reservation service requires a number of guests parameter. The service provider wants existing clients to be unable to use the service. The author adds the element into the schema as a mandatory element.

Example 5-3. Additional Mandatory Elements in Content

```
<xs:complexType name="tCheckAvailabilityV2">
    <xs:sequence>
    <xs:element  name="checkInDate" type="xs:date"/>
    <xs:element  name="checkOutDate" type="xs:date"/>
    <xs:element  name="roomType" type="xs:string"/>
    <xs:element  name="numberOfGuests" type="xs:integer"/>
    <xs:any namespace="##other" processContents="lax"/>
    </xs:sequence>
</xs:complexType>
```

The author has the choice of keeping the same namespace or using a different namespace for the additional content and the existing content. In this scenario, it is an incompatible change and the author decides to use a new name but the same namespace. This type is then used in the interface operation, and then binding and service endpoints.

5.2.5.4 Additional Optional Operation Added to Interface

Section **2.4.2 Interface Inheritance** [p.29] shows another type of versioning or extension, where the reservationInterface extends the MessageLogInterface. By definition of interface inheritance, a client that understands just the MessageLogInterface will continue to work with the reservationInterface, that it is backwards compatible.

5.2.5.5 Additional Mandatory Operation Added to Interface

Often mandatory operations are added to an interface. The Hotel service decides to add an operation to the reservation service which is a confirmation. The Hotel service requires that all clients upgrade to the new interface to use the service. They have a variety of options for indicating that the old interface is deprecated.

By the definition of interface inheritance, they cannot use interface inheritance for defining the extension.

Example 5-4. Additional Mandatory Operation Added to the Interface

```
<interface name="reservationWithConfirmation" extends="cc:creditCardFaults">
    ...
    <operation name="makeReservation">
        <input messageLabel="In" element="ghns:makeReservation" />
        <output messageLabel="Out" element="ghns:makeReservationResponse" />
        <outfault ref="invalidDataFault" messageLabel="Out" />
        <outfault ref="cc:cancelledCreditCard" messageLabel="Out" />
        <outfault ref="cc:expiredCreditCard" messageLabel="Out" />
        <outfault ref="cc:invalidCreditCardNumber" messageLabel="Out" />
        <outfault ref="cc:invalidExpirationDate" messageLabel="Out" />
    </operation>
    <operation name="confirmReservation">
        <input messageLabel="In" element="ghns:makeReservationResponse" />
        <output messageLabel="Out" element="ghns:confirmReservationResponse" />
        <outfault ref="expiredReservation" messageLabel="Out" />
    </operation>
</interface>
```

This interface cannot be bound and deployed at the existing URI and indicate incompatibility, as the service will still accept the makeReservation request. Changing the name of the interface from reservation to reservationWithConfirmation or changing the name of the operation from makeReservation to makeReservationV2 does not affect the messages that are exchanged. Thus it can't be used as a mechanism for indicating incompatibility. To indicate incompatibility, a change must be made to something that appears in the message. For a SOAP over HTTP request, the list is roughly the URI, the SOAP Action HTTP Header, or the Message content.

5.2.5.6 Indicating Incompatibility by Changing the Endpoint URI

To indicate incompatibility, the URI of the Hotel Endpoint can be changed and messages send to the old Endpoint return a Fault.

5.2.5.7 Indicating Incompatibility by Changing the SOAP Action

The SOAP Action can be set for the makeReservation request, and making it different than the earlier version should indicate incompatibility.

Example 5-5. Indicating Incompatibility by changing the SOAP Action

```
<binding name="reservationSOAPBinding"
        interface="tns:reservationInterface"
        type="http://www.w3.org/2006/01/wsdl/soap"
        wsoap:protocol="http://www.w3.org/2003/05/soap/bindings/HTTP">
    <operation ref="tns:makeReservation"
        wsoap:action="tns:makeReservationV2"/>
    . . .
```

Note that this mechanism is applicable on a per-binding basis. The SOAP HTTP Binding provides for setting Action, but other bindings may not provide such a facility.

5.2.5.8 Indicating Incompatibility by Changing the Element Content

The namespace or name of the makeReservation element can be changed, and then the interface and bindings changed. To indicate incompatibility, requests using the old makeReservation QName should probably return a fault. The new interface, with a changed makeReservation, is:

Example 5-6. Indicating incompatibility by changing the element content

```
<xs:element name="ghns2:makeReservation" type="ghns:tmakeReservation"/>

<interface . . .>

   <operation name="makeReservation">

       <input messageLabel="In" element="ghns2:makeReservation" />

</interface>
```

The binding and service endpoints require no change.

Finally, the service could also provide an interface for ghns:makeReservation that only returns a fault.

5.3 Describing Web Service Messages That Refer to Other Web Services

Hyperlinking is one of the defining characteristics of the Web. The ability to navigate from one Web page to another is extremely useful. It is therefore natural to apply this capability to Web services. This section describes references to endpoints and services, which are the Web service analogs of document hyperlinks.

A *reference to an endpoint* is an element or attribute that contains the address of a Web service endpoint. A *reference to a service* is an element or attribute that contains one or more references to the endpoints of a service. If the interface or binding that the service or endpoint implements is known at description time, it may be useful to add this information to the WSDL 2.0 document that describes the Web service. This is accomplished by using the `wsdlx:interface` or `wsdlx:binding` attribute to annotate the XML Schema component that defines the message.

One may wonder, from a Web architectural point of view, why anything more than a URI would be needed to reference a Web service. Indeed, a reference to a service does make use of one or more URIs to indicate the endpoint addresses of a service. However, it may also include additional metadata about that service, such as the WSDL 2.0 interface and binding that the service supports.

References to services and endpoints will be illustrated by expanding the GreatH example already discussed.

5.3.1 The Reservation Details Web Service

When designing a Web application it is natural to give each important concept a URI. In the GreatH hotel reservation system, the important concepts are reservations, so we begin our design by assigning a URI to each reservation. Since each reservation has a unique confirmation number, e.g OMX736, we create a URI for each reservation by appending the confirmation number to a base URI, e.g. http://greath.example.com/2004/reservation/OMX736. This URI will be the endpoint address for a Reservation Details Web service that can retrieve and update the state of a reservation. Example 5-7 [p.70] shows the format of the reservation detail.

Example 5-7. Detail for Reservation OMX736

```
<?xml version="1.0" encoding="UTF-8"?>
<reservationDetails
        xmlns="http://greath.example.com/2004/schemas/reservationDetails">

        <confirmationNumber>OMX736</confirmationNumber>
        <checkInDate>2005-06-01</checkInDate>
        <checkOutDate>2005-06-03</checkOutDate>
        <roomType>single</roomType>
        <smoking>false</smoking>

</reservationDetails>
```

The Reservation Details Web service provides operations for retrieving and updating the detail for a reservation. Example 5-8 [p.70] shows the description for this Web service. Note that there is no `service` element in this description since the set of reservations is dynamic. Instead, the endpoints for the reservations will be returned by querying the Reservation List Web service.

Example 5-8. The Reservation Details Web Service Description: reservationDetails.wsdl

```
<?xml version="1.0" encoding="utf-8" ?>
<description
        xmlns="http://www.w3.org/2006/01/wsdl"
        targetNamespace="http://greath.example.com/2004/services/reservationDetails"
        xmlns:tns="http://greath.example.com/2004/services/reservationDetails"
        xmlns:wdetails="http://greath.example.com/2004/schemas/reservationDetails"
        xmlns:wsoap="http://www.w3.org/2006/01/wsdl/soap"
        xmlns:xs="http://www.w3.org/2001/XMLSchema">

        <documentation>
                This document describes the GreatH Reservation Details Web
                services. Use these services to retrieve or update reservation
                details. Each reservation has its own service and endpoint. To
                obtain the reference for a reservation service, make a request to
                the GreatH Reservation List Web service. See
                reservationList.wsdl for a description of the Reservation List
                Web service.
        </documentation>

        <types>
                <xs:import
                        namespace="http://greath.example.com/2004/schemas/reservationDetails"
                        schemaLocation="reservationDetails.xsd" />
        </types>

        <interface name="reservationDetailsInterface">

                <operation name="retrieve"
                        pattern="http://www.w3.org/2006/01/wsdl/in-out">
                        <input messageLabel="In" element="#none" />
                        <output messageLabel="Out"
```

```
                        element="wdetails:reservationDetails" />
        </operation>

        <operation name="update"
                pattern="http://www.w3.org/2006/01/wsdl/in-out">
                <input messageLabel="In"
                        element="wdetails:reservationDetails" />
                <output messageLabel="Out"
                        element="wdetails:reservationDetails" />
        </operation>

    </interface>

    <binding name="reservationDetailsSOAPBinding"
            interface="tns:reservationDetailsInterface"
            type="http://www.w3.org/2006/01/wsdl/soap"
            wsoap:protocol="http://www.w3.org/2003/05/soap/bindings/HTTP">

            <operation ref="tns:retrieve"
                    wsoap:mep="http://www.w3.org/2003/05/soap/mep/request-response" />

            <operation ref="tns:update"
                    wsoap:mep="http://www.w3.org/2003/05/soap/mep/request-response" />

    </binding>

</description>
```

Example 5-9 [p.71] shows the XML schema elements that are used in this Web service.

Example 5-9. The Reservation Details Web Service XML Schema: reservationDetails.xsd

```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema"
        elementFormDefault="qualified"
        targetNamespace="http://greath.example.com/2004/schemas/reservationDetails"
        xmlns:tns="http://greath.example.com/2004/schemas/reservationDetails"
        xmlns:wdetails="http://greath.example.com/2004/services/reservationDetails"
        xmlns:wsdli="http://www.w3.org/2006/01/wsdl-instance"
        xmlns:wsdlx="http://www.w3.org/2006/01/wsdl-extensions"
        wsdli:wsdlLocation="http://greath.example.com/2004/services/reservationDetails reservationDetails.wsdl">

        <element name="confirmationNumber" type="string" />

        <element name="checkInDate" type="date" />

        <element name="checkOutDate" type="date" />

        <element name="reservationDetails">
                <complexType>
                        <sequence>
                                <element ref="tns:confirmationNumber" />
                                <element ref="tns:checkInDate" />
                                <element ref="tns:checkOutDate" />
                                <element name="roomType" type="string" />
                                <element name="smoking" type="boolean" />
                        </sequence>
                </complexType>
        </element>

        <simpleType name="reservationDetailsSOAPEndpointType" wsdlx:binding="wdetails:reservationDetailsSOAPBinding">
                <restriction base="anyURI"/>
        </simpleType>

        <element name="reservationDetailsSOAPEndpoint" type="tns:reservationDetailsSOAPEndpointType" />

        <element name="reservationDetailsService">
```

```
            <annotation>
                <documentation>
                    This element contains references to the Reservation
                    Details Web Service endpoints for this reservation.
                </documentation>
            </annotation>
            <complexType>
                <sequence>
                    <element name="soap" type="tns:reservationDetailsSOAPEndpointType"/>
                    <element name="secure-soap" type="tns:reservationDetailsSOAPEndpointType"/>
                </sequence>
            </complexType>
        </element>

</schema>
```

This XML schema contains the usual definitions for the elements that appear in the messages of the Web service. For example, the `reservationDetails` element is used in the messages of the `retrieve` and `update` operations. In addition, the schema defines the simple type `reservationDetailsSOAPEndpointType` which is based on `xs:anyURI` and has the annotation `wsdlx:binding = "wdetails:reservationDetailsSOAPBinding"` which means that the URI is the address of a Reservation Details Web service endpoint that implements the `wdetails:reservationDetailsSOAPBinding` binding. Note that the `wsdli:wsdlLocation` attribute is used to define the location of the WSDL 2.0 document that defines the `wdetails:reservationDetailsSOAPBinding` binding. This annotated simple type is used to define the `reservationDetailsSOAPEndpoint` element which will be used in the Reservation List service.

5.3.2 The Reservation List Web Service

Since the set of reservations changes as reservations are made and cancelled, the Reservation Detail endpoints are not described in a fixed WSDL 2.0 document. Instead they are returned as references to endpoints in response to requests made on a Reservation List Web service. The endpoint address for the Reservation List service will be http://greath.example.com/2004/reservationList.

Example 5-10 [p.72] shows the format of the response from the Reservation List service.

Example 5-10. Response from the Reservation List Web Service

```
<?xml version="1.0" encoding="UTF-8"?>
<reservationList
        xmlns="http://greath.example.com/2004/schemas/reservationList"
        xmlns:details="http://greath.example.com/2004/schemas/reservationDetails">

        <reservation>
                <details:confirmationNumber>HSG635</details:confirmationNumber>
                <details:checkInDate>2005-06-27</details:checkInDate>
                <details:checkOutDate>2005-06-28</details:checkOutDate>
                <details:reservationDetailsSOAPEndpoint>
                        http://greath.example.com/2004/reservation/HSG635
                </details:reservationDetailsSOAPEndpoint>
        </reservation>

        <reservation>
                <details:confirmationNumber>OMX736</details:confirmationNumber>
                <details:checkInDate>2005-06-01</details:checkInDate>
```

```
        <details:checkOutDate>2005-06-03</details:checkOutDate>
        <details:reservationDetailsSOAPEndpoint>
                http://greath.example.com/2004/reservation/OMX736
        </details:reservationDetailsSOAPEndpoint>
    </reservation>

    <reservation>
        <details:confirmationNumber>WUH663</details:confirmationNumber>
        <details:checkInDate>2005-06-11</details:checkInDate>
        <details:checkOutDate>2005-06-15</details:checkOutDate>
        <details:reservationDetailsSOAPEndpoint>
                http://greath.example.com/2004/reservation/WUH663
        </details:reservationDetailsSOAPEndpoint>
    </reservation>

</reservationList>
```

Here, the `<details:reservationDetailsSOAPEndpoint>` elements contain references to the Reservation Details Web service endpoints for the reservations HSG635, OMX736, and WUH663.

Example 5-11 [p.73] shows the description of the Reservation List Web service. Note that it contains operations to retrieve the entire list and to query for a list of reservations by confirmation number, check-in date, and check-out date. In each case, the operation returns a list of reservations.

Example 5-11. The Reservation List Web Service Description: reservationList.wsdl

```
<?xml version="1.0" encoding="utf-8" ?>
<description
        xmlns="http://www.w3.org/2006/01/wsdl"
        targetNamespace="http://greath.example.com/2004/services/reservationList"
        xmlns:tns="http://greath.example.com/2004/services/reservationList"
        xmlns:details="http://greath.example.com/2004/schemas/reservationDetails"
        xmlns:list="http://greath.example.com/2004/schemas/reservationList"
        xmlns:wsoap="http://www.w3.org/2006/01/wsdl/soap"
        xmlns:xs="http://www.w3.org/2001/XMLSchema">

        <documentation>
                This document describes the GreatH Reservation List Web
                services. Use this service to retrieve lists of reservations
                based on a variety of search criteria.
        </documentation>

        <types>
                <xs:import
                        namespace="http://greath.example.com/2004/schemas/reservationDetails"
                        schemaLocation="reservationDetails.xsd" />
                <xs:import
                        namespace="http://greath.example.com/2004/schemas/reservationList"
                        schemaLocation="reservationList.xsd" />
        </types>

        <interface name="reservationListInterface">

                <operation name="retrieve"
                        pattern="http://www.w3.org/2006/01/wsdl/in-out">
                        <input messageLabel="In" element="#none" />
                        <output messageLabel="Out" element="list:reservationList" />
                </operation>
```

```
        <operation name="retrieveByConfirmationNumber"
                pattern="http://www.w3.org/2006/01/wsdl/in-out">
                <input messageLabel="In"
                        element="details:confirmationNumber" />
                <output messageLabel="Out" element="list:reservationList" />
        </operation>

        <operation name="retrieveByCheckInDate"
                pattern="http://www.w3.org/2006/01/wsdl/in-out">
                <input messageLabel="In" element="details:checkInDate" />
                <output messageLabel="Out" element="list:reservationList" />
        </operation>

        <operation name="retrieveByCheckOutDate"
                pattern="http://www.w3.org/2006/01/wsdl/in-out">
                <input messageLabel="In" element="details:checkOutDate" />
                <output messageLabel="Out" element="list:reservationList" />
        </operation>

</interface>

<binding name="reservationListSOAPBinding"
        interface="tns:reservationListInterface"
        type="http://www.w3.org/2006/01/wsdl/soap"
        wsoap:protocol="http://www.w3.org/2003/05/soap/bindings/HTTP">

        <operation ref="tns:retrieve"
                wsoap:mep="http://www.w3.org/2003/05/soap/mep/request-response" />

        <operation ref="tns:retrieveByConfirmationNumber"
                wsoap:mep="http://www.w3.org/2003/05/soap/mep/request-response" />

        <operation ref="tns:retrieveByCheckInDate"
                wsoap:mep="http://www.w3.org/2003/05/soap/mep/request-response" />

        <operation ref="tns:retrieveByCheckOutDate"
                wsoap:mep="http://www.w3.org/2003/05/soap/mep/request-response" />

</binding>

<service name="reservationListService"
        interface="tns:reservationListInterface">

        <endpoint name="reservationListEndpoint"
                binding="tns:reservationListSOAPBinding"
                address="http://greath.example.com/2004/reservationList" />

</service>

</description>
```

Example 5-12 [p.74] shows the schema for the messages used in the Reservation List Web service.

Example 5-12. The Reservation List Schema: reservationList.xsd

```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema"
        elementFormDefault="qualified"
        targetNamespace="http://greath.example.com/2004/schemas/reservationList"
```

```
        xmlns:tns="http://greath.example.com/2004/schemas/reservationList"
        xmlns:details="http://greath.example.com/2004/schemas/reservationDetails"
        xmlns:wsdli="http://www.w3.org/2006/01/wsdl-instance">

        <import
                namespace="http://www.w3.org/2006/01/wsdl-instance" />

        <import
                namespace="http://greath.example.com/2004/schemas/reservationDetails"
                schemaLocation="reservationDetails.xsd" />

        <element name="reservation">
                <annotation>
                        <documentation>
                                A reservation contains the confirmation number, check-in
                                and check-out dates, and a reference to a Reservation
                                Details Web service.
                        </documentation>
                </annotation>
                <complexType>
                        <sequence>
                                <element ref="details:confirmationNumber" />
                                <element ref="details:checkInDate" />
                                <element ref="details:checkOutDate" />
                                <element ref="details:reservationDetailsSOAPEndpoint" />
                        </sequence>
                </complexType>
        </element>

        <element name="reservationList">
                <annotation>
                        <documentation>
                                A reservation list contains a sequence of zero or more
                                reservations.
                        </documentation>
                </annotation>
                <complexType>
                        <sequence>
                                <element ref="tns:reservation" minOccurs="0"
                                        maxOccurs="unbounded">
                                </element>
                        </sequence>
                        <attribute ref="wsdli:wsdlLocation" />
                </complexType>
        </element>

</schema>
```

In the preceding example, there was a single endpoint associated with each Reservation Detail Web service. Suppose GreatH hotel decided to provide a second, secure endpoint. In this case, references to services would be used to collect together the endpoints for each reservation. The reservationDetails.xsd schema defines the `reservationDetailsService` element for this purpose. It contains the nested elements `soap` and `secure-soap` which are each of type `reservationDetailsSOAPEndpointType` and therefore contain the address of an endpoint that implements the `wdetails:reservationDetailsSOAPBinding` binding.

Example 5-13 [p.76] shows an example of a message that contains a reference to the service for reservation HGS635. Note that the service contains two endpoints, one of which provides secure access to the Reservation Details Web service.

Example 5-13. A Reference to the Reservation Details Web Service

```
<?xml version="1.0" encoding="UTF-8"?>
<details:reservationDetailsService
        xmlns:details="http://greath.example.com/2004/schemas/reservationDetails"

        <details:soap>
                http://greath.example.com/2004/reservation/HSG635
        </details:soap>

        <details:secure-soap>
                https://greath.example.com/2004/reservation/HSG635
        </details:secure-soap>

</details:reservationDetailsService>
```

5.3.3 Reservation Details Web Service Using HTTP Transfer

This section presents a variation on the example in **5.3.1 The Reservation Details Web Service** [p.69] . It illustrates the use of HTTP transfer operations, GET and PUT, to retrieve and update GreatH hotel reservation details using the Representational State Transfer (REST) architectural style described by Roy Fielding [*REST [p.85]*] . REST is a distillation of the architectural properties that Dr. Fielding identified as being vital to the Web's robustness and enormous scalability.

Since each reservation in our example will have a distinct URI, the Reservation Details Web service can be offered using HTTP GET and HTTP PUT. The binding would be modified as follows:

Example 5-14. Reservation Details Web Service Using HTTP Transfer

```
. . .
<binding name="reservationDetailsHTTPBinding"
        type="http://www.w3.org/2006/01/wsdl/http"
    interface="tns:reservationDetailsInterface" >

        <operation ref="tns:retrieve"
            whttp:method="GET" />

        <operation ref="tns:update"
            whttp:method="PUT" />

</binding>
. . .
```

As with the example in **5.3.1 The Reservation Details Web Service** [p.69] , service and endpoint elements are not provided because the Reservation List Web service provides the endpoints.

5.3.4 Reservation List Web Service Using HTTP GET

This section continues the REST-style example of **5.3.3 Reservation Details Web Service Using HTTP Transfer** [p.76] by modifying the example of **5.3.2 The Reservation List Web Service** [p.72] to use HTTP GET.

The SOAP version of the Reservation List Web service above offers four different search operations. These can also be expressed as various parameters in a URI used by HTTP GET:

Example 5-15. Reservation List Web Service Using HTTP GET

```
. . .
<binding name="reservationListHTTPBinding"
        type="http://www.w3.org/2006/01/wsdl/http"
    interface="tns:reservationListInterface"
    whttp:methodDefault="GET">

  <operation ref="tns:retrieve"
      whttp:location="" />

  <operation ref="tns:retrieveByConfirmationNumber"
      whttp:location="reservationList/ConfirmationNumber/{confirmationNumber/}" />

  <operation ref="tns:retrieveByCheckInDate"
      whttp:location="reservationList/CheckInDate/{checkInDate/}" />

  <operation ref="tns:retrieveByCheckOutDate"
      whttp:location="reservationList/CheckOutDate/{checkOutDate/}" />
</binding>
. . .
<service . . . >

  <endpoint name="reservationListEndpoint"
    binding="tns:reservationListHTTPBinding"
    address="http://greath.example.com/2004/reservationList" />
. . .
</service>
. . .
```

A retrieval by Confirmation Number URI would look like: `http://greath.example.com/2004/reservationList/ConfirmationNumber/HSG635` .

Alternatively, a single query type may be provided. This query type is a sequence of optional items. Any items in the sequence are serialized into the URI query string. A query sequence for any of ConfirmationNumber, checkInDate, checkOutDate would look like this:

Example 5-16. Query Sequence Using a Single Query Type

```
<element name="reservationQuery">
  <annotation>
    <documentation>
      A reservation contains the confirmation number, check-in
      and check-out dates, and a reference to a Reservation
```

```
      Details Web service.
    </documentation>
  </annotation>
  <complexType>
    <sequence>
      <element ref="details:confirmationNumber" minOccurs="0"/>
      <element ref="details:checkInDate" minOccurs="0"/>/>
      <element ref="details:checkOutDate" minOccurs="0"/>/>
    </sequence>
    </sequence>
  </complexType>
</element>
```

The WSDL 2.0 service that offers this type serialized as a parameter would look like this:

Example 5-17. WSDL 2.0 for Using a Single Query Type

```
. . .
<interface name="reservationListInterfaceWithQuery">

  <operation name="retrieveByReservationQuery"
              pattern="http://www.w3.org/2006/01/wsdl/in-out">
    <input messageLabel="In"
        element="details:ReservationQuery" />
    <output messageLabel="Out"
        element="list:reservationList" />
  </operation>

</interface>

<binding name="reservationListQueryHTTPBinding"
        type="http://www.w3.org/2006/01/wsdl/http"
    interface="tns:reservationListInterfaceWithQuery"
    whttp:methodDefault="GET">

  <operation ref="tns:retrieveByReservationQuery"
      whttp:location="reservationList/{ReservationQuery}}" />

</binding>

. . .
  <endpoint name="reservationListEndpoint"
      binding="tns:reservationListHTTPBinding"
      address="http://greath.example.com/2004/reservationList" />
. . .
```

Various URIs would be: `http://greath.example.com/2004/reservationList/ReservationQuery?confirmationNumber=HSG635` `http://greath.example.com/2004/reservationList/ReservationQuery?checkInDate=06-06-05`.

It is important to observe that using the URI serialization can result in very flexible queries and few operations. The previous discrete SOAP operations are collapsed into one "parameterized" operation.

5.4 Multiple Interfaces for the Same Service

Suppose a Web service wishes to expose two different interfaces: a customer interface for its regular users, and a management interface for its operator. A `wsdl:service` specifies only one wsdl:interface, so to achieve the desired effect the service provider would somehow need to indicate a relationship between two services. How can a service provider indicate a relationship between services? Potential strategies include:

- **Declare both interfaces in the same wsdl:description element.** Although WSDL 2.0 does not ascribe any particular significance to the fact that two wsdl:services are declared within the same wsdl:description, an application or toolkit could interpret this to mean that they are related in some way.
- **Declare both interfaces in the same wsdl:targetNamespace.** Again, although WSDL 2.0 does not ascribe any particular significance to the fact that two wsdl:services are declared within the same wsdl:targetNamespace, an application or toolkit could interpret this to mean that they are related in some way.
- **Add an extension to WSDL 2.0** that links together all services that are related in this way. WSDL 2.0's open content model permits extension elements from other namespaces to appear in a WSDL 2.0 document.
- **Declare them in completely separate WSDL 2.0 documents, but use the same endpoint address for both.** I.e., declare a `wsdl:interface` and `wsdl:service` for the customer interface in one WSDL 2.0 document, and a `wsdl:interface` and `wsdl:service` for the management interface in a different WSDL 2.0 document, but use the same endpoint address for both. (By "different WSDL 2.0 document" we mean that both documents are never included or imported into the same WSDL 2.0 descriptions component.) Although this approach may work in some circumstances, it means that the same endpoint address would be used for two different purposes, which is apt to cause confusion or ambiguity. Furthermore, it is contrary to the Web architectural principle that different URIs should be used to identify different Web resources. (See the Web Architecture [*Web Architecture [p.83]*] section on URI collision.)
- **Use inheritance to combine the customer interface and management interface** into a single, larger wsdl:interface. Of course, this reduces modularity and means that the management interface becomes exposed to the customers, which is not good.

Bear in mind that since the above strategies step outside of the WSDL 2.0 language specifies (and are therefore neither endorsed nor forbidden by the WSDL 2.0 specification) the WSDL 2.0 specification cannot define or standardize their semantics.

The desire to express relationships between services is also relevant to Web service versioning, discussed next.

5.5 Mapping to RDF and Semantic Web

Editorial note: KevinL	20050429
This section might be removed - pending on the availability of the RDF mapping note.	

WSDL 2.0 is a language designed primarily with XML syntax. While XML is almost universally understood, it has several issues:

- The ability to compose two XML documents into one depends on the languages of those documents. WSDL 2.0 does not permit Web service descriptions in different targetNamespaces to be merged into a single (physical) XML document.
- The ability to extend XML languages with other XML languages depends on the languages again. WSDL 2.0 is extremely extensible, but the meaning of every single extension in WSDL 2.0 must be defined explicitly. Putting a piece of XMI (XML format for UML) into a WSDL 2.0 document may have different meaning from putting it into an XHTML document. Therefore XML-based extensibility has very high cost if many languages are involved.
- Similarly, extending another XML language with pieces of WSDL 2.0, while possible, has to be defined for all the possible destinations. Putting a WSDL 2.0 interface element into a UDDI registry may mean a different thing from putting that interface element into an XHTML document.
- Finally, the meaning of a portion of a WSDL 2.0 document is not defined by the WSDL 2.0 specification. While an interface element could form a single XML document, it is not a WSDL 2.0 document, so its meaning is largely undefined.

Applications that require such levels of composability (or decomposability) are increasingly being based on RDF [*RDF [p.85]*], a graph-based knowledge representation language, and Web Ontology Language (OWL) [*OWL [p.85]*], which can be thought of as an advanced schema language for RDF. Effectively, a WSDL 2.0 document represented in RDF can be more easily extended with arbitrary RDF assertions and the WSDL 2.0 information can be more easily associated with arbitrary other knowledge.

5.5.1 RDF Representation of WSDL 2.0

WSDL 2.0: Mapping to RDF [*WSDL 2.0 RDF Mapping [p.83]*] describes how WSDL 2.0 constructs can be expressed in RDF using classes of resources (described with an ontology expressed in OWL) and assertions over individual resources. As RDF represents knowledge using resources and relationships between them, we need to turn WSDL 2.0 concepts into this model. This is done as follows.

1. First, all components in WSDL 2.0 (like Interfaces, Operations, Bindings, Services, Endpoints etc., including extensions) are turned into resources identified with the appropriate URIs created according to Appendix C IRI-References for WSDL 2.0 Components of [*WSDL 2.0 Core [p.83]*].
2. Further, things are represented as resources:

   1. Element declarations gathered from XML Schema (or similarly, other components from other type systems)
   2. Message content models
   3. Message exchange patterns (the URI identifying the MEP is the URI of the resource)
   4. Operation styles (similarly to MEPs, the URI of an operation style is the URI of the resource)
3. All the resources above are given the appropriate types using rdf:type statements (an interface will belong to the class Interface and an operation within an interface will belong to the class InterfaceOperation, for example).
4. All relationships in WSDL 2.0 (like an Operation belonging to an Interface and having a given operation style) are turned into RDF statements using appropriate properties, such as `operation` and `operationStyle`.

5.6 Notes on URIs

5.6.1 XML Namespaces and Schema Locations

It is a common misperception to equate either the target namespace of an XML Schema or the value of the `xmlns` attribute in XML instances with the location of the corresponding schema. Even though namespaces are URIs, and URIs may be locations, and it may be possible to retrieve a schema from such a location, this does not mean that the retrieved schema is the *only* schema that is associated with that namespace. There can be multiple schemas associated with a particular namespace, and it is up to a processor of XML to determine which one to use in a particular processing context. The WSDL 2.0 specification provides the processing context here via the `import` mechanism, which is based on XML Schema's term for the similar concept.

5.6.2 Relative URIs

Throughout this document there are fully qualified URIs used in WSDL 2.0 and XSD examples. In some cases, fully qualified URIs were used simply to illustrate the referencing concepts. However, the use of relative URIs is allowed and warranted in many cases. For information on processing relative URIs, see RFC2396.

5.6.3 Generating Temporary URIs

In general, when a WSDL 2.0 document is published for use by others, it should only contain URIs that are globally unique. This is usually done by allocating them under a domain name that is controlled by the issuer. For example, the W3C allocates namespace URIs under its base domain name, w3.org.

However, it is sometimes desirable to make up a temporary URI for an entity, for use during development, but not make the URI globally unique for all time and have it "mean" that version of the entity (schema, WSDL 2.0 document, etc.). *Reserved Top Level DNS Names* [*IETF RFC 2606 [p.84]*] specifies some URI base names that are reserved for use for this type of behavior. For example, the base URI "http://example.org/" can be used to construct a temporary URI without any unique association to an entity. This means that two people or programs could choose to simultaneously use the temporary URI " http://example.org/userSchema" for two completely different schemas. As long as the scope of use of these URIs does not intersect, then they would be unique enough. However, it is not recommended that " http://example.org/" be used as a base for stable, fixed entities.

6. References

6.1 Normative References

[IETF RFC 2119]
: *Key words for use in RFCs to Indicate Requirement Levels*, S. Bradner, Author. Internet Engineering Task Force, June 1999. Available at http://www.ietf.org/rfc/rfc2119.txt.

[IETF RFC 3986]
: *Uniform Resource Identifiers (URI): Generic Syntax*, T. Berners-Lee, R. Fielding, L. Masinter, Authors. Internet Engineering Task Force, January 2005. Available at http://www.ietf.org/rfc/rfc3986.txt.

[IETF RFC 3987]
: *Internationalized Resource Identifiers (IRIs)*, M. Duerst, M. Suignard, Authors. Internet Engineering Task Force, January 2005. Available at http://www.ietf.org/rfc/rfc3987.txt.

[XML 1.0]
: *Extensible Markup Language (XML) 1.0 (Third Edition)*, T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, and F. Yergeau, Editors. World Wide Web Consortium, 4 February 2004. This version of the XML 1.0 Recommendation is http://www.w3.org/TR/2004/REC-xml-20040204/. The latest version of "Extensible Markup Language (XML) 1.0" is available at http://www.w3.org/TR/REC-xml.

[XML Information Set]
: *XML Information Set (Second Edition)*, J. Cowan and R. Tobin, Editors. World Wide Web Consortium, 4 February 2004. This version of the XML Information Set Recommendation is http://www.w3.org/TR/2004/REC-xml-infoset-20040204. The latest version of XML Information Set is available at http://www.w3.org/TR/xml-infoset.

[XML Namespaces]
: *Namespaces in XML*, T. Bray, D. Hollander, and A. Layman, Editors. World Wide Web Consortium, 14 January 1999. This version of the XML Information Set Recommendation is http://www.w3.org/TR/1999/REC-xml-names-19990114. The latest version of Namespaces in XML is available at http://www.w3.org/TR/REC-xml-names.

[XML Schema: Structures]
: *XML Schema Part 1: Structures*, H. Thompson, D. Beech, M. Maloney, and N. Mendelsohn, Editors. World Wide Web Consortium, 28 October 2004. This version of the XML Schema Part 1 Recommendation is http://www.w3.org/TR/2004/REC-xmlschema-1-20041028. The latest version of XML Schema Part 1 is available at http://www.w3.org/TR/xmlschema-1.

[XML Schema: Datatypes]
: *XML Schema Part 2: Datatypes*, P. Byron and A. Malhotra, Editors. World Wide Web Consortium, 28 October 2004. This version of the XML Schema Part 2 Recommendation is http://www.w3.org/TR/2004/REC-xmlschema-2-20041028. The latest version of XML Schema Part 2 is available at http://www.w3.org/TR/xmlschema-2.

[RFC 3023]
IETF "RFC 3023: XML Media Types", M. Murata, S. St. Laurent, D. Kohn, July 1998.(See*http://www.ietf.org/rfc/rfc3023.txt*.)

[WSDL 2.0 Core]
Web Services Description Language (WSDL) Version 2.0 Part 1: Core Language, R. Chinnici, J-J. Moreau, A. Ryman, S. Weerawarana, Editors. World Wide Web Consortium, 6 January 2006. This version of the "Web Services Description Language (WSDL) Version 2.0 Part 1: Core Language" Specification is available is available at http://www.w3.org/TR/2006/CR-wsdl20-20060106. The latest version of "Web Services Description Language (WSDL) Version 2.0 Part 1: Core Language" is available at http://www.w3.org/TR/wsdl20.

[WSDL 2.0 Adjuncts]
Web Services Description Language (WSDL) Version 2.0 Part 2: Adjuncts , R. Chinnici, H. Haas, A. Lewis, J-J. Moreau, D. Orchard, S. Weerawarana, Editors. World Wide Web Consortium, 6 January 2006. This version of the "Web Services Description Language (WSDL) Version 2.0 Part 2: Adjuncts" Specification is available at http://www.w3.org/TR/2006/CR-wsdl20-adjuncts-20060106. The latest version of "Web Services Description Language (WSDL) Version 2.0 Part 2: Adjuncts" is available at http://www.w3.org/TR/wsdl20-adjuncts.

[WSDL 2.0 SOAP 1.1 Binding]
Web Services Description Language (WSDL) Version 2.0 SOAP 1.1 Binding , A. Vedamuthu, Editor. World Wide Web Consortium, 6 January 2006. This version of the "Web Services Description Language (WSDL) Version 2.0 SOAP 1.1 Binding" Specification is available at http://www.w3.org/TR/2006/WD-wsdl20-soap11-binding-20060106. The latest version of "Web Services Description Language (WSDL) Version 2.0 SOAP 1.1 Binding" is available at http://www.w3.org/TR/wsdl20-soap11-binding.

[WSDL 2.0 RDF Mapping]
Web Services Description (WSDL) Version 2.0: RDF Mapping, J. Kopecký, B. Parsia, Editors. W3C Working Draft, 4 November 2005. This version of the "Web Services Description Version 2.0: RDF Mapping" Specification is available at http://www.w3.org/TR/2005/WD-wsdl20-rdf-20051104/. The latest version of "Web Services Description Version 2.0: RDF Mapping" is available at http://www.w3.org/TR/wsdl20-rdf/.

[Web Architecture]
Architecture of the World Wide Web, Volume One, Ian Jacobs, Norman Walsh, Editors. W3C Recommendation, 15 December, 2004. Available at http://www.w3.org/TR/2004/REC-webarch-20041215/ .

[WS Architecture]
Web Services Architecture, David Booth, Hugo Haas, Francis McCabe, Eric Newcomer, Michael Champion, Chris Ferris, David Orchard, Editors. W3C Working Group Note, 11 February 2004. Available at http://www.w3.org/TR/2004/NOTE-ws-arch-20040211/ .

[WS Glossary]
Web Services Glossary, Hugo Haas, Allen Brown, Editors. W3C Working Group Note, 11 February 2004. Available at http://www.w3.org/TR/2004/NOTE-ws-gloss-20040211/ .

[Describing Media Content of Binary Data in XML]
Describing Media Content of Binary Data in XML , Anish Karmarkar, Ümit Yalçınalp, Editors. W3C Working Group Note 4 May 2005. Available at http://www.w3.org/TR/xml-media-types/

6.2 Informative References

[IETF RFC 2606]
Reserved Top Level DNS Names, D. Eastlake, A. Panitz, Authors. Network Working Group, Internet Engineering Task Force, June 1999. Available at http://www.ietf.org/rfc/rfc2606.txt.

[IETF RFC 2616]
Hypertext Transfer Protocol -- HTTP/1.1, R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, T. Berners-Lee, Authors. Internet Engineering Task Force, June 1999. Available at http://www.ietf.org/rfc/rfc2616.txt.

[IETF RFC 2818]
HTTP Over TLS, E. Rescorla, Author. Internet Engineering Task Force, May 2000. Available at http://www.ietf.org/rfc/rfc2818.txt.

[SOAP 1.1]
Simple Object Access Protocol (SOAP) 1.1, D. Box, D. Ehnebuske, G. Kakivaya, A. Layman, N. Mendelsohn, H. Frystyk Nielsen, S. Thatte, D. Winer, Editors. World Wide Web Consortium, 8 May 2000. This version of the Simple Object Access Protocol 1.1 Note is http://www.w3.org/TR/2000/NOTE-SOAP-20000508.

[SOAP 1.2 Part 1: Messaging Framework]
SOAP Version 1.2 Part 1: Messaging Framework, M. Gudgin, M. Hadley, N. Mendelsohn, J-J. Moreau, H. Frystyk Nielsen, Editors. World Wide Web Consortium, 24 June 2003. This version of the "SOAP Version 1.2 Part 1: Messaging Framework" Recommendation is http://www.w3.org/TR/2003/REC-soap12-part1-20030624/. The latest version of "SOAP Version 1.2 Part 1: Messaging Framework" is available at http://www.w3.org/TR/soap12-part1/.

[SOAP 1.2 Part 2: Adjuncts]
SOAP Version 1.2 Part 2: Adjuncts, M. Gudgin, M. Hadley, N. Mendelsohn, J-J. Moreau, and H. Frystyk Nielsen, Editors. World Wide Web Consortium, 7 May 2003. This version of the "SOAP Version 1.2 Part 2: Adjuncts" Recommendation is http://www.w3.org/TR/2003/REC-soap12-part2-20030624/. The latest version of "SOAP Version 1.2 Part 2: Adjuncts" is available at http://www.w3.org/TR/soap12-part2/.

[SOAP MTOM]
SOAP Message Transmission Optimization Mechanism , M. Gudgin, N. Mendelsohn, M. Nottingham, H. Ruellan, Editors. World Wide Web Consortium, 25 January, 2005. This version of SOAP Message Transmission Optimization Mechanism is available at http://www.w3.org/TR/2005/REC-soap12-mtom-20050125/.

[WSD Requirements]
Web Services Description Requirements, J. Schlimmer, Editor. World Wide Web Consortium, 17 October 2002. This version of the Web Services Description Requirements document is http://www.w3.org/TR/2002/WD-ws-desc-reqs-20021028. The latest version of Web Services Description Requirements is available at http://www.w3.org/TR/ws-desc-reqs.

[WS-Addressing]
Web Services Addressing 1.0 - Core, Martin Gudgin, Microsoft Corp, Marc Hadley, Sun Microsystems, Inc, Editor. World Wide Web Consortium, 17 August 2005. This version of the Web Services Addressing 1.0 - Core document is available at http://www.w3.org/TR/ws-addr-core/. The latest version of Web Services Description Requirements is available at http://www.w3.org/TR/ws-addr-core/.

[XPointer Framework]
XPointer Framework,Paul Grosso, Eve Maler, Jonathan Marsh, Norman Walsh, Editors. World Wide Web Consortium, 22 November 2002. This version of the XPointer Framework Proposed Recommendation is http://www.w3.org/TR/2003/REC-xptr-framework-20030325/ The latest version of XPointer Framework is available at http://www.w3.org/TR/xptr-framework/.

[W3C TAG Finding: Use of HTTP GET]
URIs, Addressability, and the use of HTTP GET and POST, Ian Jacobs, Editor. World Wide Web Consortium, 21 March 2004. This version of TAG finding is available at http://www.w3.org/2001/tag/doc/whenToUseGet.html

[W3C TAG Finding: Versioning]
Versioning XML Languages David Orchard, Norman Walsh. Proposed TAG Finding 16 November 2003. Available at http://www.w3.org/2001/tag/doc/versioning.html

[WebArch: Extensible Languages]
Web Architecture: Extensible Languages , Tim Berners-Lee, Dan Connolly, Authors. W3C Note 10 Feb 1998. Available at http://www.w3.org/TR/NOTE-webarch-extlang

[XML Schema: Versioning Use-Cases]
XML Schema Versioning Use Cases , Hoylen Sue. W3C XML Schema Working Group Draft, 15 April 2005. Available at http://www.w3.org/XML/2005/xsd-versioning-use-cases/

[SW VocabManagementNote]
Vocabulary Management , Thomas Baker, et al. Semantic Web Best Practices and Deployment Working Group Note, 8 Feb 2005. Available at http://esw.w3.org/topic/VocabManagementNote

[RELAX NG]
RELAX NG Specification, James Clark, MURATA Makoto, Editors. OASIS Committee Specification, 3 December 2001. Available at http://www.oasis-open.org/committees/relax-ng/spec-20011203.html

[JAX RPC 1.1]
Java(TM) API for XML-based Remote Procedure Call (JAX-RPC) Specification, version 1.1, Roberto Chinnici,et al. 14 October, 2003. Available at http://java.sun.com/xml/downloads/jaxrpc.html

[REST]
Representational State Transfer (REST), Roy Thomas Fielding, Author. 2000. Available at http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm

[RDF]
Resource Description Framework (RDF): Concepts and Abstract Syntax, Graham Klyne, Jeremy J. Carroll, Editors. W3C Recommendation, 10 February 2004. Available at http://www.w3.org/TR/rdf-concepts/

[OWL]
OWL Web Ontology Language Reference, Mike Dean,Guus Schreiber, Editors. W3C Recommendation 10 February 2004 . Available at http://www.w3.org/TR/owl-ref/

[Alternative Schema Languages Support]
Discussion of Alternative Schema Languages and Type System Support in WSDL, A. Lewis, B. Parsia, Editors.

A. Acknowledgements (Non-Normative)

This document is the work of the W3C Web Service Description Working Group.

Members of the Working Group are (at the time of writing, and by alphabetical order): Charlton Barreto (Adobe Systems Inc.), Allen Brookes (Rogue Wave Softwave), Dave Chappell (Sonic Software), Helen Chen (Agfa-Gevaert N. V.), Roberto Chinnici (Sun Microsystems), Kendall Clark (University of Maryland), Glen Daniels (Sonic Software), Paul Downey (British Telecommunications), Youenn Fablet (Canon), Hugo Haas (W3C), Tom Jordahl (Macromedia), Anish Karmarkar (Oracle Corporation), Jacek Kopecky (DERI Innsbruck at the Leopold-Franzens-Universität Innsbruck, Austria), Amelia Lewis (TIBCO Software, Inc.), Michael Liddy (Education.au Ltd.), Kevin Canyang Liu (SAP AG), Jonathan Marsh (Microsoft Corporation), Josephine Micallef (SAIC - Telcordia Technologies), Jeff Mischkinsky (Oracle Corporation), Dale Moberg (Cyclone Commerce), Jean-Jacques Moreau (Canon), Mark Nottingham (BEA Systems, Inc.), David Orchard (BEA Systems, Inc.), Vivek Pandey (Sun Microsystems), Bijan Parsia (University of Maryland), Gilbert Pilz (BEA Systems, Inc.), Tony Rogers (Computer Associates), Arthur Ryman (IBM), Adi Sakala (IONA Technologies), Asir Vedamuthu (Microsoft Corporation), Sanjiva Weerawarana (WSO2), Ümit Yalçınalp (SAP AG).

Previous members were: Lily Liu (webMethods, Inc.), Don Wright (Lexmark), Joyce Yang (Oracle Corporation), Daniel Schutzer (Citigroup), Dave Solo (Citigroup), Stefano Pogliani (Sun Microsystems), William Stumbo (Xerox), Stephen White (SeeBeyond), Barbara Zengler (DaimlerChrysler Research and Technology), Tim Finin (University of Maryland), Laurent De Teneuille (L'Echangeur), Johan Pauhlsson (L'Echangeur), Mark Jones (AT&T), Steve Lind (AT&T), Sandra Swearingen (U.S. Department of Defense, U.S. Air Force), Philippe Le Hégaret (W3C), Jim Hendler (University of Maryland), Dietmar Gaertner (Software AG), Michael Champion (Software AG), Don Mullen (TIBCO Software, Inc.), Steve Graham (Global Grid Forum), Steve Tuecke (Global Grid Forum), Michael Mahan (Nokia), Bryan Thompson (Hicks & Associates), Ingo Melzer (DaimlerChrysler Research and Technology), Sandeep Kumar (Cisco Systems), Alan Davies (SeeBeyond), Jacek Kopecky (Systinet), Mike Ballantyne (Electronic Data Systems), Mike Davoren (W. W. Grainger), Dan Kulp (IONA Technologies), Mike McHugh (W. W. Grainger), Michael Mealling (Verisign), Waqar Sadiq (Electronic Data Systems), Yaron Goland (BEA Systems, Inc.), Ümit Yalçınalp (Oracle Corporation), Peter Madziak (Agfa-Gevaert N. V.), Jeffrey Schlimmer (Microsoft Corporation), Hao He (The Thomson Corporation), Erik Ackerman (Lexmark), Jerry Thrasher (Lexmark), Prasad Yendluri (webMethods, Inc.), William Vambenepe (Hewlett-Packard Company), David Booth (W3C), Sanjiva Weerawarana (IBM), Charlton Barreto (webMethods, Inc.), Asir Vedamuthu (webMethods, Inc.), Igor Sedukhin (Computer Associates), Martin Gudgin (Microsoft Corporation), Rebecca Bergersen (IONA Technologies), Ugo Corda (SeeBeyond).

The people who have contributed to discussions on www-ws-desc@w3.org are also gratefully acknowledged.