SOAP Version 1.2

W3C Working Draft 9 July 2001

This version:
: http://www.w3.org/TR/2001/WD-soap12-20010709/

Latest version:
: http://www.w3.org/TR/soap12/

Editors:
: Martin Gudgin (DevelopMentor)
: Marc Hadley (Sun Microsystems)
: Jean-Jacques Moreau (Canon)
: Henrik Frystyk Nielsen (Microsoft Corp.)

Copyright ©2001 W3C® (MIT, INRIA, Keio), All Rights Reserved. W3C liability, trademark, document use and software licensing rules apply.

Abstract

SOAP version 1.2 is a lightweight protocol for exchange of information in a decentralized, distributed environment. It is an XML based protocol that consists of four parts: an envelope that defines a framework for describing what is in a message and how to process it, a set of encoding rules for expressing instances of application-defined data types, a convention for representing remote procedure calls and responses and a binding convention for exchanging messages using an underlying protocol. SOAP can potentially be used in combination with a variety of other protocols; however, the only bindings defined in this document describe how to use SOAP in combination with HTTP and the experimental HTTP Extension Framework.

Status of this Document

This section describes the status of this document at the time of its publication. Other documents may supersede this document. The latest status of this document series is maintained at the W3C.

This is the first W3C Working Draft of the SOAP version 1.2 specification for review by W3C members and other interested parties. It has been produced by the XML Protocol Working Group (WG), which is part of the XML Protocol Activity.

The XML Protocol Protocol Working Group has, in keeping with its charter, produced a set of requirements and usage scenarios that have been published as a Working Draft. To better evaluate SOAP/1.1 against these requirements and usage scenarios, the Working Group has produced an abstract model and a glossary of terms and concepts used by the Working Group. In addition, the Working Group has produced an issues list that describes issues and concerns raised by mapping its requirements and the XMLP abstract model against the SOAP/1.1 specification as well as issues raised on the <xml-dist-app@w3.org> mailing list against SOAP/1.1.

The current name for this specification is SOAP version 1.2, this first Working Draft being based on SOAP/1.1 as per the Working Group's charter (see change log in appendix D)

Comments on this document should be sent to xmlp-comments@w3.org (public archives). It is inappropriate to send discussion emails to this address.

Discussion of this document takes place on the public <xml-dist-app@w3.org> mailing list (Archives) per the email communication rules in the XML Protocol Working Group Charter.

This is a public W3C Working Draft. It is a draft document and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use W3C Working Drafts as reference material or to cite them as other than "work in progress". A list of all W3C technical reports can be found at http://www.w3.org/TR/.

Table of Contents

1. Introduction

- 1.1 Design Goals
- 1.2 Notational Conventions
- 1.3 Examples of SOAP Messages
- 1.4 SOAP Terminology
  - 1.4.1 Protocol Concepts
  - 1.4.2 Data Encapsulation Concepts
  - 1.4.3 Message Sender and Receiver Concepts
  - 1.4.4 Data Encoding Concepts

2. The SOAP Message Exchange Model

- 2.1 Nodes
- 2.2 Actors and Nodes
- 2.3 Targeting SOAP Header Blocks
- 2.4 Understanding Headers
- 2.5 Processing Messages

3. Relation to XML

4. SOAP Envelope

  - 4.1.1 SOAP encodingStyle Attribute
  - 4.1.2 Envelope Versioning Model
- 4.2 SOAP Header
  - 4.2.1 Use of Header Attributes
  - 4.2.2 SOAP actor Attribute
  - 4.2.3 SOAP mustUnderstand Attribute
- 4.3 SOAP Body
  - 4.3.1 Relationship between SOAP Header and Body
- 4.4 SOAP Fault
  - 4.4.1 SOAP Fault Codes
  - 4.4.2 MustUnderstand Faults

- 5. SOAP Encoding
  - 5.1 Rules for Encoding Types in XML
  - 5.2 Simple Types
    - 5.2.1 Strings
    - 5.2.2 Enumerations
    - 5.2.3 Array of Bytes
  - 5.3 Polymorphic Accessor
  - 5.4 Compound Types
    - 5.4.1 Compound Values and References to Values
    - 5.4.2 Arrays
      - 5.4.2.1 PartiallyTransmitted Arrays
      - 5.4.2.2 SparseArrays
    - 5.4.3 Generic Compound Types
  - 5.5 Default Values
  - 5.6 SOAP root Attribute
- 6. Using SOAP in HTTP
  - 6.1 SOAP HTTP Request
    - 6.1.1 The SOAPAction HTTP Header Field
  - 6.2 SOAP HTTP Response
  - 6.3 The HTTP Extension Framework
  - 6.4 SOAP HTTP Examples
- 7. Using SOAP for RPC
  - 7.1 RPC and SOAP Body
  - 7.2 RPC and SOAP Header
- 8. Security Considerations
- 9. References
  - 9.1. Normative references
  - 9.2. Informative references
- A. SOAP Envelope Examples
  - A.1 Sample Encoding of Call Requests
  - A.2 Sample Encoding of Response
- B. Acknowledgements
- C. Version Transition From SOAP/1.1 to SOAP/1.2
- D. Change Log
  - D.1 SOAP Specification Changes
  - D.2 XML Schema Changes

1. Introduction

SOAP version 1.2 provides a simple and lightweight mechanism for exchanging structured and typed information between peers in a decentralized, distributed environment using XML. SOAP does not itself define any application semantics such as a programming model or implementation specific semantics; rather it defines a simple mechanism for expressing application semantics by providing a modular packaging model and encoding mechanisms for encoding application defined data. This allows SOAP to be used in a large variety of systems ranging from messaging systems to remote procedure calls (RPC).

SOAP consists of four parts:

1. The SOAP envelope (see section 4) construct defines an overall framework for expressing **what** is in a message, **who** should deal with it, and whether it is **optional** or **mandatory**.
2. The SOAP encoding rules (see section 5) defines a serialization mechanism that can be used to exchange instances of application-defined datatypes.
3. The SOAP RPC representation (see section 7) defines a convention that can be used to represent remote procedure calls and responses.
4. The SOAP binding (see section 6) defines a convention for exchanging SOAP envelopes between peers using an underlying protocol for transport.

To simplify the specification, these four parts are functionally orthogonal. In particular, the envelope and the encoding rules are defined in different namespaces.

This specification defines two SOAP bindings that describe how a SOAP message can be carried in HTTP [5] messages either with or without the experimental HTTP Extension Framework [6].

1.1 Design Goals

Two major design goals for SOAP are simplicity and extensibility. SOAP attempts to meet these goals by omitting features often found in messaging systems and distributed object systems such as:

- distributed garbage collection;
- boxcarring or batching of messages;
- objects-by-reference (which requires distributed garbage collection);
- activation (which requires objects-by-reference).

1.2 Notational Conventions

The keywords "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC-2119 [2].

The namespace prefixes "env" and "enc" used in the prose sections of this document are associated with the SOAP namespace names "http://www.w3.org/2001/06/soap-envelope" and "http://www.w3.org/2001/06/soap-encoding" respectively.

The namespace prefixes "xs" and "xsi" used in the prose sections of this document are associated with the namespace names "http://www.w3.org/2001/XMLSchema" and "http://www.w3.org/2001/XMLSchema-instance" respectively, both of which are defined in the XML Schemas specification [10,11].

Note that the choice of any namespace prefix is arbitrary and not semantically significant.

Namespace URIs of the general form "http://example.org/..." and "http://example.com/..." represent an

application-dependent or context-dependent URI [4].

This specification uses the augmented Backus-Naur Form (BNF) as described in RFC-2616 [5].

Editorial notes are indicated with yellow background (may not appear in all media) and prefixed with "Ednote".

1.3 Examples of SOAP Messages

The first example shows a simple notification message expressed in SOAP. The message contains the header block "alertcontrol" and the body block "alert" which are both application defined and not defined by SOAP. The header block contains the parameters "priority" and "expires" which may be of use to intermediaries as well as the ultimate destination of the message. The body block contains the actual notification message to be delivered.

Example 0

```
<env:Envelope xmlns:env="http://www.w3.org/2001/06/soap-envelope">
 <env:Header>
  <n:alertcontrol xmlns:n="http://example.org/alertcontrol">
   <n:priority>1</n:priority>
   <n:expires>2001-06-22T14:00:00-05:00</n:expires>
  </n:alertcontrol>
 </env:Header>
 <env:Body>
  <m:alert xmlns:m="http://example.org/alert">
   <m:msg>Pick up Mary at school at 2pm</m:msg>
  </m:alert>
 </env:Body>
</env:Envelope>
```

Sample SOAP Message containing a header block and a body block

SOAP messages may be bound to different underlying protocols and used in a variety of message exchange patterns. The following example shows SOAP used in connection with HTTP as the underlying protocol taking advantage of the request/response mechanism provided by HTTP (see section section 6).

Examples 1 and 2 show a sample SOAP/HTTP request and a sample SOAP/HTTP response. The SOAP/HTTP request contains a block called GetLastTradePrice which takes a single parameter, the ticker symbol for a stock. As in the previous example, the GetLastTradePrice element is not defined by SOAP itself. The service's response to this request contains a single parameter, the price of the stock. The SOAP Envelope element is the top element of the XML document representing the SOAP message. XML namespaces are used to disambiguate SOAP identifiers from application specific identifiers.

Example 1

```
POST /StockQuote HTTP/1.1
Host: www.stockquoteserver.com
Content-Type: text/xml; charset="utf-8"
Content-Length: nnnn
SOAPAction: "http://example.org/2001/06/quotes"

<env:Envelope xmlns:env="http://www.w3.org/2001/06/soap-envelope" >
 <env:Body>
  <m:GetLastTradePrice
        env:encodingStyle="http://www.w3.org/2001/06/soap-encoding"
        xmlns:m="http://example.org/2001/06/quotes">
    <symbol>DIS</symbol>
  </m:GetLastTradePrice>
 </env:Body>
</env:Envelope>
```

Sample SOAP Message embedded in an HTTP Request

Example 2 shows the SOAP message sent by the StockQuote service in the corresponding HTTP response to the request from Example 1.

Example 2

```
HTTP/1.1 200 OK
Content-Type: text/xml; charset="utf-8"
Content-Length: nnnn

<env:Envelope xmlns:env="http://www.w3.org/2001/06/soap-envelope" >
 <env:Body>
  <m:GetLastTradePriceResponse
        env:encodingStyle="http://www.w3.org/2001/06/soap-encoding"
        xmlns:m="http://example.org/2001/06/quotes">
   <Price>34.5</Price>
  </m:GetLastTradePriceResponse>
 </env:Body>
</env:Envelope>
```

Sample SOAP Message embedded in an HTTP Response

More examples are available in Appendix A.

1.4 SOAP Terminology

1.4.1 Protocol Concepts

SOAP

The formal set of conventions governing the format and processing rules of a SOAP message and basic control of interaction among applications generating and accepting SOAP messages for the purpose of exchanging information along a SOAP message path.

SOAP binding

The formal set of rules for carrying a SOAP message within or on top of another protocol (underlying protocol) for the purpose of transmission. Typical SOAP bindings include carrying a SOAP message within an HTTP message, or on top of TCP.

SOAP node

A SOAP node processes a SOAP message according to the formal set of conventions defined by SOAP. The SOAP node is responsible for enforcing the rules that govern the exchange of SOAP messages and accesses the services provided by the underlying protocols through SOAP bindings. Non-compliance with SOAP conventions can cause a SOAP node to generate a SOAP fault (see also SOAP receiver and SOAP sender).

1.4.2 Data Encapsulation Concepts

SOAP message

A SOAP message is the basic unit of communication between peer SOAP nodes.

SOAP envelope

The outermost syntactic construct or structure of a SOAP message defined by SOAP within which all other syntactic elements of the message are enclosed.

SOAP block

A syntactic construct or structure used to delimit data that logically constitutes a single computational unit as seen by a SOAP node. A SOAP block is identified by the fully qualified name of the outer element for the block, which consists of the namespace URI and the local name. A block encapsulated within the SOAP header is called a header block and a block encapsulated within a SOAP body is called a body block.

SOAP header

A collection of zero or more SOAP blocks which may be targeted at any SOAP receiver within the SOAP message path.

SOAP body

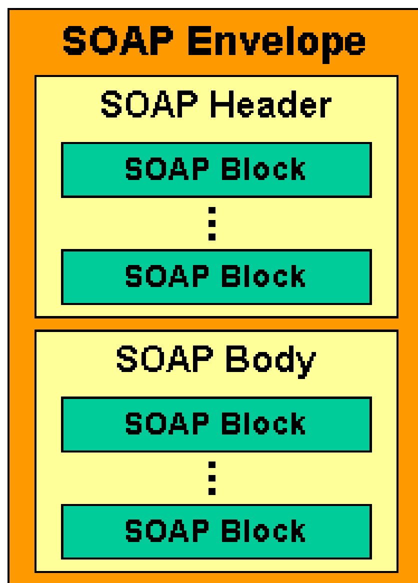
A collection of zero, or more SOAP blocks targeted at the ultimate SOAP receiver within the SOAP message path.

SOAP fault

A special SOAP block which contains fault information generated by a SOAP node.

The following diagram illustrates how a SOAP message is composed.

Figure 1: Encapsulation model illustrating the parts of a SOAP message



1.4.3 Message Sender and Receiver Concepts

SOAP sender

A SOAP sender is a SOAP node that transmits a SOAP message.

SOAP receiver

A SOAP receiver is a SOAP node that accepts a SOAP message.

SOAP message path

The set of SOAP senders and SOAP receivers through which a single SOAP message passes. This includes the initial SOAP sender, zero or more SOAP intermediaries, and the ultimate SOAP receiver.

initial SOAP sender

The SOAP sender that originates a SOAP message as the starting point of a SOAP message path.

SOAP intermediary

A SOAP intermediary is both a SOAP receiver and a SOAP sender, target-able from within a SOAP message. It

processes a defined set of blocks in a SOAP message along a SOAP message path. It acts in order to forward the SOAP message towards the ultimate SOAP receiver.

ultimate SOAP receiver

The SOAP receiver that the initial sender specifies as the final destination of the SOAP message within a SOAP message path. A SOAP message may not reach the ultimate recipient because of a SOAP fault generated by a SOAP node along the SOAP message path.

1.4.4 Data Encoding Concepts

SOAP data model

A set of abstract constructs that can be used to describe common data types and link relationships in data.

SOAP data encoding

The syntactic representation of data described by the SOAP data model within one or more SOAP blocks in a SOAP message.

2. The SOAP Message Exchange Model

SOAP messages are fundamentally one-way transmissions from a SOAP sender to a SOAP receiver, but as illustrated above, SOAP messages are often combined to implement patterns such as request/response.

SOAP implementations can be optimized to exploit the unique characteristics of particular network systems. For example, the HTTP binding described in section 6 provides for SOAP response messages to be delivered as HTTP responses, using the same connection as the inbound request.

2.1 SOAP Nodes

A SOAP node can be the initial SOAP sender, the ultimate SOAP receiver, or a SOAP intermediary, in which case it is both a SOAP sender and a SOAP receiver. SOAP does not provide a routing mechanism, however SOAP does recognise that a SOAP sender originates a SOAP message which is sent to an ultimate SOAP receiver, via zero or more SOAP intermediaries.

A SOAP node receiving a SOAP message MUST perform processing, generate SOAP faults, SOAP responses, and if appropriate send additional SOAP messages, as provided by the remainder of this specification.

2.2 SOAP Actors and SOAP Nodes

In processing a SOAP message, a SOAP node is said to act in the role of one or more SOAP actors, each of which is identified by a URI known as the SOAP actor name. Each SOAP node MUST act in the role of the special SOAP actor named "http://www.w3.org/2001/06/soap-envelope/actor/next", and can additionally assume the roles of zero or more other SOAP actors. A SOAP node can establish itself as the ultimate SOAP receiver by acting in the (additional) role of the anonymous SOAP actor. The roles assumed MUST be invariant during the processing of an individual SOAP message; because this specification deals only with the processing of individual SOAP messages, no statement is made regarding the possibility that a given piece of software might or might not act in varying roles when processing more than one SOAP message.

While the purpose of a SOAP actor name is to identify a SOAP node, there are no routing or message exchange semantics associated with the SOAP actor name. For example, SOAP Actors MAY be named with a URI useable to route SOAP messages to an appropriate SOAP node. Conversely, it is also appropriate to use SOAP actor roles with names that are related more indirectly to message routing (e.g. "http://example.org/banking/anyAccountMgr") or which are unrelated to routing (e.g. a URI meant to identify "all cache management software"; such a header might be used, for example, to carry an indication to any concerned software that the containing SOAP message is idempotent, and can safely be cached and

replayed.)

2.3 Targeting SOAP Header Blocks

SOAP header blocks carry optional env:actor attributes (see section 4.2.2) that are used to target them to the appropriate SOAP node(s). SOAP header blocks with no env:actor attribute and the SOAP body are implicitly targeted at the anonymous SOAP actor, implying that they are to be processed by the ultimate SOAP receiver. We refer to the (implicit or explicit) value of the SOAP actor attribute as the SOAP actor for the corresponding SOAP block (either a SOAP header block or a SOAP body block).

We say that a SOAP block is targeted to a SOAP node if the SOAP actor (if present) on the block matches (see [8]) a role played by the SOAP node, or in the case of a SOAP block with no actor attribute (including SOAP body blocks), if the SOAP node has assumed the role of the anonymous SOAP actor.

2.4 Understanding SOAP Headers

We presume that specifications for a wide variety of header functions will be developed over time, and that each SOAP node MAY include the software necessary to implement one or more such extensions. We say that a SOAP header block is understood by a SOAP node if the software at that SOAP node has been written to fully conform to and implement the semantics conveyed by the fully qualified name of the outer-most element of that block.

When a SOAP header block is tagged with a SOAP mustUnderstand attribute with a value of "1", the targeted SOAP node MUST: either process the SOAP block according to the semantics conveyed by the fully qualified name of the outer-most element of that block; or not process the SOAP message at all, and fail (see section 4.4).

2.5 Processing SOAP Messages

This section sets out the rules by which SOAP messages are processed. Unless otherwise stated, processing must be semantically equivalent to performing the following steps separately, and in the order given. Note however that nothing in this specification should be taken to prevent the use of optimistic concurrency, roll back, or other techniques that might provide increased flexibility in processing order as long as all SOAP messages, SOAP faults and application-level side effects are equivalent to those that would be obtained by direct implementation of the following rules.

1. Generate a single SOAP mustUnderstand fault if one or more SOAP blocks targeted at the SOAP node carry the attribute env:mustUnderstand="1" and are not understood by that node. If such a fault is generated, any further processing MUST NOT be done.
2. Process SOAP blocks targeted at the SOAP node, generating SOAP faults if necessary. A SOAP node MUST process SOAP blocks identified as env:mustUnderstand="1". A SOAP node MAY process or ignore SOAP blocks not so identified. In all cases where a SOAP block is processed, the SOAP node must understand the SOAP block and must do such processing in a manner fully conformant with the specification for that SOAP block. Faults, if any, must also conform to the specification for the processed SOAP block. It is possible that the processing of particular SOAP block would control or determine the order of processing for other SOAP blocks. For example, one could create a SOAP header block to force processing of other SOAP header blocks in lexical order. In the absence of such a SOAP block, the order of processing is at the discretion of the SOAP node. SOAP nodes can make reference to any information in the SOAP envelope when processing a SOAP block. For example, a caching function can cache the entire SOAP message, if desired.

If the SOAP node is a SOAP intermediary, the SOAP message pattern and results of processing (e.g. no fault generated) MAY require that the SOAP message be sent further along the SOAP message path. Such relayed SOAP messages MUST contain all SOAP header blocks and the SOAP body blocks from the original SOAP message, in the original order, except that SOAP header blocks targeted at the SOAP intermediary MUST be removed (such SOAP blocks are removed regardless of whether they were processed or ignored). Additional SOAP header blocks MAY be inserted at any point in the SOAP message, and such inserted SOAP header blocks MAY be indistinguishable from one or more just removed (effectively leaving them in place, but emphasizing the need to reinterpret at each SOAP node along the SOAP message path.)

3. Relation to XML

All SOAP messages are encoded using XML (see [7] for more information on XML).

A SOAP application SHOULD include the proper SOAP namespace on all elements and attributes defined by SOAP in messages that it generates. A SOAP application MUST be able to process SOAP namespaces in messages that it receives. It MUST discard messages that have incorrect namespaces (see section 4.4) and it MAY process SOAP messages without SOAP namespaces as though they had the correct SOAP namespaces.

SOAP defines the following namespaces (see [8] for more information on XML namespaces):

- The SOAP envelope has the namespace identifier "http://www.w3.org/2001/06/soap-envelope"
- The SOAP serialization has the namespace identifier "http://www.w3.org/2001/06/soap-encoding"
- The SOAP mustUnderstand fault namespace identifier "http://www.w3.org/2001/06/soap-faults"
- The SOAP upgrade namespace identifier "http://www.w3.org/2001/06/soap-upgrade"

Schema documents for these namespaces can be found by dereferencing the namespace identifiers.

A SOAP message MUST NOT contain a Document Type Declaration. A SOAP message MUST NOT contain Processing Instructions. [7]

SOAP uses the local, unqualified "id" attribute of type "ID" to specify the unique identifier of an encoded element. SOAP uses the local, unqualified attribute "href" of type "anyURI" to specify a reference to that value, in a manner conforming to the XML Specification [7], XML Schema Specification [11], and XML Linking Language Specification [9].

With the exception of the SOAP mustUnderstand attribute (see section 4.2.3) and the SOAP actor attribute (see section 4.2.2), it is generally permissible to have attributes and their values appear in XML instances or alternatively in schemas, with equal effect. That is, declaration in a DTD or schema with a default or fixed value is semantically equivalent to appearance in an instance.

4. SOAP Envelope

A SOAP message is an XML document that consists of a mandatory SOAP envelope, an optional SOAP Header, and a mandatory SOAP Body. This XML document is referred to as a SOAP message for the rest of this specification. The namespace identifier for the elements and attributes defined in this section is "http://www.w3.org/2001/06/soap-envelope". A SOAP message contains the following:

- A SOAP envelope. This is the top element of the XML document representing the SOAP message.
- A SOAP Header. This is a generic mechanism for adding features to a SOAP message in a decentralized manner without prior agreement between the communicating parties (a SOAP sender, a SOAP receiver, and possibly one or more SOAP intermediaries). SOAP defines a few attributes that can be used to indicate who should deal with a feature and whether it is optional or mandatory (see section 4.2)
- A SOAP Body. This is a container for mandatory information intended for the ultimate SOAP receiver (see section 4.3). SOAP defines a SOAP fault for reporting errors.

The grammar rules are as follows:

1. SOAP envelope
   - The element name is "Envelope".
   - The element MUST be present in a SOAP message

   - The element MAY contain namespace declarations as well as additional attributes. If present, such additional attributes MUST be namespace-qualified. Similarly, the element MAY contain additional sub-elements. If present these sub-elements MUST be namespace-qualified and MUST come immediately after the SOAP Body.

2. SOAP Header (see section 4.2)
   - The element name is "Header".
   - The element MAY be present in a SOAP message. If present, the element MUST be the first immediate child element of a SOAP envelope.
   - The element MAY contain a set of SOAP header blocks, each being an immediate child element of the SOAP Header. All immediate child elements of the SOAP Header MUST be namespace-qualified.

3. SOAP Body (see section 4.3)
   - The element name is "Body".
   - The element MUST be present in a SOAP message and MUST be an immediate child element of a SOAP Envelope. It MUST come immediately after the SOAP Header, if present. Otherwise, it MUST be the first immediate child element of the SOAP envelope.
   - The element MAY contain a set of SOAP body blocks, each being an immediate child element of the SOAP Body. Immediate child elements of the SOAP Body MAY be namespace-qualified. At most one child element MAY be a SOAP fault. The SOAP fault is used to carry error information (see section 4.4).

4.1.1 SOAP encodingStyle Attribute

The SOAP encodingStyle global attribute can be used to indicate the serialization rules used in a SOAP message. This attribute MAY appear on any element, and is scoped to that element's contents and all child elements not themselves containing such an attribute, much as an XML namespace declaration is scoped. There is no default encoding defined for a SOAP message.

The attribute value is an ordered list of one or more URIs identifying the serialization rule or rules that can be used to deserialize the SOAP message indicated in the order of most specific to least specific. Example 3 shows three sample values for the encodingStyle attribute.

Example 3

```
encodingStyle="http://www.w3.org/2001/06/soap-encoding"
encodingStyle="http://example.org/encoding/restricted http://example.org/encoding/"
encodingStyle=""
```

Example values for the encodingStyle attribute

The serialization rules defined by SOAP in section 5 are identified by the URI "http://www.w3.org/2001/06/soap-encoding". SOAP messages using this particular serialization SHOULD indicate this using the SOAP encodingStyle attribute. In addition, all URIs syntactically beginning with "http://www.w3.org/2001/06/soap-encoding" indicate conformance with the SOAP encoding rules defined in section 5 (though with potentially tighter rules added).

A value of the zero-length URI ("") explicitly indicates that no claims are made for the encoding style of contained elements. This can be used to turn off any claims from containing elements.

4.1.2 Envelope Versioning Model

SOAP does not define a traditional versioning model based on major and minor version numbers. A SOAP message MUST contain a SOAP envelope associated with the "http://www.w3.org/2001/06/soap-envelope" namespace. If a SOAP message

is received by a SOAP node in which the SOAP envelope is associated with a different namespace, the SOAP node MUST treat this as a version error and generate a VersionMismatch SOAP fault (see section 4.4). A SOAP VersionMismatch fault message MUST use the SOAP/1.1 envelope namespace "http://schemas.xmlsoap.org/soap/envelope/" (see Appendix C).

4.2 SOAP Header

SOAP provides a flexible mechanism for extending a SOAP message in a decentralized and modular way without prior knowledge between the communicating parties. Typical examples of extensions that can be implemented as SOAP header blocks are authentication, transaction management, payment, etc.

The SOAP Header is encoded as the first immediate child element of the SOAP envelope. All immediate child elements of the SOAP Header are called SOAP header blocks.

The encoding rules for SOAP header blocks are as follows:

1. A SOAP header block is identified by its fully qualified element name, which consists of the namespace URI and the local name. All immediate child elements of the SOAP Header MUST be namespace-qualified.
2. The SOAP encodingStyle attribute MAY be used to indicate the encoding style used for the SOAP header blocks (see section 4.1.1).
3. The SOAP actor attribute (see section 4.2.2) and SOAP mustUnderstand attribute (see section 4.2.3) MAY be used to indicate which SOAP node will process the SOAP header block, and how it will be processed (see section 4.2.1).

4.2.1 Use of Header Attributes

The SOAP Header attributes defined in this section determine how a SOAP receiver should process an incoming SOAP message, as described in section 2. A SOAP sender generating a SOAP message SHOULD only use the SOAP Header attributes on immediate child elements of the SOAP Header. A SOAP receiver MUST ignore all SOAP Header attributes that are not applied to an immediate child element of the SOAP Header.

An example is a SOAP header block with an element identifier of "Transaction", a "mustUnderstand" value of "1", and a value of 5, as shown in Example 4.

Example 4

```
<env:Header xmlns:env="http://www.w3.org/2001/06/soap-envelope" >
  <t:Transaction xmlns:t="http://example.org/2001/06/tx" env:mustUnderstand="1" >
    5
  </t:Transaction>
</env:Header>
```

Example header with a single header block

4.2.2 SOAP actor Attribute

EdNote: This section partially overlaps with section 2. We expect this to be reconciled in a future revision of the specification.

A SOAP message travels from an initial SOAP sender to an ultimate SOAP receiver, potentially passing through a set of SOAP intermediaries along a SOAP message path. Both intermediaries as well as the ultimate SOAP receiver are identified by a URI.

Not all parts of a SOAP message may be intended for the ultimate SOAP receiver. They may be intended instead for one or more SOAP intermediaries on the SOAP message path. However, a SOAP intermediary MUST NOT forward further a SOAP header block intended for it. This would be considered as a breach of contract, the contract being only between the SOAP node which generated the SOAP header block, and the SOAP intermediary itself. However, the SOAP intermediary MAY instead insert a similar SOAP header block, which effectively sets up a new contract between that SOAP intermediary and the SOAP node at which the SOAP header block is targeted.

The SOAP actor global attribute can be used to indicate the SOAP node at which a particular SOAP header block is targeted. The value of the SOAP actor attribute is a URI. The special URI "http://www.w3.org/2001/06/soap-envelope/actor/next" indicates that the SOAP header block is intended for the very first SOAP node that processes the message. This is similar to the hop-by-hop scope model represented by the Connection header field in HTTP.

Omitting the SOAP actor attribute indicates that the SOAP header block is targeted at the ultimate SOAP receiver.

This attribute MUST appear in the SOAP message itself in order to be effective, and not in an eventual corresponding XML Schema (see section 3 and 4.2.1).

4.2.3 SOAP mustUnderstand Attribute

EdNote: This section partially overlaps with section 2. We expect this to be reconciled in a future revision of the specification.

The SOAP mustUnderstand global attribute can be used to indicate whether the processing of a SOAP header block is mandatory or optional at the target SOAP node. The target SOAP node itself is defined by the SOAP actor attribute (see section 4.2.2). The value of the SOAP mustUnderstand attribute is either "1" or "0". The absence of this attribute is semantically equivalent to its presence with the value "0", which means processing the block is optional.

When a SOAP header block is tagged with a SOAP mustUnderstand attribute with a value of "1", the targeted SOAP node MUST: either process the SOAP block according to the semantics conveyed by the fully qualified name of the outer-most element of that block; or not process the SOAP message at all, and fail (see section 4.4).

The SOAP mustUnderstand attribute allows for robust evolution. Elements tagged with the SOAP mustUnderstand attribute with a value of "1" MUST be presumed to somehow modify the semantics of their parent or peer elements. Tagging elements in this manner assures that this change in semantics will not be silently (and, presumably, erroneously) ignored by those who may not fully understand it.

This attribute MUST appear in the SOAP message itself in order to be effective, and not in an eventual corresponding XML Schema (see section 3 and 4.2.1).

4.3 SOAP Body

The SOAP Body element provides a simple mechanism for exchanging mandatory information intended for the ultimate SOAP receiver of a SOAP message. Typical uses of SOAP Body include marshalling RPC calls and error reporting.

The SOAP Body element is an immediate child element of a SOAP envelope. If a SOAP Header is present then the SOAP Body MUST immediately follow the SOAP Header, otherwise it MUST be the first immediate child element of the SOAP envelope.

All immediate child elements of the SOAP Body are called SOAP body blocks, and each SOAP body block is encoded as an independent element within the SOAP Body.

The encoding rules for SOAP body blocks are as follows:

1. A SOAP body block is identified by its fully qualified element name, which consists of the namespace URI and the local name. Immediate child elements of the SOAP Body element MAY be namespace-qualified.
2. The SOAP encodingStyle attribute MAY be used to indicate the encoding style used for the SOAP body blocks (see section 4.1.1).

SOAP defines one particular SOAP body block, the SOAP fault, which is used for reporting errors (see section 4.4).

4.3.1 Relationship between SOAP Header and Body

While both SOAP Header and SOAP Body are defined as independent elements, they are in fact related. The relationship

between a SOAP body block and a SOAP header block is as follows: a SOAP body block is semantically equivalent to a SOAP header block targeted at the anonymous actor and with a SOAP mustUnderstand attribute with a value of "1". The anonymous actor is indicated by omitting the actor attribute (see section 4.2.2).

4.4 SOAP Fault

The SOAP fault is used to carry error and/or status information within a SOAP message. If present, the SOAP fault MUST appear as a SOAP body block and MUST NOT appear more than once within a SOAP Body.

The SOAP Fault defines the following four sub-elements:

faultcode

> The faultcode element is intended for use by software to provide an algorithmic mechanism for identifying the fault. The faultcode MUST be present in a SOAP fault and the faultcode value MUST be a qualified name as defined in [8], section 3. SOAP defines a small set of SOAP fault codes covering basic SOAP faults (see section 4.4.1)

faultstring

> The faultstring element is intended to provide a human readable explanation of the fault and is not intended for algorithmic processing. The faultstring element is similar to the 'Reason-Phrase' defined by HTTP (see [5], section 6.1). It MUST be present in a SOAP fault and SHOULD provide at least some information explaining the nature of the fault.

faultactor

> The faultactor element is intended to provide information about which SOAP node on the SOAP message path caused the fault to happen (see section 2). It is similar to the SOAP actor attribute (see section 4.2.2) but instead of indicating the target of a SOAP header block, it indicates the source of the fault. The value of the faultactor attribute is a URI identifying the source. SOAP nodes that do not act as the ultimate SOAP receiver MUST include the faultactor element in the SOAP fault. The ultimate SOAP receiver MAY use the faultactor element to indicate explicitly that it generated the fault (see also the detail element below).

detail

> The detail element is intended for carrying application specific error information related to the SOAP Body. It MUST be present when the contents of the SOAP Body could not be processed successfully . It MUST NOT be used to carry error information about any SOAP header blocks. Detailed error information for SOAP header blocks MUST be carried within the SOAP header blocks themselves, see section 4.4.2 for an example.
>
> The absence of the detail element in the SOAP fault indicates that the fault is not related to the processing of the SOAP Body. This can be used to find out whether the SOAP Body was at least partially processed by the ultimate SOAP receiver before the fault occurred, or not.
>
> All immediate child elements of the detail element are called detail entries, and each detail entry is encoded as an independent element within the detail element.
>
> The encoding rules for detail entries are as follows (see also example 10):
>
> 1. A detail entry is identified by its fully qualified element name, which consists of the namespace URI and the local name. Immediate child elements of the detail element MAY be namespace-qualified.
> 2. The SOAP encodingStyle attribute MAY be used to indicate the encoding style used for the detail entries (see section 4.1.1).

4.4.1 SOAP Fault Codes

The SOAP faultcode values defined in this section MUST be used in the SOAP faultcode element when describing faults defined by this specification. The namespace identifier for these SOAP faultcode values is

"http://www.w3.org/2001/06/soap-envelope". Use of this space is recommended (but not required) in the specification of methods defined outside of the present specification.

The default SOAP faultcode values are defined in an extensible manner that allows for new SOAP faultcode values to be defined while maintaining backwards compatibility with existing SOAP faultcode values. The mechanism used is very similar to the 1xx, 2xx, 3xx etc basic status classes classes defined in HTTP (see [5] section 10). However, instead of integers, they are defined as XML qualified names (see [8] section 3). The character "." (dot) is used as a separator of SOAP faultcode values indicating that what is to the left of the dot is a more generic fault code value than the value to the right. This is illustrated in Example 5.

Example 5

```
Client.Authentication
```

Example of an authentication fault code

The faultcode values defined by SOAP are listed in the following table.

Name	Meaning
VersionMismatch	The processing party found an invalid namespace for the SOAP envelope element (see section 4.1.2)
MustUnderstand	An immediate child element of the SOAP Header element that was either not understood or not obeyed by the processing party contained a SOAP mustUnderstand attribute with a value of "1" (see section 4.2.3)
Client	The Client class of errors indicate that the message was incorrectly formed or did not contain the appropriate information in order to succeed. For example, the message could lack the proper authentication or payment information. It is generally an indication that the message should not be resent without change. See also section 4.4 for a description of the SOAP Fault detail sub-element.
Server	The Server class of errors indicate that the message could not be processed for reasons not directly attributable to the contents of the message itself but rather to the processing of the message. For example, processing could include communicating with an upstream SOAP node, which did not respond. The message may succeed at a later point in time. See also section 4.4 for a description of the SOAP Fault detail sub-element.

4.4.2 MustUnderstand Faults

When a SOAP node generates a MustUnderstand fault, it SHOULD provide, in the generated fault message, header blocks as described below which detail the qualified names (QNames, per the XML Schema Datatypes specification) of the particular header block(s) which were not understood.

Each such header block has a local name of Misunderstood and a namespace name of "http://www.w3.org/2001/06/soap-faults". Each block has an unqualified attribute with a local name of qname whose value is the QName of a header block which the faulting node failed to understand.

For example, the message shown in Example 6 will result in the fault message shown in Example 7 if the recipient of the initial message does not understand the two header elements `abc:Extension1` and `def:Extension2`.

Example 6

```
<env:Envelope xmlns:env='http://www.w3.org/2001/06/soap-envelope'>
  <env:Header>
    <abc:Extension1 xmlns:abc='http://example.org/2001/06/ext'
                    env:mustUnderstand='1' />
    <def:Extension2 xmlns:def='http://example.com/stuff'
                    env:mustUnderstand='1' />
  </env:Header>
  <env:Body>
  . . .
  </env:Body>
```

```
</env:Envelope>
```

SOAP envelope that will cause a SOAP MustUnderstand fault if Extension1 or Extension2 are not understood

Example 7

```
<env:Envelope xmlns:env='http://www.w3.org/2001/06/soap-envelope'
                      xmlns:f='http://www.w3.org/2001/06/soap-faults' >
  <env:Header>
    <f:Misunderstood qname='abc:Extension1'
                                xmlns:abc='http://example.org/2001/06/ext' />
    <f:Misunderstood qname='def:Extension2'
                                xmlns:def='http://example.com/stuff' />
  </env:Header>
  <env:Body>
    <env:Fault>
      <faultcode>MustUnderstand</faultcode>
      <faultstring>One or more mandatory headers not understood</faultstring>
    </env:Fault>
  </env:Body>
</env:Envelope>
```

SOAP fault generated as a result of not understanding Extension1 and Extension2 in Example 6

Note that there is no requirement that the namespace prefix returned in the value of the qname attribute match the namespace prefix of the original header element. Provided the prefix maps to the same namespace name the faulting node may use any prefix.

Note also that there is no guarantee that each MustUnderstand error contains ALL misunderstood header QNames. SOAP nodes MAY generate a fault after the first header block that causes an error containing details about that single header block only, alternatively SOAP nodes MAY generate a combined fault detailing all of the MustUnderstand problems at once.

5. SOAP Encoding

The SOAP encoding style is based on a simple type system that is a generalization of the common features found in type systems in programming languages, databases and semi-structured data. A type either is a simple (scalar) type or is a compound type constructed as a composite of several parts, each with a type. This is described in more detail below. This section defines rules for serialization of a graph of typed objects. It operates on two levels. First, given a schema in any notation consistent with the type system described, a schema for an XML grammar may be constructed. Second, given a type-system schema and a particular graph of values conforming to that schema, an XML instance may be constructed. In reverse, given an XML instance produced in accordance with these rules, and given also the original schema, a copy of the original value graph may be constructed.

The namespace identifier for the elements and attributes defined in this section is "http://www.w3.org/2001/06/soap-encoding". The encoding samples shown assume all namespace declarations are at a higher element level.

Use of the data model and encoding style described in this section is encouraged but not required; other data models and encodings can be used in conjunction with SOAP (see section 4.1.1).

5.1 Rules for Encoding Types in XML

XML allows very flexible encoding of data. SOAP defines a narrower set of rules for encoding. This section defines the encoding rules at a high level, and the next section describes the encoding rules for specific types when they require more detail. The encodings described in this section can be used in conjunction with the mapping of RPC calls and responses specified in Section 7.

To describe encoding, the following terminology is used:

1. A "value" is a string, the name of a measurement (number, date, enumeration, etc.) or a composite of several such primitive values. All values are of specific types.
2. A "simple value" is one without named parts. Examples of simple values are particular strings, integers, enumerated values etc.
3. A "compound value" is an aggregate of relations to other values. Examples of Compound Values are particular purchase orders, stock reports, street addresses, etc.
4. Within a compound value, each related value is potentially distinguished by a role name, ordinal or both. This is called its "accessor." Examples of compound values include particular Purchase Orders, Stock Reports etc. Arrays are also compound values. It is possible to have compound values with several accessors each named the same, as for example, RDF does.
5. An "array" is a compound value in which ordinal position serves as the only distinction among member values.
6. A "struct" is a compound value in which accessor name is the only distinction among member values, and no accessor has the same name as any other.
7. A "simple type" is a class of simple values. Examples of simple types are the classes called "string," "integer," enumeration classes, etc.
8. A "compound type" is a class of compound values. An example of a compound type is the class of purchase order values sharing the same accessors (shipTo, totalCost, etc.) though with potentially different values (and perhaps further constrained by limits on certain values).
9. Within a compound type, if an accessor has a name that is distinct within that type but is not distinct with respect to other types, that is, the name plus the type together are needed to make a unique identification, the name is called "locally scoped." If however the name is based in part on a Uniform Resource Identifier, directly or indirectly, such that the name alone is sufficient to uniquely identify the accessor irrespective of the type within which it appears, the name is called "universally scoped."
10. Given the information in the schema relative to which a graph of values is serialized, it is possible to determine that some values can only be related by a single instance of an accessor. For others, it is not possible to make this determination. If only one accessor can reference it, a value is considered "single-reference". If referenced by more than one, actually or potentially, it is "multi-reference." Note that it is possible for a certain value to be considered "single-reference" relative to one schema and "multi-reference" relative to another.
11. Syntactically, an element may be "independent" or "embedded." An independent element is any element appearing at the top level of a serialization. All others are embedded elements.

Although it is possible to use the xsi:type attribute such that a graph of values is self-describing both in its structure and the types of its values, the serialization rules permit that the types of values MAY be determinate only by reference to a schema. Such schemas MAY be in the notation described by "XML Schema Part 1: Structures" [10] and "XML Schema Part 2: Datatypes" [11] or MAY be in any other notation. Note also that, while the serialization rules apply to compound types other than arrays and structs, many schemas will contain only struct and array types.

The rules for serialization are as follows:

1. All values are represented as element content. A multi-reference value MUST be represented as the content of an independent element. A single-reference value SHOULD not be (but MAY be).
2. For each element containing a value, the type of the value MUST be represented by at least one of the following conditions: (a) the containing element instance contains an xsi:type attribute, (b) the containing element instance is itself contained within an element containing a (possibly defaulted) enc:arrayType attribute or (c) or the name of the element bears a definite relation to the type, that type then determinable from a schema.

3. A simple value is represented as character data, that is, without any subelements. Every simple value must have a type that is either listed in the XML Schemas Specification, part 2 [11] or whose source type is listed therein (see also section 5.2).

4. A Compound Value is encoded as a sequence of elements, each accessor represented by an embedded element whose name corresponds to the name of the accessor. Accessors whose names are local to their containing types have unqualified element names; all others have qualified names (see also section 5.4).

5. A multi-reference simple or compound value is encoded as an independent element containing a local, unqualified attribute named "id" and of type "ID" per the XML Specification [7]. Each accessor to this value is an empty element having a local, unqualified attribute named "href" and of type "uri-reference" per the XML Schema Specification [11], with a "href" attribute value of a URI fragment identifier referencing the corresponding independent element.

6. Strings and byte arrays are represented as multi-reference simple types, but special rules allow them to be represented efficiently for common cases (see also section 5.2.1 and 5.2.3). An accessor to a string or byte-array value MAY have an attribute named "id" and of type "ID" per the XML Specification [7]. If so, all other accessors to the same value are encoded as empty elements having a local, unqualified attribute named "href" and of type "uri-reference" per the XML Schema Specification [11], with a "href" attribute value of a URI fragment identifier referencing the single element containing the value.

7. It is permissible to encode several references to a value as though these were references to several distinct values, but only when from context it is known that the meaning of the XML instance is unaltered.

8. Arrays are compound values (see also section 5.4.2). SOAP arrays are defined as having a type of "enc:Array" or a type derived there from.

   SOAP arrays have one or more dimensions (rank) whose members are distinguished by ordinal position. An array value is represented as a series of elements reflecting the array, with members appearing in ascending ordinal sequence. For multi-dimensional arrays the dimension on the right side varies most rapidly. Each member element is named as an independent element (see rule 2).

   SOAP arrays can be single-reference or multi-reference values, and consequently may be represented as the content of either an embedded or independent element.

   SOAP arrays MUST contain a "enc:arrayType" attribute whose value specifies the type of the contained elements as well as the dimension(s) of the array. The value of the "enc:arrayType" attribute is defined as follows:

arrayTypeValue	=	atype asize
atype	=	QName *(rank)
rank	=	"[" *(",") "]"
asize	=	"[" #length "]"
length	=	1*DIGIT

   The "atype" construct is the type name of the contained elements expressed as a QName as would appear in the "type" attribute of an XML Schema element declaration and acts as a type constraint (meaning that all values of contained elements are asserted to conform to the indicated type; that is, the type cited in enc:arrayType must be the type or a supertype of every array member). In the case of arrays of arrays or "jagged arrays", the type component is encoded as the "innermost" type name followed by a rank construct for each level of nested arrays starting from 1. Multi-dimensional arrays are encoded using a comma for each dimension starting from 1.

   The "asize" construct contains a comma separated list of zero, one, or more integers indicating the lengths of each dimension of the array. A value of zero integers indicates that no particular quantity is asserted but that the size may be determined by inspection of the actual members.

   For example, an array with 5 members of type array of integers would have an arrayTypeValue value of "int[][5]" of which the atype value is "int[]" and the asize value is "[5]". Likewise, an array with 3 members of type two-dimensional arrays of integers would have an arrayTypeValue value of "int[,][3]" of which the atype value is

"int[,]" and the asize value is "[3]".

A SOAP array member MAY contain a "enc:offset" attribute indicating the offset position of that item in the enclosing array. This can be used to indicate the offset position of a partially represented array (see section 5.4.2.1). Likewise, an array member MAY contain a "enc:position" attribute indicating the position of that item in the enclosing array. This can be used to describe members of sparse arrays (see section 5.4.2.2). The value of the "enc:offset" and the "enc:position" attribute is defined as follows:

arrayPoint = "[" #length "]"

with offsets and positions based at 0.

9. A NULL value or a default value MAY be represented by omission of the accessor element. A NULL value MAY also be indicated by an accessor element containing the attribute xsi:null with value '1' or possibly other application-dependent attributes and values.

Note that rule 2 allows independent elements and also elements representing the members of arrays to have names which are not identical to the type of the contained value.

5.2 Simple Types

For simple types, SOAP adopts all the types found in the section "Built-in datatypes" of the "XML Schema Part 2: Datatypes" Specification [11], both the value and lexical spaces. Examples include:

Type	Example
int	58502
float	314159265358979E+1
negativeInteger	-32768
string	Louis "Satchmo" Armstrong

The datatypes declared in the XML Schema specification may be used directly in element schemas. Types derived from these may also be used. For example, for the following schema:

Example 7

```
<!-- schema document -->
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" >

  <xs:element name="age" type="xs:int" />
  <xs:element name="height" type="xs:float" />
  <xs:element name="displacement" type="xs:negativeInteger" />
  <xs:element name="color" >
    <xs:simpleType base="xsd:string">
      <xs:restriction base="xs:string" >
        <xs:enumeration value="Green"/>
        <xs:enumeration value="Blue"/>
          </xs:restriction>
    </xs:simpleType>
  </xs:element>

</xs:schema>
```

Schema with simple types

the following elements would be valid instances:

Example 8

```
<!-- Example instance elements -->
```

```
<age>45</age>
<height>5.9</height>
<displacement>-450</displacement>
<color>Blue</color>
```

Message fragment corresponding to the schema in Example 7

All simple values MUST be encoded as the content of elements whose type is either defined in "XML Schema Part 2: Datatypes" Specification [11], or is based on a type found there by using the mechanisms provided in the XML Schema specification.

If a simple value is encoded as an independent element or member of a heterogenous array it is convenient to have an element declaration corresponding to the datatype. Because the "XML Schema Part 2: Datatypes" Specification [11] includes type definitions but does not include corresponding element declarations, the enc schema and namespace declares an element for every simple datatype. These MAY be used.

Example 9

```
<enc:int xmlns:enc="http://www.w3.org/2001/06/soap-encoding" id="int1">45</enc:int>
```

5.2.1 Strings

The datatype "string" is defined in "XML Schema Part 2: Datatypes" Specification [11]. Note that this is not identical to the type called "string" in many database or programming languages, and in particular may forbid some characters those languages would permit. (Those values must be represented by using some datatype other than xsd:string.)

A string MAY be encoded as a single-reference or a multi-reference value.

The containing element of the string value MAY have an "id" attribute. Additional accessor elements MAY then have matching "href" attributes.

For example, two accessors to the same string could appear, as follows:

Example 10

```
<greeting id="String-0">Hello</greeting>
<salutation href="#String-0"/>
```

Two accessors for the same string

However, if the fact that both accessors reference the same instance of the string (or subtype of string) is immaterial, they may be encoded as two single-reference values as follows:

Example 11

```
<greeting>Hello</greeting>
<salutation>Hello</salutation>
```

Two accessors for the same string

Schema fragments for these examples could appear similar to the following:

Example 12

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
           xmlns:enc="http://www.w3.org/2001/06/soap-encoding" >

  <xs:import namespace="http://www.w3.org/2001/06/soap-encoding" />

  <xs:element name="greeting" type="enc:string" />
  <xs:element name="salutation" type="enc:string" />

</xs:schema>
```

Schema for Example 11

(In this example, the type enc:string is used as the element's type as a convenient way to declare an element whose datatype is "xsd:string" and which also allows an "id" and "href" attribute. See the SOAP Encoding schema for the exact definition. Schemas MAY use these declarations from the SOAP Encoding schema but are not required to.)

5.2.2 Enumerations

The "XML Schema Part 2: Datatypes" Specification [11] defines a mechanism called "enumeration." The SOAP data model adopts this mechanism directly. However, because programming and other languages often define enumeration somewhat differently, we spell-out the concept in more detail here and describe how a value that is a member of an enumerated list of possible values is to be encoded. Specifically, it is encoded as the name of the value.

"Enumeration" as a concept indicates a set of distinct names. A specific enumeration is a specific list of distinct values appropriate to the base type. For example the set of color names ("Green", "Blue", "Brown") could be defined as an enumeration based on the string built-in type. The values ("1", "3", "5") are a possible enumeration based on integer, and so on. "XML Schema Part 2: Datatypes" [11] supports enumerations for all of the simple types except for boolean. The language of "XML Schema Part 1: Structures" Specification [10] can be used to define enumeration types. If a schema is generated from another notation in which no specific base type is applicable, use "string". In the following schema example "EyeColor" is defined as a string with the possible values of "Green", "Blue", or "Brown" enumerated, and instance data is shown accordingly.

Example 13

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
                   xmlns:tns="http://example.org/2001/06/samples"
                   targetNamespace="http://example.org/2001/06/samples" >

  <xs:element name="EyeColor" type="tns:EyeColor" />
  <xs:simpleType name="EyeColor" >
    <xs:restriction base="xs:string" >
      <xs:enumeration value="Green" />
      <xs:enumeration value="Blue" />
      <xs:enumeration value="Brown" />
    </xs:restriction>
  </xs:simpleType>

</xs:schema>
```

Schema with enumeration

Example 14

```
<p:EyeColor xmlns:p="http://example.org/2001/06/samples" >Brown</p:EyeColor>
```

Message fragment corresponding to the schema in Example 13

5.2.3 Array of Bytes

An array of bytes MAY be encoded as a single-reference or a multi-reference value. The rules for an array of bytes are similar to those for a string.

In particular, the containing element of the array of bytes value MAY have an "id" attribute. Additional accessor elements MAY then have matching "href" attributes.

The recommended representation of an opaque array of bytes is the 'base64' encoding defined in XML Schemas [10][11], which uses the base64 encoding algorithm defined in 2045 [13]. However, the line length restrictions that normally apply to base64 data in MIME do not apply in SOAP. A "enc:base64" subtype is supplied for use with SOAP.

Example 15

```
<picture xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xmlns:enc="http://www.w3.org/2001/06/soap-encoding"
         xsi:type="enc:base64" >
   aG93IG5vDyBicm73biBjb3cNCg==
</picture>
```

Image with base64 encoding

5.3 Polymorphic Accessor

Many languages allow accessors that can polymorphically access values of several types, each type being available at run time. A polymorphic accessor instance MUST contain an "xsi:type" attribute that describes the type of the actual value.

For example, a polymorphic accessor named "cost" with a value of type "xsd:float" would be encoded as follows:

Example 16

```
<cost xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:xs="http://www.w3.org/2001/XMLSchema"
          xsi:type="xs:float">29.95</cost>
```

Polymorphic accessor

as contrasted with a cost accessor whose value's type is invariant, as follows:

Example 17

```
<cost>29.95</cost>
```

Accessor whose value type is invariant

5.4 Compound types

SOAP defines types corresponding to the following structural patterns often found in programming languages:

Struct

A "struct" is a compound value in which accessor name is the only distinction among member values, and no accessor has the same name as any other.

Array

An "array" is a compound value in which ordinal position serves as the only distinction among member values.

SOAP also permits serialization of data that is neither a Struct nor an Array, for example data such as is found in a Directed-Labeled-Graph Data Model in which a single node has many distinct accessors, some of which occur more than once. SOAP serialization does not require that the underlying data model make an ordering distinction among accessors, but if such an order exists, the accessors MUST be encoded in that sequence.

5.4.1 Compound Values, Structs and References to Values

The members of a Compound Value are encoded as accessor elements. When accessors are distinguished by their name (as for example in a struct), the accessor name is used as the element name. Accessors whose names are local to their containing types have unqualified element names; all others have qualified names.

The following is an example of a struct of type "Book":

Example 18

```
<e:Book xmlns:e="http://example.org/2001/06/books" >
   <author>Henry Ford</author>
   <preface>Prefactory text</preface>
   <intro>This is a book.</intro>
</e:Book>
```

Book structure

And this is a schema fragment describing the above structure:

Example 19

```
<xs:element name="Book"
            xmlns:xs='http://www.w3.org/2001/XMLSchema' >
  <xs:complexType>
    <xs:sequence>
      <xs:element name="author" type="xs:string" />
      <xs:element name="preface" type="xs:string" />
      <xs:element name="intro" type="xs:string" />
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

Schema for Example 18

Below is an example of a type with both simple and complex members. It shows two levels of referencing. Note that the "href" attribute of the "Author" accessor element is a reference to the value whose "id" attribute matches. A similar construction appears for the "Address".

Example 20

```
<e:Book xmlns:e="http://example.org/2001/06/books" >
  <title>My Life and Work</title>
  <author href="#Person-1"/>
</e:Book>
<e:Person xmlns:e="http://example.org/2001/06/books"
          id="Person-1" >
  <name>Henry Ford</name>
  <address href="#Address-2"/>
</e:Person>
<e:Address xmlns:e="http://example.org/2001/06/books"
           id="Address-2" >
  <email>mailto:henryford@hotmail.com</email>
  <web>http://www.henryford.com</web>
</e:Address>
```

Book with muli-reference addresses

The form above is appropriate when the "Person" value and the "Address" value are multi-reference. If these were instead both single-reference, they SHOULD be embedded, as follows:

Example 21

```
<e:Book xmlns:e="http://example.org/2001/06/books" >
   <title>My Life and Work</title>
   <author>
       <name>Henry Ford</name>
       <address>
          <email>mailto:henryford@hotmail.com</email>
          <web>http://www.henryford.com</web>
       </address>
   </author>
</e:Book>
```

Book with single-reference addresses

If instead there existed a restriction that no two persons can have the same address in a given instance and that an address can be either a Street-address or an Electronic-address, a Book with two authors would be encoded as follows:

Example 22

```
<e:Book xmlns:e="http://example.org/2001/06/books" >
   <title>My Life and Work</title>
   <firstauthor href="#Person-1"/>
   <secondauthor href="#Person-2"/>
</e:Book>
<e:Person xmlns:e="http://example.org/2001/06/books"
          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          id="Person-1" >
   <name>Henry Ford</name>
   <address xsi:type="e:ElectronicAddressType">
       <email>mailto:henryford@hotmail.com</email>
       <web>http://www.henryford.com</web>
   </address>
</e:Person>
<e:Person xmlns:e="http://example.org/2001/06/books"
          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          id="Person-2">
   <name>Samuel Crowther</name>
   <address xsi:type="e:StreetAddressType">
       <street>Martin Luther King Rd</street>
       <city>Raleigh</city>
       <state>North Carolina</state>
   </address>
</e:Person>
```

Book with two authors having different addresses

Serializations can contain references to values not in the same resource:

Example 23

```
<e:Book xmlns:e="http://example.org/2001/06/books" >
   <title>Paradise Lost</title>
   <firstAuthor href="http://www.dartmouth.edu/~milton/" />
</e:Book>
```

Book with external references

And this is a schema fragment describing the above structures:

Example 24

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
           xmlns:tns="http://example.org/2001/06/books"
                   targetNamespace="http://example.org/2001/06/books" >

  <xs:element name="Book" type="tns:BookType" />
  <xs:complexType name="BookType" >
    <xs:annotation>
          <xs:documentation>
            <info>
        Either the following group must occur or else the
        href attribute must appear, but not both.
                <info>
          </xs:documentation>
    </xs:annotation>
    <xs:sequence minOccurs="0" maxOccurs="1" >
      <xs:element name="title" type="xs:string" />
```

```
      <xs:element name="firstAuthor" type="tns:PersonType" />
      <xs:element name="secondAuthor" type="tns:PersonType" />
    </xs:sequence>
    <xs:attribute name="href" type="xs:anyURI" />
    <xs:attribute name="id" type="xs:ID" />
    <xs:anyAttribute namespace="##other" />
  </xs:complexType>

  <xs:element name="Person" type="tns:PersonType" />
  <xs:complexType name="PersonType" >
    <xs:annotation>
          <xs:documentation>
            <info>
        Either the following group must occur or else the
        href attribute must appear, but not both.
                <info>
          </xs:documentation>
        </xs:annotation>
    <xs:sequence minOccurs="0" maxOccurs="1" >
      <xs:element name="name" type="xs:string" />
      <xs:element name="address" type="tns:AddressType" />
    </xs:sequence>
    <xs:attribute name="href" type="xs:anyURI" />
    <xs:attribute name="id" type="xs:ID" />
    <xs:anyAttribute namespace="##other" />
  </xs:complexType>

  <xs:element name="Address" base="tns:AddressType" />
  <xs:complexType name="AddressType" abstract="true" >
    <xs:annotation>
          <xs:documentation>
            <info>
        Either one of the following sequences must occur or
                else the href attribute must appear, but not both.
                </info>
          </xs:documentation>
        </xs:annotation>
    <xs:choice>
          <xs:sequence minOccurs="0" maxOccurs="1" >
            <xs:element name="email" type="xs:string" />
                <xs:element name="web" type="xs:anyURI" />
          </xs:sequence>
          <xs:sequence minOccurs='0' maxOccurs='1' >
        <xs:element name="street" type="xs:string" />
        <xs:element name="city" type="xs:string" />
        <xs:element name="state" type="xs:string"/>
          </xs:sequence>
        </xs:choice>
    <xs:attribute name="href" type="xs:anyURI"/>
    <xs:attribute name="id" type="xs:ID"/>
    <xs:anyAttribute namespace="##other"/>
  </xs:complexType>

  <xs:complexType name="StreetAddressType">
```

```
  <xs:annotation>
        <xs:documentation>
          <info>
      Either the second sequence in the following group
              must occur or else the href attribute must appear,
              but not both.
              </info>
        </xs:documentation>
      </xs:annotation>
      <xs:complexContent>
        <xs:restriction base="tns:AddressType" >
          <xs:sequence>
            <xs:sequence minOccurs="0" maxOccurs="0" >
              <xs:element name="email" type="xs:string" />
                  <xs:element name="web" type="xs:anyURI" />
            </xs:sequence>
        <xs:sequence minOccurs="0" maxOccurs="1">
          <xs:element name="street" type="xs:string" />
          <xs:element name="city" type="xs:string" />
          <xs:element name="state" type="xs:string"/>
        </xs:sequence>
              </xs:sequence>
      <xs:attribute name="href" type="xs:anyURI"/>
      <xs:attribute name="id" type="xs:ID"/>
      <xs:anyAttribute namespace="##other"/>
        </xs:restriction>
      </xs:complexContent>
</xs:complexType>

<xs:complexType name="ElectronicAddressType">
  <xs:annotation>
        <xs:documentation>
          <info>
      Either the first sequence in the following group
              must occur or else the href attribute must appear,
              but not both.
              </info>
        </xs:documentation>
      </xs:annotation>
      <xs:complexContent>
        <xs:restriction base="tns:AddressType" >
          <xs:sequence>
        <xs:sequence minOccurs="0" maxOccurs="1">
          <xs:element name="email" type="xs:string" />
          <xs:element name="web" type="xs:anyURI" />
        </xs:sequence>
                <xs:sequence minOccurs="0" maxOccurs="0">
          <xs:element name="street" type="xs:string" />
          <xs:element name="city" type="xs:string" />
          <xs:element name="state" type="xs:string"/>
        </xs:sequence>
              </xs:sequence>
          <xs:attribute name="href" type="xs:anyURI"/>
      <xs:attribute name="id" type="xs:ID"/>
```

```
        <xs:anyAttribute namespace="##other"/>
          </xs:restriction>
        </xs:complexContent>
  </xs:complexType>

</xs:schema>
```

Schema for example 22

5.4.2 Arrays

SOAP arrays are defined as having a type of enc:Array or a derived type having that type in its derivation hierarchy (see also rule 8). Such derived types would be restrictions of the enc:Array type and could be used to represent, for example, arrays limited to integers or arrays of some user-defined enumeration. Arrays are represented as element values, with no specific constraint on the name of the containing element (just as values generally do not constrain the name of their containing element). The elements which make up the array can themselves can be of any type, including nested arrays.

The representation of the value of an array is an ordered sequence of elements constituting the items of the array. Within an array value, element names are not significant for distinguishing accessors. Elements may have any name. In practice, elements will frequently be named so that their declaration in a schema suggests or determines their type. As with compound types generally, if the value of an item in the array is a single-reference value, the item contains its value. Otherwise, the item references its value via an "href" attribute.

The following example is a schema fragment and an array containing integer array members:

Example 25

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
           xmlns:enc="http://www.w3.org/2001/06/soap-encoding" >
  <xs:import namespace="http://www.w3.org/2001/06/soap-encoding" />
  <xs:element name="myFavoriteNumbers" type="enc:Array" />
</xs:schema>
```

Schema declaring an array of integers

Example 26

```
<myFavoriteNumbers xmlns:xs="http://www.w3.org/2001/XMLSchema"
                   xmlns:enc="http://www.w3.org/2001/06/soap-encoding"
                   enc:arrayType="xs:int[2]" >
  <number>3</number>
  <number>4</number>
</myFavoriteNumbers>
```

Array conforming to the schema in Example 25

In that example, the array myFavoriteNumbers contains several members each of which is a value of type xs:int. This can be determined by inspection of the enc:arrayType attribute. Note that the enc:Array type allows both unqualified element names and qualified element names from any namespace. These convey no type information, so when used they must either have an xsi:type attribute or the containing element must have a enc:arrayType attribute. Naturally, types derived from enc:Array may declare local elements, with type information.

As previously noted, the enc schema contains declarations of elements with names corresponding to each simple type in the "XML Schema Part 2: Datatypes" Specification [11]. It also contains a declaration for "Array". Using these, we might write:

Example 27

```
<enc:Array xmlns:enc="http://www.w3.org/2001/06/soap-encoding"
           xmlns:xs="http://www.w3.org/2001/XMLSchema"
```

```
           enc:ArrayType="xs:int[2]" >
  <enc:int>3</enc:int>
  <enc:int>4</enc:int>
</enc:Array>
```

Using the enc:Array element

Arrays can contain instances of any subtype of the specified arrayType. That is, the members may be of any type that is substitutable for the type specified in the arrayType attribute, according to whatever substitutability rules are expressed in the schema. So, for example, an array of integers can contain any type derived from integer (for example "int" or any user-defined derivation of integer). Similarly, an array of "address" might contain a restricted or extended type such as "internationalAddress". Because the supplied enc:Array type admits members of any type, arbitrary mixtures of types can be contained unless specifically limited by use of the arrayType attribute.

Types of member elements can be specified using the xsi:type attribute in the instance, or by declarations in the schema of the member elements, as the following two arrays demonstrate respectively:

Example 28

```
<enc:Array xmlns:enc="http://www.w3.org/2001/06/soap-encoding"
           xmlns:xs="http://www.w3.org/2001/XMLSchema"
                   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
           enc:arrayType="xs:anyType[4]">
   <thing xsi:type="xs:int">12345</thing>
   <thing xsi:type="xs:decimal">6.789</thing>
   <thing xsi:type="xs:string">
      Of Mans First Disobedience, and the Fruit
      Of that Forbidden Tree, whose mortal tast
      Brought Death into the World, and all our woe,
   </thing>
   <thing xsi:type="xs:anyURI">
      http://www.dartmouth.edu/~milton/reading_room/
   </thing>
</enc:Array>
```

Array with elements of varying types

example 29

```
<enc:Array xmlns:xs="http://www.w3.org/2001/XMLSchema"
           xmlns:enc="http://www.w3.org/2001/06/soap-encoding"
           enc:arrayType="xs:anyType[4]" >
   <enc:int>12345</enc:int>
   <enc:decimal>6.789</enc:decimal>
   <enc:string>
      Of Mans First Disobedience, and the Fruit
      Of that Forbidden Tree, whose mortal tast
      Brought Death into the World, and all our woe,
   </enc:string>
   <enc:anyURI>
      http://www.dartmouth.edu/~milton/reading_room/
   </enc:anyURI >
</enc:Array>
```

Array with elements of varying types

Array values may be structs or other compound values. For example an array of "xyz:Order" structs :

Example 30

```
<enc:Array xmlns:enc="http://www.w3.org/2001/06/soap-encoding"
           xmlns:xyz="http://example.org/2001/06/Orders"
           enc:arrayType="xyz:Order[2]">
   <Order>
       <Product>Apple</Product>
       <Price>1.56</Price>
   </Order>
   <Order>
       <Product>Peach</Product>
       <Price>1.48</Price>
   </Order>
</enc:Array>
```

Arrays containing structs and other compound values

Arrays may have other arrays as member values. The following is an example of an array of two arrays, each of which is an array of strings.

Example 31

```
<enc:Array xmlns:xs="http://www.w3.org/2001/XMLSchema"
           xmlns:enc="http://www.w3.org/2001/06/soap-encoding"
           enc:arrayType="xs:string[][2]" >
   <item href="#array-1"/>
   <item href="#array-2"/>
</enc:Array>
<enc:Array xmlns:xs="http://www.w3.org/2001/XMLSchema"
           xmlns:enc="http://www.w3.org/2001/06/soap-encoding"
           id="array-1"
           enc:arrayType="xs:string[2]">
   <item>r1c1</item>
   <item>r1c2</item>
   <item>r1c3</item>
</enc:Array>
<enc:Array xmlns:xs="http://www.w3.org/2001/XMLSchema"
           xmlns:enc="http://www.w3.org/2001/06/soap-encoding"
           id="array-2"
           enc:arrayType="xs:string[2]">
   <item>r2c1</item>
   <item>r2c2</item>
</enc:Array>
```

Array containing other arrays

The element containing an array value does not need to be named "enc:Array". It may have any name, provided that the type of the element is either enc:Array or is derived from enc:Array by restriction. For example, the following is a fragment of a schema and a conforming instance array:

Example 32

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
           xmlns:enc="http://www.w3.org/2001/06/soap-encoding"
           xmlns:tns="http://example.org/2001/06/numbers"
           targetNamespace="http://example.org/2001/06/numbers" >

  <xs:simpleType name="phoneNumberType" >
    <xs:restriction base="xs:string" />
  </xs:simpleType>
```

```
  <xs:element name="ArrayOfPhoneNumbers" type="tns:ArrayOfPhoneNumbersType" />

  <xs:complexType name="ArrayOfPhoneNumbersType" >
    <xs:complexContent>
      <xs:restriction base="enc:Array" >
            <xs:sequence>
          <xs:element name="phoneNumber" type="tns:phoneNumberType"
maxOccurs="unbounded" />
            </xs:sequence>
        <xs:attributeGroup ref="enc:arrayAttributes" />
        <xs:attributeGroup ref="enc:commonAttributes" />
      </xs:restriction>
    </xs:complexContent>
  </xs:complexType>

</xs:schema>
```

Schema for an array

Example 33

```
<abc:ArrayOfPhoneNumbers xmlns:abc="http://example.org/2001/06/numbers"
                         xmlns:enc="http://www.w3.org/2001/06/soap-encoding"
                         enc:arrayType="abc:phoneNumberType[2]" >
   <phoneNumber>206-555-1212</phoneNumber>
   <phoneNumber>1-888-123-4567</phoneNumber>
</abc:ArrayOfPhoneNumbers>
```

Array conforming to the schema in Example 32

Arrays may be multi-dimensional. In this case, more than one size will appear within the asize part of the arrayType attribute:

Example 34

```
<enc:Array xmlns:xs="http://www.w3.org/2001/XMLSchema"
           xmlns:enc="http://www.w3.org/2001/06/soap-encoding"
           enc:arrayType="xs:string[2,3]" >
   <item>r1c1</item>
   <item>r1c2</item>
   <item>r1c3</item>
   <item>r2c1</item>
   <item>r2c2</item>
   <item>r2c3</item>
</enc:Array>
```

Multi-dimensonal array

While the examples above have shown arrays encoded as independent elements, array values MAY also appear embedded and SHOULD do so when they are known to be single reference.

The following is an example of a schema fragment and an array of phone numbers embedded in a struct of type "Person" and accessed through the accessor "phone-numbers":

Example 34

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
           xmlns:enc="http://www.w3.org/2001/06/soap-encoding"
           xmlns:tns="http://example.org/2001/06/numbers"
                   targetNamespace="http://example.org/2001/06/numbers" >
```

```
  <xs:import namespace="http://www.w3.org/2001/06/soap-encoding" />

  <xs:simpleType name="phoneNumberType" >
    <xs:restriction base="xs:string" />
  </xs:simpleType>


  <xs:element name="ArrayOfPhoneNumbers" type="tns:ArrayOfPhoneNumbersType" />

  <xs:complexType name="ArrayOfPhoneNumbersType" >
    <xs:complexContent>
      <xs:restriction base="enc:Array" >
            <xs:sequence>
          <xs:element name="phoneNumber" type="tns:phoneNumberType" 
maxOccurs="unbounded" />
            </xs:sequence>
        <xs:attributeGroup ref="enc:arrayAttributes" />
        <xs:attributeGroup ref="enc:commonAttributes" />
      </xs:restriction>
    </xs:complexContent>
  </xs:complexType>

  <xs:element name="Person">
    <xs:complexType>
          <xs:sequence>
        <xs:element name="name" type="xs:string" />
        <xs:element name="phoneNumbers" type="tns:ArrayOfPhoneNumbersType" />
      </xs:sequence>
    </xs:complexType>
  </xs:element>

</xs:schema>
```

Schema fragment for array of phone numbers embedded in a struct

Example 35

```
<def:Person xmlns:def="http://example.org/2001/06/numbers"
            xmlns:enc="http://www.w3.org/2001/06/soap-encoding" >
   <name>John Hancock</name>
   <phoneNumbers enc:arrayType="def:phoneNumber[2]">
       <phoneNumber>206-555-1212</phoneNumber>
       <phoneNumber>1-888-123-4567</phoneNumber>
   </phoneNumbers>
</def:Person>
```

Array of phone numbers embedded in a struct conforming to the schema in Example 34

Here is another example of a single-reference array value encoded as an embedded element whose containing element name is the accessor name:

Example 36

```
<xyz:PurchaseOrder xmlns:xyz="http://example.org/2001/06/Orders" >
   <CustomerName>Henry Ford</CustomerName>
   <ShipTo>
```

```
        <Street>5th Ave</Street>
        <City>New York</City>
        <State>NY</State>
        <Zip>10010</Zip>
    </ShipTo>
    <PurchaseLineItems xmlns:enc="http://www.w3.org/2001/06/soap-encoding"
                       enc:arrayType="xyz:Order[2]">
        <Order>
            <Product>Apple</Product>
            <Price>1.56</Price>
        </Order>
        <Order>
            <Product>Peach</Product>
            <Price>1.48</Price>
        </Order>
    </PurchaseLineItems>
</xyz:PurchaseOrder>
```

Single-reference array encoded as en embedded element

5.4.2.1 Partially Transmitted Arrays

SOAP provides support for partially transmitted arrays, known as "varying" arrays in some contexts [12]. A partially transmitted array indicates in an "enc:offset" attribute the zero-origin offset of the first element transmitted. If omitted, the offset is taken as zero.

The following is an example of an array of size five that transmits only the third and fourth element counting from zero:

Example 37

```
<enc:Array xmlns:enc="http://www.w3.org/2001/06/soap-encoding"
           xmlns:xs="http://www.w3.org/2001/XMLSchema"
           enc:arrayType="xs:string[6]"
           enc:offset="[3]" >
  <item>The fourth element</item>
  <item>The fifth element</item>
</enc:Array>
```

Array of size five that transmits only the third and fourth element

5.4.2.2 Sparse Arrays

SOAP provides support for sparse arrays. Each element representing a member value contains a "enc:position" attribute that indicates its position within the array. The following is an example of a sparse array of two-dimensional arrays of strings. The size is 4 but only position 2 is used:

Example 38

```
<enc:Array xmlns:enc="http://www.w3.org/2001/06/soap-encoding"
           xmlns:xs="http://www.w3.org/2001/XMLSchema"
           enc:arrayType="xs:string[,][4]" >
   <enc:Array href="#array-1" enc:position="[2]" />
</enc:Array>
<enc:Array id="array-1"
           enc:arrayType="xs:string[10,10]" >
   <item enc:position="[2,2]">Third row, third col</item>
   <item enc:position="[7,2]">Eighth row, third col</item>
</enc:Array>
```

Sparse array

If the only reference to array-1 occurs in the enclosing array, this example could also have been encoded as follows:

Example 39

```
<enc:Array xmlns:enc="http://www.w3.org/2001/06/soap-encoding"
           xmlns:xs="http://www.w3.org/2001/XMLSchema"
           enc:arrayType="xs:string[,][4]" >
  <enc:Array enc:position="[2]" enc:arrayType="xs:string[10,10]" >
    <item enc:position="[2,2]">Third row, third col</item>
    <item enc:position="[7,2]">Eighth row, third col</item>
  </enc:Array>
</enc:Array>
```

Another sparse array

5.4.3 Generic Compound Types

The encoding rules just cited are not limited to those cases where the accessor names are known in advance. If accessor names are known only by inspection of the immediate values to be encoded, the same rules apply, namely that the accessor is encoded as an element whose name matches the name of the accessor, and the accessor either contains or references its value. Accessors containing values whose types cannot be determined in advance MUST always contain an appropriate xsi:type attribute giving the type of the value.

Similarly, the rules cited are sufficient to allow serialization of compound types having a mixture of accessors distinguished by name and accessors distinguished by both name and ordinal position. (That is, having some accessors repeated.) This does not require that any schema actually contain such types, but rather says that if a type-model schema does have such types, a corresponding XML syntactic schema and instance may be generated.

Example 40

```
<xyz:PurchaseOrder xmlns:xyz="http://example.org/2001/06/Orders" >
  <CustomerName>Henry Ford</CustomerName>
  <ShipTo>
    <Street>5th Ave</Street>
    <City>New York</City>
    <State>NY</State>
    <Zip>10010</Zip>
  </ShipTo>
  <PurchaseLineItems>
    <Order>
      <Product>Apple</Product>
      <Price>1.56</Price>
    </Order>
    <Order>
      <Product>Peach</Product>
      <Price>1.48</Price>
    </Order>
  </PurchaseLineItems>
</xyz:PurchaseOrder>
```

Generic compound types

Similarly, it is valid to serialize a compound value that structurally resembles an array but is not of type (or subtype) enc:Array. For example:

Example 41

```
<PurchaseLineItems>
    <Order>
        <Product>Apple</Product>
        <Price>1.56</Price>
```

```
    </Order>
    <Order>
        <Product>Peach</Product>
        <Price>1.48</Price>
    </Order>
</PurchaseLineItems>
```

Compound value

5.5 Default Values

An omitted accessor element implies either a default value or that no value is known. The specifics depend on the accessor, method, and its context. For example, an omitted accessor typically implies a Null value for polymorphic accessors (with the exact meaning of Null accessor-dependent). Likewise, an omitted Boolean accessor typically implies either a False value or that no value is known, and an omitted numeric accessor typically implies either that the value is zero or that no value is known.

5.6 SOAP root Attribute

The SOAP root attribute can be used to label serialization roots that are not true roots of an object graph so that the object graph can be deserialized. The attribute can have one of two values, either "1" or "0". True roots of an object graph have the implied attribute value of "1". Serialization roots that are not true roots can be labeled as serialization roots with an attribute value of "1" An element can explicitly be labeled as not being a serialization root with a value of "0".

The SOAP root attribute MAY appear on any subelement within the SOAP Header and SOAP Body elements. The attribute does not have a default value.

6. Using SOAP in HTTP

This section describes how to use SOAP within HTTP with or without using the experimental HTTP Extension Framework. Binding SOAP to HTTP provides the advantage of being able to use the formalism and decentralized flexibility of SOAP with the rich feature set of HTTP. Carrying SOAP in HTTP does not mean that SOAP overrides existing semantics of HTTP but rather that SOAP over HTTP inherits HTTP semantics.

SOAP naturally follows the HTTP request/response message model by providing a SOAP request message in a HTTP request and SOAP response message in a HTTP response. Note, however, that SOAP intermediaries are NOT the same as HTTP intermediaries. That is, an HTTP intermediary addressed with the HTTP Connection header field cannot be expected to inspect or process the SOAP entity body carried in the HTTP request.

HTTP applications MUST use the media type "text/xml" according to RFC 2376 [3] when including SOAP messages in HTTP exchanges.

6.1 SOAP HTTP Request

Although SOAP might be used in combination with a variety of HTTP request methods, this binding only defines SOAP within HTTP POST requests (see section 7 for how to use SOAP for RPC and section 6.3 for how to use the HTTP Extension Framework).

6.1.1 The SOAPAction HTTP Header Field

The SOAPAction HTTP request header field can be used to indicate the intent of the SOAP HTTP request. The value is a URI identifying the intent. SOAP places no restrictions on the format or specificity of the URI or that it is resolvable. An HTTP client MUST use this header field when issuing a SOAP HTTP Request.

```
soapaction     = "SOAPAction" ":" [ <"> URI-reference <"> ]
URI-reference  = <as defined in RFC 2396 [4]>
```

The presence and content of the SOAPAction header field can be used by servers such as firewalls to appropriately filter SOAP request messages in HTTP. The header field value of empty string ("") means that the intent of the SOAP message is provided by the HTTP Request-URI. No value means that there is no indication of the intent of the message.

Examples:

Example 42

```
SOAPAction: "http://electrocommerce.org/abc#MyMessage"
SOAPAction: "myapp.sdl"
SOAPAction: ""
SOAPAction:
```

Examples of values for SOAPAction

6.2 SOAP HTTP Response

SOAP over HTTP follows the semantics of the HTTP Status codes for communicating status information in HTTP. For example, a 2xx status code indicates that the client's request including the SOAP component was successfully received, understood, and accepted etc.

If an error occurs while processing the request, the SOAP HTTP server MUST issue an HTTP 500 "Internal Server Error" response and include a SOAP message in the response containing a SOAP fault (see section 4.4) indicating the SOAP processing error.

6.3 The HTTP Extension Framework

A SOAP message MAY be used together with the experimental HTTP Extension Framework [6] in order to identify the presence and intent of a SOAP HTTP request.

Whether to use the Extension Framework or plain HTTP is a question of policy and capability of the communicating parties. Clients can force the use of the experimental HTTP Extension Framework by using a mandatory extension declaration and the "M-" HTTP method name prefix. Servers can force the use of the HTTP Extension Framework by using the 510 "Not Extended" HTTP status code. That is, using one extra round trip, either party can detect the policy of the other party and act accordingly.

The extension identifier used to identify SOAP using the Extension Framework is

`http://www.w3.org/2001/06/soap-envelope`

6.4 SOAP HTTP Examples

Example 43

```
POST /StockQuote HTTP/1.1
Content-Type: text/xml; charset="utf-8"
Content-Length: nnnn
SOAPAction: "http://electrocommerce.org/abc#MyMessage"

<env:Envelope xmlns:env="http://www.w3.org/2001/06/soap-envelope" >
  . . .
</env:Envelope>
```

SOAP HTTP Request Using POST

Example 44

```
HTTP/1.1 200 OK
Content-Type: text/xml; charset="utf-8"
```

```
Content-Length: nnnn

<env:Envelope xmlns:env="http://www.w3.org/2001/06/soap-envelope" >
 . . .
</env:Envelope>
```

SOAP HTTP Response to Example 43

Example 45

```
M-POST /StockQuote HTTP/1.1
Man: "http://www.w3.org/2001/06/soap-envelope"; ns=NNNN
Content-Type: text/xml; charset="utf-8"
Content-Length: nnnn
NNNN-SOAPAction: "http://electrocommerce.org/abc#MyMessage"

<env:Envelope xmlns:env="http://www.w3.org/2001/06/soap-envelope" >
 . . .
</env:Envelope>
```

SOAP HTTP Request using the experimental HTTP Extension Framework

Example 46

```
HTTP/1.1 200 OK
Ext:
Content-Type: text/xml; charset="utf-8"
Content-Length: nnnn

<env:Envelope xmlns:env="http://www.w3.org/2001/06/soap-envelope" >
 . . .
</env:Envelope>
```

SOAP HTTP Response to Example 45

7. Using SOAP for RPC

One of the design goals of SOAP is to encapsulate remote procedure call functionality using the extensibility and flexibility of XML. This section defines a uniform representation of RPC invocations and responses.

Although it is anticipated that this representation is likely to be used in combination with the encoding style defined in section 5, other representations are possible. The SOAP encodingStyle attribute (see section 4.3.2) can be used to indicate the encoding style of the RPC invocation and/or the response using the representation described in this section.

Using SOAP for RPC is orthogonal to the SOAP protocol binding (see section 6). In the case of using HTTP as the protocol binding, an RPC invocation maps naturally to an HTTP request and an RPC response maps to an HTTP response. However, using SOAP for RPC is not limited to the HTTP protocol binding.

To invoke an RPC, the following information is needed:

- The URI of the target SOAP node
- A procedure or method name
- An optional procedure or method signature
- The parameters to the procedure or method

- Optional header data

SOAP relies on the protocol binding to provide a mechanism for carrying the URI. For example, for HTTP the request URI indicates the resource that the invocation is being made against. Other than it be a valid URI, SOAP places no restriction on the form of an address (see [4] for more information on URIs).

7.1 RPC and SOAP Body

RPC invocations and responses are both carried in the SOAP Body element (see section 4.3) using the following representation:

- An RPC invocation is modeled as a struct.
- The invocation is viewed as a single struct containing an accessor for each [in] or [in/out] parameter. The struct is both named and typed identically to the procedure or method name.
- Each [in] or [in/out] parameter is viewed as an accessor, with a name corresponding to the name of the parameter and type corresponding to the type of the parameter. These appear in the same order as in the procedure or method signature.
- An RPC response is modeled as a struct.
- The response is viewed as a single struct containing an accessor for the return value and each [out] or [in/out] parameter. The first accessor is the return value followed by the parameters in the same order as in the procedure or method signature.
- Each parameter accessor has a name corresponding to the name of the parameter and type corresponding to the type of the parameter. The name of the return value accessor is not significant. Likewise, the name of the struct is not significant. However, a convention is to name it after the procedure or method name with the string "Response" appended.
- An invocation fault is encoded using a SOAP fault (see section 4.4). If a protocol binding adds additional rules for fault expression, those MUST also be followed.

As noted above, RPC invocation and response structs can be encoded according to the rules in section 5, or other encodings can be specified using the encodingStyle attribute (see section 4.1.1).

Applications MAY process invocations with missing parameters but also MAY return a fault.

Because a result indicates success and a fault indicates failure, it is an error for an RPC response to contain both a result and a fault.

7.2 RPC and SOAP Header

Additional information relevant to the encoding of an RPC invocation but not part of the formal procedure or method signature MAY be expressed in the RPC encoding. If so, it MUST be expressed as a header block.

An example of the use of a header block is the passing of a transaction ID along with a message. Since the transaction ID is not part of the signature and is typically held in an infrastructure component rather than application code, there is no direct way to pass the necessary information with the invocation. By adding a header block with a fixed name, the transaction manager on the receiving side can extract the transaction ID and use it without affecting the coding of remote procedure calls.

8. Security Considerations

Not described in this document are methods for integrity and privacy protection. Such issues will be addressed more fully in a future version(s) of this document.

9. References

9.1. Normative references

[2] IETF "RFC 2119: Key words for use in RFCs to Indicate Requirement Levels", S. Bradner, March 1997. Available at http://www.ietf.org/rfc/rfc2119.txt

[3] IETF "RFC 2376: XML Media Types", E. Whitehead, M. Murata, July 1998. Available at http://www.ietf.org/rfc/rfc2376.txt

[4] IETF "RFC 2396: Uniform Resource Identifiers (URI): Generic Syntax", T. Berners-Lee, R. Fielding, L. Masinter, August 1998. Available at http://www.ietf.org/rfc/rfc2396.txt

[5] IETF "RFC 2616: Hypertext Transfer Protocol -- HTTP/1.1", R. Fielding, J. Gettys, J. C. Mogul, H. Frystyk, T. Berners-Lee, January 1997. Available at http://www.ietf.org/rfc/rfc2616.txt

[6] IETF "RFC 2774: An HTTP Extension Framework", H. Nielsen, P. Leach, S. Lawrence, February 2000. Available at http://www.ietf.org/rfc/rfc2774.txt

[7] W3C Recommendation "Extensible Markup Language (XML) 1.0 (Second Edition)", Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler, 6 October 2000. Available at http://www.w3.org/TR/2000/REC-xml-20001006

[8] W3C Recommendation "Namespaces in XML", Tim Bray, Dave Hollander, Andrew Layman, 14 January 1999. Available at http://www.w3.org/TR/1999/REC-xml-names-19990114/

[9] W3C Proposed Recommendation "XML Linking Language (XLink) Version 1.0", Steve DeRose, Eve Maler, David Orchard, 20 December 2000. Available at http://www.w3.org/TR/2000/PR-xlink-20001220/

[10] W3C Recommendation "XML Schema Part 1: Structures", Henry S. Thompson, David Beech, Murray Maloney, Noah Mendelsohn, 2 May 2001. Available at http://www.w3.org/TR/2001/REC-xmlschema-1-20010502/

[11] W3C Recommendation "XML Schema Part 2: Datatypes", Paul V. Biron, Ashok Malhotra, 2 May 2001. Available at http://www.w3.org/TR/2001/REC-xmlschema-2-20010502/

9.2. Informative references

[12] Transfer Syntax NDR, in Open Group Technical Standard "DCE 1.1: Remote Procedure Call", August 1997. Available at http://www.opengroup.org/public/pubs/catalog/c706.htm

[13] IETF "RFC2045: Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies", N. Freed, N. Borenstein, November 1996. Available at http://www.ietf.org/rfc/rfc2045.txt

A. SOAP Envelope Examples

A.1 Sample Encoding of Call Requests

Example 47

```
POST /StockQuote HTTP/1.1
Host: www.stockquoteserver.com
Content-Type: text/xml; charset="utf-8"
Content-Length: nnnn
SOAPAction: "http://example.org/2001/06/quotes"

<env:Envelope
```

```
  xmlns:env="http://www.w3.org/2001/06/soap-envelope" >
   <env:Header>
       <t:Transaction
           xmlns:t="http://example.org/2001/06/tx"
           env:encodingStyle="http://www.w3.org/2001/06/soap-encoding"
           env:mustUnderstand="1" >
               5
       </t:Transaction>
   </env:Header>
   <env:Body >
       <m:GetLastTradePrice
             env:encodingStyle="http://www.w3.org/2001/06/soap-encoding"
             xmlns:m="http://example.org/2001/06/quotes" >
           <m:symbol>DEF</m:symbol>
       </m:GetLastTradePrice>
   </env:Body>
</env:Envelope>
```

Similar to Example 1 but with a Mandatory Header

Example 48

```
POST /StockQuote HTTP/1.1
Host: www.stockquoteserver.com
Content-Type: text/xml; charset="utf-8"
Content-Length: nnnn
SOAPAction: "http://example.org/2001/06/quotes"

<env:Envelope xmlns:env="http://www.w3.org/2001/06/soap-envelope" >
   <env:Body>
       <m:GetLastTradePriceDetailed
             env:encodingStyle="http://www.w3.org/2001/06/soap-encoding"
             xmlns:m="http://example.org/2001/06/quotes" >
           <Symbol>DEF</Symbol>
           <Company>DEF Corp</Company>
           <Price>34.1</Price>
       </m:GetLastTradePriceDetailed>
   </env:Body>
</env:Envelope>
```

Similar to Example 1 but with multiple request parameters

A.2 Sample Encoding of Response

Example 49

```
HTTP/1.1 200 OK
Content-Type: text/xml; charset="utf-8"
Content-Length: nnnn

<env:Envelope xmlns:env="http://www.w3.org/2001/06/soap-envelope" >
   <env:Header>
       <t:Transaction xmlns:t="http://example.org/2001/06/tx"
                      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                      xmlns:xs="http://www.w3.org/2001/XMLSchema"
                      xsi:type="xs:int"
```

```
                      env:encodingStyle="http://www.w3.org/2001/06/soap-encoding"
                      env:mustUnderstand="1" >
           5
       </t:Transaction>
   </env:Header>
   <env:Body>
       <m:GetLastTradePriceResponse
             env:encodingStyle="http://www.w3.org/2001/06/soap-encoding"
             xmlns:m="http://example.org/2001/06/quotes" >
           <Price>34.5</Price>
       </m:GetLastTradePriceResponse>
   </env:Body>
</env:Envelope>
```

Similar to Example 2 but with a Mandatory Header

Example 50

```
HTTP/1.1 200 OK
Content-Type: text/xml; charset="utf-8"
Content-Length: nnnn

<env:Envelope xmlns:env="http://www.w3.org/2001/06/soap-envelope" >
   <env:Body>
       <m:GetLastTradePriceResponse
             env:encodingStyle="http://www.w3.org/2001/06/soap-encoding"
             xmlns:m="http://example.org/2001/06/quotes" >
           <PriceAndVolume>
               <LastTradePrice>34.5</LastTradePrice>
               <DayVolume>10000</DayVolume>
           </PriceAndVolume>
       </m:GetLastTradePriceResponse>
   </env:Body>
</env:Envelope>
```

Similar to Example 2 but with a Struct

Example 51

```
HTTP/1.1 500 Internal Server Error
Content-Type: text/xml; charset="utf-8"
Content-Length: nnnn

<env:Envelope xmlns:env="http://www.w3.org/2001/06/soap-envelope">
   <env:Body>
       <env:Fault>
           <faultcode>env:MustUnderstand</faultcode>
           <faultstring>SOAP Must Understand Error</faultstring>
       </env:Fault>
   </env:Body>
</env:Envelope>
```

Similar to Example 2 but Failing to honor Mandatory Header

Example 52

```
HTTP/1.1 500 Internal Server Error
Content-Type: text/xml; charset="utf-8"
Content-Length: nnnn

<env:Envelope xmlns:env="http://www.w3.org/2001/06/soap-envelope" >
  <env:Body>
    <env:Fault>
      <faultcode>env:Server</faultcode>
      <faultstring>Server Error</faultstring>
      <detail>
        <e:myfaultdetails xmlns:e="http://example.org/2001/06/faults" >
          <message>My application didn't work</message>
          <errorcode>1001</errorcode>
        </e:myfaultdetails>
      </detail>
    </env:Fault>
 </env:Body>
</env:Envelope>
```

Similar to Example 2 but Failing to handle Body

B. Acknowledgements

This document is the work of the W3C XML Protocol Working Group.

Members of the Working Group are (at the time of writing, and by alphabetical order): Yasser al Safadi (Philips Research), Vidur Apparao (Netscape), Don Box (DevelopMentor), David Burdett (Commerce One), Charles Campbell (Informix Software), Alex Ceponkus (Bowstreet), Michael Champion (Software AG), David Clay (Oracle), Ugo Corda (Xerox), Paul Cotton (Microsoft Corporation), Ron Daniel (Interwoven), Glen Daniels (Allaire), Doug Davis (IBM), Ray Denenberg (Library of Congress), Paul Denning (MITRE Corporation), Frank DeRose (TIBCO Software, Inc.), Brian Eisenberg (Data Channel), David Ezell (Hewlett-Packard), James Falek (TIBCO Software, Inc.), David Fallside (IBM), Chris Ferris (Sun Microsystems), Daniela Florescu (Propel), Dan Frantz (BEA Systems), Dietmar Gaertner (Software AG), Scott Golubock (Epicentric), Rich Greenfield (Library of Congress), Martin Gudgin (Develop Mentor), Hugo Haas (W3C), Marc Hadley (Sun Microsystems), Mark Hale (Interwoven), Randy Hall (Intel), Gerd Hoelzing (SAP AG), Oisin Hurley (IONA Technologies), Yin-Leng Husband (Compaq), John Ibbotson (IBM), Ryuji Inoue (Matsushita Electric Industrial Co., Ltd.), Scott Isaacson (Novell, Inc.), Kazunori Iwasa (Fujitsu Software Corporation), Murali Janakiraman (Rogue Wave), Mario Jeckle (Daimler-Chrysler Research and Technology), Eric Jenkins (Engenia Software), Mark Jones (AT&T), Jay Kasi (Commerce One), Jeffrey Kay (Engenia Software), Richard Koo (Vitria Technology Inc.), Jacek Kopecky (IDOOX s.r.o.), Alan Kropp (Epicentric), Yves Lafon (W3C), Tony Lee (Vitria Technology Inc.), Michah Lerner (AT&T), Richard Martin (Active Data Exchange), Noah Mendelsohn (Lotus Development), Nilo Mitra (Ericsson Research Canada), Jean-Jacques Moreau (Canon), Masahiko Narita (Fujitsu Software Corporation), Mark Needleman (Data Research Associates), Eric Newcomer (IONA Technologies), Henrik Frystyk Nielsen (Microsoft Corporation), Mark Nottingham (Akamai Technologies), David Orchard (JamCracker), Kevin Perkins (Compaq), Jags Ramnaryan (BEA Systems), Andreas Riegg (Daimler-Chrysler Research and Technology), Hervé Ruellan (Canon), Marwan Sabbouh (MITRE Corporation), Shane Sesta (Active Data Exchange), Miroslav Simek (IDOOX s.r.o.), Simeon Simeonov (Allaire), Nick Smilonich (Unisys), Soumitro Tagore (Informix Software), James Tauber (Bowstreet), Lynne Thompson (Unisys), Patrick Thompson (Rogue Wave), Randy Waldrop (WebMethods), Ray Whitmer (Netscape), Volker Wiechers (SAP AG), Stuart Williams (Hewlett-Packard), Amr Yassin (Philips Research) and Dick Brooks (Group 8760). *Previous members were*: Eric Fedok (Active Data Exchange) Susan Yee (Active Data Exchange) Alex Milowski (Lexica), Bill Anderson (Xerox), Ed Mooney (Sun Microsystems), Mary Holstege (Calico Commerce), Rekha Nagarajan (Calico Commerce), John Evdemon (XML Solutions), Kevin Mitchell (XML Solutions), Yan Xu (DataChannel) Mike Dierken (DataChannel) Julian Kumar (Epicentric) Miles Chaston (Epicentric) Bjoern Heckel (Epicentric) Dean Moses (Epicentric) Michael Freeman (Engenia Software) Jim Hughes (Fujitsu Software Corporation) Francisco Cubera (IBM), Murray Maloney (Commerce One), Krishna Sankar (Cisco), Steve Hole (MessagingDirect Ltd.) John-Paul Sicotte (MessagingDirect Ltd.) Vilhelm Rosenqvist (NCR) Lew Shannon (NCR) Henry Lowe (OMG) Jim Trezzo (Oracle) Peter Lecuyer (Progress Software) Andrew Eisenberg (Progress Software) David Cleary (Progress Software) George Scott (Tradia Inc.) Erin Hoffman (Tradia Inc.) Conleth O'Connell (Vignette) Waqar Sadiq

(Vitria Technology Inc.) Tom Breuel (Xerox) David Webber (XMLGlobal Technologies) Matthew MacKenzie (XMLGlobal Technologies) and Mark Baker (Sun Microsystems).

This document is based on the SOAP/1.1 specification whose authors were: Don Box (Develop Mentor), David Ehnebuske (IBM), Gopal Kakivaya (Microsoft Corp.), Andrew Layman (Microsoft Corp.) Noah Mendelsohn (Lotus Development Corp.), Henrik Frystyk Nielsen (Microsoft Corp.), Satish Thatte (Microsoft Corp.) and Dave Winer (UserLand Software, Inc.).

We also wish to thank all the people who have contributed to discussions on xml-dist-app@w3.org.

C. Version Transition From SOAP/1.1 to SOAP Version 1.2

EdNote: The scope of the mechanism provided in this section is for transition between SOAP/1.1 and SOAP version 1.2. The Working Group is considering providing a more general transition mechanism that can apply to any version. Such a general mechanism may or may not be the mechanism provided here depending on whether it is deemed applicable.

The SOAP/1.1 specification says the following on versioning in section 4.1.2:

> "SOAP does not define a traditional versioning model based on major and minor version numbers. A SOAP message MUST have an Envelope element associated with the "http://schemas.xmlsoap.org/soap/envelope/" namespace. If a message is received by a SOAP application in which the SOAP Envelope element is associated with a different namespace, the application MUST treat this as a version error and discard the message. If the message is received through a request/response protocol such as HTTP, the application MUST respond with a SOAP VersionMismatch faultcode message (see section 4.4) using the SOAP "http://schemas.xmlsoap.org/soap/envelope/" namespace."

That is, rather than a versioning model based on shortnames (typically version numbers), SOAP uses a declarative extension model which allows a sender to include the desired features within the SOAP envelope construct. SOAP says nothing about the granularity of extensions nor how extensions may or may not affect the basic SOAP processing model. It is entirely up to extension designers be it either in a central or a decentralized manner to determine which features become SOAP extensions.

The SOAP extensibility model is based on the following four basic assumptions:

1. SOAP versioning is directed only at the SOAP envelope. It explicitly does not address versioning of blocks, encodings, protocol bindings, or otherwise.
2. A SOAP node must determine whether it supports the version of a SOAP message on a per message basis. In the following, "support" means understanding the semantics of the envelope version identified by the QName of the Envelope element:
    - A SOAP node receiving an envelope that it doesn't support must not attempt to process the message according to any other processing rules regardless of other up- or downstream SOAP nodes.
    - A SOAP node may provide support for multiple envelope versions. However, when processing a message a SOAP node must use the semantics defined by the version of that message.
3. It is essential that the envelope remains stable over time and that new features are added using the SOAP extensibility mechanism. Changing the envelope inherently affects interoperability, adds complexity, and requires central control of extensions -- all of which directly conflicts with the SOAP requirements.
4. No versioning model or extensibility model can prevent buggy implementations. Even though significant work has been going into clarifying the SOAP processing model, there is no guarantee that a SOAP 1.2 implementation will behave correctly. Only extensive testing within the SOAP community and design simplicity at the core can help prevent/catch bugs.

The rules for dealing with the possible SOAP/1.1 and SOAP Version 1.2 interactions are as follows:

1. Because of the SOAP/1.1 rules, a compliant SOAP/1.1 node receiving a SOAP Version 1.2 message will generate a VersionMismatch SOAP fault using an envelope qualified by the "http://schemas.xmlsoap.org/soap/envelope/" namespace identifier.

2. A SOAP Version 1.2 node receiving a SOAP/1.1 message may either process the message as SOAP/1.1 or generate a SOAP VersionMismatch fault using the "http://schemas.xmlsoap.org/soap/envelope/" namespace identifier. As part of the SOAP VersionMismatch fault, a SOAP Version 1.2 node should include the list of envelope versions that it supports using the SOAP upgrade extension identified by the "http://www.w3.org/2001/06/soap-upgrade" identifier.

The upgrade extension contains an ordered list of namespace identifiers of SOAP envelopes that the SOAP node supports in the order most to least preferred. Following is an example of a VersionMismatch fault generated by a SOAP Version 1.2 node including the SOAP upgrade extension:

Example 53

```
<env:Envelope xmlns:env="http://schemas.xmlsoap.org/soap/envelope/">
  <env:Header>
    <V:Upgrade xmlns:V="http://www.w3.org/2001/06/soap-upgrade">
      <envelope qname="ns1:Envelope"
xmlns:ns1="http://www.w3.org/2001/06/soap-envelope"/>
    </V:Upgrade>
  </env:Header>
  <env:Body>
    <env:Fault>
      <faultcode>env:VersionMismatch</faultcode>
      <faultstring>Version Mismatch</faultstring>
    </env:Fault>
  </env:Body>
</env:Envelope>
```

VersionMismatch fault generated by a SOAP Version 1.2 node, and including a SOAP upgrade extension

Note that existing SOAP/1.1 nodes are not likely to indicate which envelope versions they support. If nothing is indicated then this means that SOAP/1.1 is the only supported envelope.

D. Change Log

D.1 SOAP Specification Changes

Date	Author	Description
20010629	MJG	Amended description of routing and intermediaries in Section 2.1
20010629	JJM	Changed "latest version" URI to end with soap12
20010629	JJM	Remove "previous version" URI
20010629	JJM	Removed "Editor copy" in <title>
20010629	JJM	Removed "Editor copy" in the title.
20010629	JJM	Added "Previous version" to either point to SOAP/1.1, or explicitly mention there was no prior draft.
20010629	JJM	Pre-filed publication URIs.
20010629	JJM	Incorporated David's suggested changes for the examples in section 4.1.1 to 4.4.2
20010629	JJM	Fixed some remaining typos.
20010629	MJH	Fixed a couple of typos.
20010628	MJG	Made various formatting, spelling and grammatical fixes.
20010628	MJG	Moved soap:encodingStyle from soap:Envelope to children of soap:Header/soap:Body in examples 1, 2, 47, 48, 49 and 50
20010628	MJG	Changed text in Section 2.1 from 'it is both a SOAP sender or a SOAP receiver' to 'it is both a SOAP sender and a SOAP receiver'
20010628	MJG	Fixed caption on Example 24

20010628	MJH	Fixed a couple of capitalisation errors where the letter A appeared as a capital in the middle of a sentence.
20010628	MJH	Updated figure 1, removed ednote to do so.
20010622	HFN	Removed the introductory text in terminology section 1.4.3 as it talks about model stuff that is covered in section 2. It was left over from original glossary which also explained the SOAP model.
20010622	HFN	Moved the definition of block to encapsulation section in terminology
20010622	HFN	Removed introductory section in 1.4.1 as this overlaps with the model description in section 2 and doesn't belong in a terminology section
20010622	HFN	Removed reference to "Web Characterization Terminology & Definitions Sheet" in terminology section as this is not an active WD
20010622	HFN	Added revised glossary
20010622	HFN	Added example 0 to section 1.3 and slightly modified text for example 1 and 2 to make it clear that HTTP is used as a protocol binding
20010622	MJG	Added http://example.com/... to list of application/context specific URIs in section 1.2
20010622	MJG	Updated examples in section 4.1.1 to be encodingStyle attributes rather than just the values of attributes
20010622	MJG	Added table.norm, td.normitem and td.normtext styles to stylesheet. Used said styles for table of fault code values in section 4.4.1
20010622	MJG	In Appendix C, changed upgrade element to Upgrade and env to envelope. Made envelope unqualified. Updated schema document to match.
20010622	MJG	Moved MisunderstoodHeader from envelope schema into seperate faults schema. Removed entry in envelope schema change table in Appendix D.2 that refered to additon of said element. Modified example in section 4.4.2 to match. Added reference to schema document to section 4.4.2
20010622	MJH	Added binding as a component of SOAP in introduction. Fixed a couple of typos and updated a couple of example captions.
20010622	MJG	Made BNF in section 6.1.1 into a table.
20010622	MJG	Made BNFs in section 5.1 clause 8 into tables. Added associated 'bnf' style for table and td elements to stylesheet
20010622	MJG	Amended text regarding namespace prefix mappings in section 1.2
20010622	MJG	Added link to schema for the http://www.w3.org/2001/06/soap-upgrade namespace to Appendix C. Updated associated ednote.
20010622	MJG	Added reference numbers for XML Schema Recommendation to text prior to schema change tables in Appendix D.2 and linked said numbers to local references in this document
20010622	MJG	Reordered entries in schema change classification table in Appendix D.2
20010622	MJG	Changed type of mustUnderstand and root attributes to standard boolean and updated schema change tables in Appendix D.2 accordingly
20010622	JJM	Manually numbered all the examples (53 in total!)
20010622	JJM	Added caption text to all the examples
20010622	JJM	Replaced remaining occurrences of SOAP/1.2 with SOAP Version 1.2 (including <title>)
20010621	HFN	Added ednote to section 4.2.2 and 4.2.3 that we know they have to be incorporated with section 2
20010621	HFN	Added version transition appendix C
20010621	HFN	Applied new styles to examples
20010621	HFN	Changed term "transport" to "underlying protocol
20010621	HFN	Changed example URNs to URLs of the style http://example.org/...
20010621	MJH	Updated the Acknowledgements section.
20010621	JJM	Added new style sheet definitions (from XML Schema) for examples, and used them for example 1 and 2.
20010621	JJM	Incorporated David Fallside's comments on section Status and Intro sections.

20010620	HFN	Changed the status section
20010620	HFN	Changed title to SOAP Version 1.2 and used that first time in abstract and in body
20010620	HFN	Removed question from section 2.4 as this is an issue and is to be listed in the issues list
20010620	HFN	Moved change log to appendix
20010615	JJM	Renamed default actor to anonymous actor for now (to be consistent)
20010615	JJM	Fixed typos in section 2
20010614	JJM	Updated section 2 to adopt the terminology used elsewhere in the spec.
20010613	MJH	Updated mustUnderstand fault text with additions from Martin Gudgin.
20010613	MJH	Added schema changes appendix from Martin Gudgin.
20010613	MJH	Added mustUnderstand fault text from Glen Daniels.
20010612	MJH	Fixed document <title>.
20010612	MJH	Moved terminology subsection from message exchange model section to introduction section.
20010612	MJH	Fixed capitalisation errors by replacing "... A SOAP ..." with "... a SOAP ..." where appropriate.
20010612	MJH	Removed trailing "/" from encoding namespace URI.
20010612	MJH	Fixed links under namespace URIs to point to W3C space instead of schemas.xmlsoap.org.
20010612	MJH	Removed some odd additional links with text of "/" pointing to the encoding schema following the text of the encoding namespace URI in several places.
20010611	MJH	Incorporated new text for section 2.
20010611	JJM	Changed remaining namespaces, in particular next.
20010609	JJM	Changed the spec name from XMLP/SOAP to SOAP.
20010609	JJM	Changed the version number from 1.1 to 1.2.
20010609	JJM	Changed the namespaces from http://schemas.xmlsoap.org/soap/ to http://www.w3.org/2001/06/soap-.
20010609	JJM	Replaced the remaining XS and XE prefixes to env and enc, respectively.
20010601	MJH	Updated the examples in section 1, 6 and appendix A with text suggested by Martin Gudgin to comply with XML Schema Recommendation.
20010601	JJM	Updated the examples in section 4 and 5 with text suggested by Martin Gudgin, to comply with XML Schema Recommendation.
20010531	HFN	Removed appendices C and D and added links to live issues list and separate schema files.
20010531	MJH	Added this change log and updated schemas in appendix C to comply with XML Schema Recommendation.

D.2 XML Schema Changes

The envelope and encoding schemas have been updated to be compliant with the XML Schema Recomendation[10,11]. The table below shows the categories of change.

Class	Meaning
Addition	New constructs have been added to the schema
Clarification	The meaning of the schema has been changed to more accurately match the specification
Deletion	Constructs have been removed from the schema
Name	The schema has been changed due to a datatype name change in the XML Schema specification
Namespace	A namespace name has been changed
Semantic	The meaning of the schema has been changed
Style	Style changes have been made to the schema
Syntax	The syntax of the schema has been updated due to changes in the XML Schema specification

The table below lists the changes to the envelope schema.

Class	Description
Namespace	Updated to use the http://www.w3.org/2001/XMLSchema namespace

Namespace	Value of targetNamespace attribute changed to http://www.w3.org/2001/06/soap-envelope
Clarification	Changed element and attribute wildcards in Envelope complex type to namespace="##other"
Clarification	Changed element and attribute wildcards in Header complex type to namespace="##other"
Clarification	Added explicit namespace="##any" to element and attribute wildcards in Body complex type
Clarification	Added explicit namespace="##any" to element and attribute wildcards in detail complex type
Clarification	Added an element wildcard with namespace="##other" to the Fault complex type
Name	Changed item type of encodingStyle from uri-reference to anyURI
Name	Changed type of actor attribute from uri-reference to anyURI
Name	Changed type of faultactor attribute from uri-reference to anyURI
Semantic	Added processContents="lax" to all element and attribute wildcards
Semantic	Changed type of the mustUnderstand attribute from restriction of boolean that only allowed 0 or 1 as lexical values to the standard boolean in the http://www.w3.org/2001/XMLSchema namespace. The lexical forms 0, 1, false, true are now allowed.
Style	Where possible comments have been changed into annotations
Syntax	Changed all occurences of maxOccurs="*" to maxOccurs="unbounded"
Syntax	Added <xs:sequence> to all complex type definitions derived implicitly from the ur-type
Syntax	Added <xs:sequence> to all named model group definitions

The table below lists the changes to the encoding schema.

Class	Description
Namespace	Updated to use the http://www.w3.org/2001/XMLSchema namespace
Namespace	Value of targetNamespace attribute changed to http://www.w3.org/2001/06/soap-encoding
Semantic	Changed type of the root attribute from restriction of boolean that only allowed 0 or 1 as lexical values to the standard boolean in the http://www.w3.org/2001/XMLSchema namespace. The lexical forms 0, 1, false, true are now allowed.
Addition	Added processContents="lax" to all element and attribute wildcards
Syntax	Changed base64 simple type to be a vacuous restriction of the base64Binary type in the http://www.w3.org/2001/XMLSchema namespace
Syntax	Updated all complex type definitions with simple base types to new syntax
Syntax	Added <xs:sequence> to all complex type definitions derived implicitly from the ur-type
Syntax	Added <xs:sequence> to all named model group definitions
Deletion	Removed the timeDuration datatype
Addition	Added duration datatype derived by extension from the duration datatype in the http://www.w3.org/2001/XMLSchema namespace.
Deletion	Removed the timeInstant datatype
Addition	Added dateTime datatype derived by extension from the dateTime datatype in the http://www.w3.org/2001/XMLSchema namespace.
Addition	Added gYearMonth datatype derived by extension from the gYearMonth datatype in the http://www.w3.org/2001/XMLSchema namespace.
Addition	Added gYear datatype derived by extension from the gYear datatype in the http://www.w3.org/2001/XMLSchema namespace.
Addition	Added gMonthDay datatype derived by extension from the gMonthDay datatype in the http://www.w3.org/2001/XMLSchema namespace.
Addition	Added gDay datatype derived by extension from the gDay datatype in the http://www.w3.org/2001/XMLSchema namespace.
Addition	Added gDay datatype derived by extension from the gDay datatype in the http://www.w3.org/2001/XMLSchema namespace.
Deletion	Removed the binary datatype
Addition	Added hexBinary datatype derived by extension from the hexBinary datatype in the http://www.w3.org/2001/XMLSchema namespace.

Addition	Added base64Binary datatype derived by extension from the base64Binary datatype in the http://www.w3.org/2001/XMLSchema namespace.
Deletion	Removed the uriReference datatype
Addition	Added anyURI datatype derived by extension from the anyURI datatype in the http://www.w3.org/2001/XMLSchema namespace.
Addition	Added normalizedString datatype derived by extension from the normalizedString datatype in the http://www.w3.org/2001/XMLSchema namespace.
Addition	Added token datatype derived by extension from the token datatype in the http://www.w3.org/2001/XMLSchema namespace.
Clarification	Added explicit namespace="##any" to all element and attribute wildcards which did not previously have an explicit namespace attribute
Style	Where possible comments have been changed into annotations

In addition several changes occured in the names of datatypes in the XML Schema specification and some datatypes were removed. The following table lists those changes.

Datatype	Class	Description
timeDuration	Renamed	New name is duration
timeInstant	Renamed	New name is dateTime
recurringDuration	Removed	The recurringDuration datatype no longer exists.
recurringInstant	Removed	The recurringInstant datatype no longer exists.
binary	Removed	The binary datatype has been replaced by the hexBinary and base64Binary datatypes.
month	Renamed	New name is gYearMonth
timePeriod	Removed	The timePeriod datatype no longer exists
year	Renamed	New name is gYear
century	Removed	The century datatype no longer exists
recurringDate	Renamed	New name is gMonthDay
recurringDay	Renamed	New name is gDay

Last Modified: $Date: 2001/07/09 13:39:15 $ UTC