OMG Document No: ptc/2006-06-03
Date: June 2, 2006

Convenience Document for CORBA WSDL Interworking Spec RTF

CORBA to WSDL/SOAP Interworking 1

1.1 Overview

This specification maps a valid set of OMG IDL constructs to WSDL. The mapping is done based on SOAP 1.1 and WSDL 1.1 versions.

The following have been assumed:

- Standard preprocessing of IDL is performed by the mapping tool.
- While the mapping of WSDL portTypes and SOAP bindings may be standardized, the generation of SOAP endpoints will require information above that found in the input IDL, and so is beyond the scope of this mapping specification.
- All data types referred in WSDL are XML Schema data types.

1.1.1 Conformance Requirements

The mandatory conformance point for this standard requires that the specification is implemented in its entirety except for the processing of valuetypes as noted in Section 1.2.7.10, “ValueType,” on page 1-16. It should be noted that to be conformant with the mandatory conformance point the implementation must generate the WSDL for valuetypes.

Implementations may claim two optional conformance points comprising the processing of valuetypes as noted in Section 1.2.7.10, “ValueType,” on page 1-16.

- Simple Value type Support - An implementation may claim run time support for value type mapping, without supporting value type sharing.
- Value type Sharing Support - An implementation that claims Simple value type support may also claim to support value type sharing.
- There are also run-time conformance points corresponding to soap encoding support, as follows:

- An implementation may claim support of RPC style soap bindings with soap encoding, or
- an implementation may claim support of WS-I conformant soap bindings with use literal, or
- an implementation may claim conformance to both forms of soap binding above.

1.2 IDL - WSDL Mapping

1.2.1 Overall Goals

The overall goal of this specification is to provide a natural mapping from IDL to WSDL that is also suitable for a reverse mapping, from the mapped subset of WSDL back to IDL.

It will also be important to closely track the JAX-RPC specification, and it is an agreed goal of the submitters of this specification to follow its approach where possible, to allow the interchange of JAX-RPC and CORBA-derived types.

The upcoming WS-I specifications have mandated use of soap bindings with style=rpc and use=literal. The first two submissions supported soap bindings with style=rpc and use=encoded. To allow compatibility with WS-I conformant systems, this revision of the mapping requires generation to both rpc/encoded and rpc/literal soap bindings. An implementation of this specification may claim run time conformance to either rpc/encoded, or rpc/literal, or both.

1.2.2 Conventions

1.2.2.1 XML Namespaces

Table 1-1 lists the XML namespaces used throughout this document, together with the namespace prefixes used for brevity.

Table 1-1 XML Namespaces

xsd	http://www.w3.org/2001/XMLSchema/
w	http://schemas.xmlsoap.org/wsdl/
soapenc	http://schemas.xmlsoap.org/soap/encoding/
CORBA	http://www.omg.org/IDL-WSDL/1.0/ This namespace is used for standard CORBA entities (e.g., SystemException)
tns	http://www.omg.org/IDL-Mapped/ This is the namespace used for generated WSDL code.

1.2.3 Identifying the Source IDL

In some cases it will be necessary to identify the source IDL used in generating a WSDL mapping. The generated WSDL will provide a hint, in the form of an XML schema annotation giving both a reference to the source IDL and the version of the mapping used. Hints may be provided that refer to the source IDL file, or to the repository ID for a given generated construct (including any prefixes defined by #pragma prefix directives).

The following schema, in the CORBA namespace, defines the basic form of this annotation for identifying a source IDL file:

```
<xsd:element name="SourceIDL" minoccurs=0>
    <xsd:annotation>
        <xsd:documentation>IDL/WSDL Mapping Info</xsd:documentation>
    </xsd:annotation>
    <xsd:complexType>
        <xsd:sequence>
            <xsd:element name="source"
                type="xsd:string"
                minOccurs="1"
                maxOccurs="1"/>
            <xsd:element name="version"
                type="xsd:string"
                minOccurs="1"
                maxOccurs="1"/>
        </xsd:sequence>
    </xsd:complexType>
</xsd:element>
```

The generated WSDL will then use this schema to indicate the mapping information as shown in the following example:

```
<w:documentation>
 <CORBA:SourceIDL>
  <CORBA:source>IDL Source</CORBA:source>
  <CORBA:version>1.0</CORBA:version>
 </CORBA:SourceIDL>
</w:documentation>
```

The following schema, in the CORBA namespace, defines the basic form of this annotation for providing the repository ID for a mapped construct:

```
<xsd:element name="SourceRepositoryID" minoccurs=0>
    <xsd:annotation>
        <xsd:documentation>IDL Mapped Repository ID </xsd:documentation>
    </xsd:annotation>
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="repositoryID" type="xsd:string"minOccurs="1" maxOccurs="1"/>
       <xsd:element name="version" type="xsd:string" minOccurs="1" maxOccurs="1"/>
      </xsd:sequence>
    </xsd:complexType>
</xsd:element>
```

The generated WSDL will then use this schema to indicate the mapping information as shown in the following example:

```
<w:documentation>
 <CORBA:SourceRepositoryID>
   <CORBA:repositoryID>repositoryIDtext </CORBA:repositoryID>
  <CORBA:version>1.0</CORBA:version>
 </CORBA:SourceRepositoryID>
</w:documentation>
```

1.2.4 Modules

For every module name that does not collide with another module name, across the set of IDL artifacts that are being mapped to a WSDL document, then the scoped name of the module starting from the root module name using the "." character as separator between names shall be the effective Id used in the corresponding WSDL.

If there is a module name collision, apply the following rules to the module name that causes the collision:

a. If there are no pragma or typeid/typeprefix directives that apply to a module name, then the scoped name of the module starting from the root module name using the "." character as separator between names shall be the effective Id used in the corresponding WSDL.

b. If there is a pragma prefix or a typeprefix directive that applies to the module name, then the prefix specified in the directive/pragma shall be of the form:

   <prefix string> "_" <effective Id from (1)>

   If no such directives apply to the module name in question, then the effective Id shall be the same as in (1).

c. If there is a pragma version that applies to the scopename, then the version string shall be appended to the effective Id obtained in (2), that is the effective Id will be of the form:

   <effective Id from (2)> "_" <version string>

d. If a pragma id or a typeid directive applies to the module name in question, then:
   - If the first four characters of the string specified in the directive/pragma is "IDL:", then the string with the "IDL:" prefix removed and with the "/" characters in the remaining string substituted by the "." character shall be the effective Id.
   - If the Repository Id type prefix in the directive/pragma is something other than "IDL:", then the entire id string shall be used as the effective Id.
   - If any characters that are not valid for XML element names are encountered in the mapped module name, they shall be replaced with "_."

It would seem desirable to map the module construct to an XML namespace; however, there are several problems with this approach. Having a separate namespace for each imported module results in a large number of files for the WSDL processor to deal with when constructing a gateway. The current mapping allows the user to refer to the scoped name without having to import each namespace used by the schema.

1.2.5 Object References

Object references are mapped to a sequence of URIs. Each URI in the sequence corresponds to one mechanism, which can be used to access the implementation of the CORBA object referenced. This sequence can include one or more HTTP URL for soap endpoint(s) corresponding to the object reference, and/or one or more of the following types (which are defined in section 2.5 of the CORBA Naming Service 1.1 specification, and section 13.6.10 of the CORBA 3.0 specification:

- corbaloc:
- corbaname:
- ior:

The object reference sequence is defined in WSDL as follows (in the CORBA namespace):

```
<xsd:complexType name="ObjectReference">
    <xsd:sequence>
        <xsd:element
                name="url" type="xsd:anyURI"
                minOccurs="1" maxOccurs="unbounded"/>
    </xsd:sequence>
</xsd:complexType>
```

1.2.6 Primitive Types

Each IDL primitive type maps onto a corresponding type from the XML schema specification. (In the following table, these XML Schema types are shown in the conventional "xsd" namespace, to distinguish them from the IDL types).

Table 1-2 Mapping of IDL primitive types

IDL	**WSDL**
boolean	xsd:boolean
char	<xsd:simpleType name= "char"> <xsd:restricition base="xsd:string"> <xsd:length value="1" fixed="true"/> </xsd:restricition> </xsd:simpleType>
wchar	<xsd:simpleType name= "wchar"> <xsd:restricition base="xsd:string"/> </xsd:simpleType>
double	xsd:double
float	xsd:float
octet	xsd:unsignedByte
long	xsd:int
long long	xsd:long
short	xsd:short
string	xsd:string Bounded strings are supported by use of an XML restriction on the "length" attribute. **// IDL** **typedef string<10> boundedString;** <!-- WSDL --> <xsd:simpleType name="boundedString"> <xsd:restriction base="xsd:string"> <xsd:maxLength value="10" fixed="true"/> </xsd:restriction> </xsd:simpleType>
wstring	xsd:string

Table 1-2 Mapping of IDL primitive types

IDL	WSDL
TypeCode	Primitive TypeCode are not mapped. All type information is represented by the appropriate XML Schema declaration. The mapping for CORBA::TypeCode objects is discussed in Section 1.2.7.7, “TypeCode,” on page 1-15.
unsigned short	xsd:unsignedShort
unsigned long	xsd:unsignedInt
unsigned long long	xsd:unsignedLong

1.2.6.1 Constants

IDL constants are mapped by substituting their value directly in the generated WSDL. Consider the following example:

```
// IDL
const short S = 5;
typedef sequence<string,S> strSeq;
```

which maps to (only the WS-I BP compliant version is shown for simplicity):

```
<!--WSDL -->
<xsd:complexType name="strSeq">
        <xsd:sequence>
            <xsd:element name="item"
                minOccurs="0" maxOccurs="5"
                type="xsd:string"/>
        </xsd:sequence>
</xsd:complexType>
```

1.2.7 Constructed Types

1.2.7.1 Enum

CORBA has enumerators that are not explicitly tagged with values. The constraint is that any language mapping that permits two enumerators to be compared or defines successor or predecessor functions on enumerators must conform to the ordering of the enumerators as specified in the OMG IDL. Enum in IDL is mapped to ‘enumeration’ of XML Schema with restriction placed on ‘string.’

```
// OMG IDL
enum myEnum {A, B, C};
```

This maps to:

```
<!-- WSDL -->
<xsd:simpleType name=”myEnum”>
    <xsd:restriction base=”xsd:string”>
        <xsd:enumeration value=”A”/>
        <xsd:enumeration value=”B”/>
        <xsd:enumeration value=”C”/>
    </xsd:restriction>
</xsd:simpleType>
```

1.2.7.2 Structures

IDL structures are mapped to XML Schema complexType definitions. The elements of the IDL structure are encapsulated in an XML Schema sequence within the complexType.

Consider the following IDL structure:

```
module Example {
    struct myStruct {
        char c;
        string str;
        octet o;
        short s;
        unsigned long long ull;
        float f;
        double d;
    };
};
```

This structure maps onto the following XML schema:

```
<xsd:complexType name="Example.myStruct">
    <xsd:sequence>
        <xsd:element name="c" type="xsd:string"
            maxOccurs="1" minOccurs="1"/>
        <xsd:element name="str" type="xsd:string"
            nillable="true"
            maxOccurs="1" minOccurs="1"/>
        <xsd:element name="o" type="xsd:byte"
            maxOccurs="1" minOccurs="1"/>
        <xsd:element name="s" type="xsd:short"
            maxOccurs="1" minOccurs="1"/>
        <xsd:element name="ull" type="xsd:unsignedLong"
            maxOccurs="1" minOccurs="1"/>
        <xsd:element name="f" type="xsd:float"
            maxOccurs="1" minOccurs="1"/>
        <xsd:element name="d" type="xsd:double"
            maxOccurs="1" minOccurs="1"/>
    </xsd:sequence>
</xsd:complexType>
```

1.2.7.3 Typedefs

IDL type definitions are mapped to XML schema type restrictions. Consider the following IDL fragment:

```
module Example {
    typedef long Number;
    typedef Number OtherNumber;
};
```

The corresponding XML schema definitions would be:

```
<xsd:simpleType name="Example.Number">
        <xsd:restriction base="xsd:int" />
</xsd:simpleType>

<xsd:simpleType name="Example.OtherNumber">
        <xsd:restriction base="Example.Number" />
</xsd:simpleType>
```

Another example, involving a complex type:

```
// IDL
struct S { long dummy; };
typedef S S_t;
```

The corresponding XML schema definitions would be:

```
<xsd:complexType name="S">
    <xsd:sequence>
        <xsd:element name="dummy" type="xsd:int" maxOccurs="1" minOccurs="1" />
    </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="S_t">
    <xsd:complexContent>
    <xsd:restriction base="tns:S">
    <xsd:sequence>
        xsd:element name="dummy" type="xsd:int" maxOccurs="1" minOccurs="1" />
    </xsd:sequence>
    </xsd:restriction>
    </xsd:complexContent>
</xsd:complexType>
```

1.2.7.4 Unions

The XML Schema specification provides a “choice” element, which allows us to represent IDL unions in a very straightforward manner. Unions are mapped onto a “complexType,” containing a sequence of elements: the discriminator, and the union cases, each mapped to a “choice” element.

Any valid value for the discriminant, other than those used in the union definition, may be inserted in the XML generated, at run time, for the default case.

Consider the following example:

```
module Example {
    union myUnion switch (long) {
        case 0: long l;
        case 1: string str;
        case 2:
        case 3: float f;
        default: octet o;
    };
};
```

This union maps onto the following XML schema definition:

```
<xsd:complexType name="Example.myUnion">
   <xsd:sequence>
      <xsd:element name="discriminator" type="xsd:int" />
      <xsd:choice>

         <!-- case 0 -->
         <xsd:element name="l" type="xsd:int"
            minOccurs="0" maxOccurs="1" />

         <!-- case 1 -->
         <xsd:element name="str" type="xsd:string"
            nillable="true" minOccurs="0" maxOccurs="1" />

         <!-- case 2, 3 -->
         <xsd:element name="f" type="xsd:float"
            minOccurs="0" maxOccurs="1" />

         <!-- default case -->
         <xsd:element name="o" type="xsd:byte"
            minOccurs="0" maxOccurs="1" />
      </xsd:choice>
   </xsd:sequence>
</xsd:complexType>
```

1.2.7.5 Sequences

This specification supports two mappings for IDL Sequences and Arrays:

- SOAP Encoding - The SOAP specification provides an encoding for arrays. IDL sequences are mapped onto these SOAP arrays. For each IDL sequence a corresponding complex type is created as a restriction of soapenc:Array. The name of the IDL type is prefixed with “_SE_”, before applying any required IDL moduleprefixes.
- XML Schema for WS-I conformant RPC/Literal Soap Binding – an IDL sequence is mapped onto a sequence complex type (the SOAP encoding soapenc:Array type MUST NOT be used with WS-I conformant soap bindings).

If the IDL sequence is unbounded, then so is the corresponding schema definition. Bounded sequences have their bounds represented accordingly.

Consider the following IDL:

```
module Example {
    typedef sequence<long> longSeq;
    typedef sequence<string,10> strSeq;

        struct myStruct {
        // ...
    };

    typedef sequence<myStruct> structSeq;
};
```

This is mapped onto the following WSDL (the first of each pair of mappings, with name prefix "_SE_" is for Soap Encoding (rpc/encoded soap bindings), the second is for WS-I conformant systems (rpc/literal soap bindings):

```
<xsd:complexType name="Example._SE_longSeq">
   <xsd:complexContent>
      <xsd:restriction base="soapenc:Array">
         <xsd:sequence>
            <xsd:element name="item"
               minOccurs="0" maxOccurs="unbounded"
               type="xsd:int"/>
         </xsd:sequence>
         <xsd:attribute ref="soapenc:arrayType"
               w:arrayType="xsd:int[]"/>
      </xsd:restriction>
   </xsd:complexContent>
</xsd.complexType>

<xsd:complexType name="Example.longSeq">
         <xsd:sequence>
            <xsd:element name="item"
               minOccurs="0" maxOccurs="unbounded"
               type="xsd:int"/>
         </xsd:sequence>
</xsd.complexType>

<xsd:complexType name="Example._SE_strSeq">
   <xsd:complexContent>
      <xsd:restriction base="soapenc:Array">
         <xsd:sequence>
            <xsd:element name="item"
               minOccurs="0" maxOccurs="10"
               type="xsd:string"/>
         </xsd:sequence>
         <xsd:attribute ref="soapenc:arrayType"
               w:arrayType="xsd:string[]"/>
      </xsd:restriction>
   </xsd:complexContent>
</xsd.complexType>

<xsd:complexType name="Example.strSeq">
         <xsd:sequence>
            <xsd:element name="item"
               minOccurs="0" maxOccurs="10"
               type="xsd:string"/>
```

```
            </xsd:sequence>
</xsd.complexType>

<xsd.complexType name="Example._SE_structSeq">
    <xsd:complexContent>
        <xsd:restriction base="soapenc:Array">
            <xsd:sequence>
                <xsd:element name="item"
                    minOccurs="0" maxOccurs="unbounded"
                    type="Example.myStruct"/>
            </xsd:sequence>
            <xsd:attribute ref="soapenc:arrayType"
                    w:arrayType="Example.myStruct[]"/>
        </xsd:restriction>
    </xsd:complexContent>
</xsd.complexType>

<xsd.complexType name="Example.structSeq">
            <xsd:sequence>
                <xsd:element name="item"
                    minOccurs="0" maxOccurs="unbounded"
                    type="Example.myStruct"/>
            </xsd:sequence>
</xsd.complexType>
```

1.2.7.6 Arrays

IDL arrays are mapped just like bounded sequences. There are two forms of arrays, those declared implicitly as part of a field or parameter, and those declared using a typedef declaration. Implicit declarations follow the same pattern as sequences. However, implicit declarations in IDL will require an explicit declaration in XML. This follows the same pattern as the explicit IDL case, but the name of the XML type is constructed by concatenating the name of the enclosing type and the name of the field, using an underscore '_' as a separator.

The explicit IDL declaration

```
module Example {
    typedef long arrayLong[10];
};
```

is mapped to the following: (one for soap encoding, the other for WS-I conformance):

```
<xsd:complexType name="Example._SE_arrayLong" >
    <xsd:complexContent >
        <xsd:restriction base="soapenc:Array" >
            <xsd:sequence >
                <xsd:element
                    name="item" type="xsd:int"
                    minOccurs="10" maxOccurs="10"/>
            </xsd:sequence>
            <xsd:attribute
                ref="soapenc:arrayType"
                w:arrayType="xsd:int[]"/>
        </xsd:restriction>
```

```
    </xsd:complexContent>
</xsd:complexType>

<xsd:complexType name="Example.arrayLong" >
        <xsd:sequence >
            <xsd:element
                name="item" type="xsd:int"
                minOccurs="10" maxOccurs="10"/>
        </xsd:sequence>
</xsd:complexType>
```

The implicit IDL declaration:

```
struct T {
    long field[10];
};
```

is mapped to the following in XML (one for Soap encoding, the other for WS-I conformance):

```
<xsd:complexType name="_SE_T.field_ArrayOfint">
    <xsd:complexContent >
        <xsd:restriction base="soapenc:Array" >
            <xsd:sequence >
                <xsd:element
                    name="item" type="xsd:int"
                    minOccurs="10" maxOccurs="10"/>
            </xsd:sequence>
            <xsd:attribute
                ref="soapenc:arrayType"
                w:arrayType="xsd:int[]"/>
        </xsd:restriction>
    </xsd:complexContent>
</xsd:complexType>

<xsd:complexType name="_SE_T">
    <xsd:sequence>
        <xsd:element
            name="field" type="_SE_T.field_ArrayOfint"
            nillable="true"
            maxOccurs="1" minOccurs="1"/>
    </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="T.field_ArrayOfint">
            <xsd:sequence >
                <xsd:element
                    name="item" type="xsd:int"
                    minOccurs="10" maxOccurs="10"/>
            </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="T">
    <xsd:sequence>
        <xsd:element
            name="field" type="T.field_ArrayOfint"
            nillable="true"
```

```
            maxOccurs="1" minOccurs="1"/>
    </xsd:sequence>
</xsd:complexType>
```

Multi-dimensional arrays in IDL are mapped by generating intermediate types for each of the sub-arrays. For example, a two-dimensional array of strings would map onto a uni-dimensional “ArrayOfString,” and then an array of that type.

In a case where the generated identifier for an intermediate array collides with an already mapped IDL intermediate array type definition, append the suffix “_n” to uniquely define the identifier of the nth arising name collision with a previously generated intermediate array identifier.

Note – Note: A type name collision only occurs when an intermediate array generated for one multidimensional array (e.g., long matrix[5][3] -> ArrayOfInt[5]) has the same name as that generated for another mutidimensional array, and has a different array range value (e.g., long anotherMatrix[6][4] -> ArrayOfInt[6]).

This mapping for multidimensional arrays results in a different order of transfer (on the wire) for the array elements, than the order in which GIOP marshalling transfers the mapped multidimensional IDL array.

For example, consider the following IDL:

typedef long matrix[5][3];

maps onto the following XML Schema code (one set for Soap encoding, the other for WS-I conformance):

```
<xsd:complexType name="_SE_ArrayOfint" >
    <xsd:complexContent >
        <xsd:restriction base="soapenc:Array" >
            <xsd:sequence >
                <xsd:element
                    name="item" type="xsd:int"
                    minOccurs="5" maxOccurs="5"/>
            </xsd:sequence>
            <xsd:attribute
                ref="soapenc:arrayType"
                w:arrayType="xsd:int[]"/>
        </xsd:restriction>
    </xsd:complexContent>
</xsd:complexType>

<xsd:complexType name="_SE_matrix" >
    <xsd:complexContent >
        <xsd:restriction base="soapenc:Array" >
            <xsd:sequence >
                <xsd:element
                    name="item1" type="_SE_ArrayOfint"
                    minOccurs="3" maxOccurs="3"/>
            </xsd:sequence>
            <xsd:attribute
```

```
                ref="soapenc:arrayType"
                w:arrayType="_SE_ArrayOfint"/>
        </xsd:restriction>
    </xsd:complexContent>
</xsd:complexType>

<xsd:complexType name="ArrayOfint" >
        <xsd:sequence >
            <xsd:element
                name="item" type="xsd:int"
                minOccurs="5" maxOccurs="5"/>
        </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="matrix" >
        <xsd:sequence >
            <xsd:element
                name="item1" type="ArrayOfint"
                minOccurs="3" maxOccurs="3"/>
        </xsd:sequence>
</xsd:complexType>
```

1.2.7.7 TypeCode

A typecode is composed of two parts: a URL to the WSDL descriptive document and the typename (fully scoped) within the “tns” target namespace of the WSDL document. In the case of primitive types, these will just refer directly to the XML Schema specification.

```
<xsd:complexType name="CORBA.TypeCode">
    <xsd:sequence>
        <xsd:element
                name="definition" type="xsd:anyURI"
                maxOccurs="1" minOccurs="1"/>
        <xsd:element
                name="typename" type="xsd:string"
                maxOccurs="1" minOccurs="1"/>
    </xsd:sequence>
</xsd:complexType>
```

1.2.7.8 Any

A CORBA::Any value is represented by a typecode, together with value represented as an xsd:anyType. The user may interrogate the typecode in order to coerce/interpret the value.

```
<xsd:complexType name="CORBA.Any">
    <xsd:sequence>
        <xsd:element
                name="type" type="CORBA.TypeCode"
                maxOccurs="1" minOccurs="1"/>
        <xsd:element
                name="value" type="xsd:anyType"
                maxOccurs="1" minOccurs="1"/>
```

```
    </xsd:sequence>
</xsd:complexType>
```

1.2.7.9 Fixed

CORBA fixed types are mapped to the XML Schema “decimal” type, with appropriate restrictions according to the original IDL (the “totalDigits” and “fractionDigits” attributes will be set appropriately). For example:

```
// IDL
typedef fixed<10,2> MyFixed
```

this maps to:

```
<!-- WSDL -->
<xsd:simpleType name=”MyFixed”>
    <xsd:restriction base=”xsd:decimal”>
    <xsd:totalDigits
        value=”10”/>
    <xsd:fractionDigits
         value=”2”
        fixed=”true”/>
    </xsd:restriction>
</xsd:simpleType>
```

1.2.7.10 ValueType

Value types in CORBA are mapped the same way as structures in WSDL. The value type WSDL will always be generated, but it is up to individual implementations to decide whether they will support the processing of these types. For example, DII/DSI bridges will be unable to process value types. An implementation may reject a message containing a value type parameter, raising a **NO_IMPLEMENT** exception with a standard minor code, as follows:

Minor Code	Description
9	Valuetypes not supported by CORBA-WSDL/SOAP implementation.
10	Valuetype sharing not supported by CORBA-WSDL/SOAP implementation.

Value type inheritance is supported by expanding the inheritance graph to include all inherited public and private state members in the representation for the inheriting value type. The state members from the inherited value type appear in the mapped struct before the members added in the inheriting value type definition.

The following examples show how each value type is mapped.

valuetype

Basic valuetypes are mapped to structs, as noted above. Both Public and Private fields are mapped. For example, consider the following:

```
// IDL
valuetype sampleX {
      public short a;
      private long b;
}
```

This maps to:

```
<xsd:complexType name="sampleX">
    <xsd:sequence>
        <xsd:element name="a" type="xsd:short" maxOccurs="1" minOccurs="1"/>
        <xsd:element name="b" type="xsd:int" maxOccurs="1" minOccurs="1"/>
    </xsd:sequence>
    <xsd:attribute name=id" type="xsd:ID" use="optional" />
    <!-- id  must be present if value type instance is shared (i.e.,
        referenced as a struct or value type member using xml  IDREF)-->
</xsd:complexType>
```

If run time support for value type sharing is not claimed, then this attribute will not be supported, and will never be present.

If run time support for value type sharing is claimed, then the id attribute:

- must be present in all mapped value type element instances, which are themselves “referenced” as a member of another value type or instance.
- may be present in any mapped value type element.

valuebox

Valueboxes for primitive types are mapped to a struct with a single member (called “value”), whose data type is mapped according to the primitive mapping table given in Section 1.2.6, “Primitive Types,” on page 1-6.

Valueboxes for sequences, arrays, and structs are mapped similarly to a struct with a single member (called “value”), whose data type will be the XML schema type defined by the appropriate mapping for that sequence, array, or struct.

Mapping of value types as struct or value type members

IDL value types can have recursive definitions, allowing complex structures, such as graphs, to be represented. This occurs whenever a value type has a state member that is the same type as its containing value type. Such cycles in an instance graph can be “chained” through a series of value types, which contain members of each other’s own type.

Due to its complexity, support at run time for mapping shared value types is a separate, optional, conformance point for this specification.

Whenever an IDL struct, sequence, or value type includes a member that is a value type, the corresponding sequence element for the value type member must be mapped to an xsd:choice element, containing either:

- an element that is a value of that valuetype or,
- an element that is a reference to a value type instance included in the same XML document.

The reference element has its element name prefixed with "_REF_", a type name of "corba:_VALREF", and includes an attribute called "ref" of type xsd:IDREF).

If value type sharing is not supported by the run-time implementation, then reference elements of type _VALREF will not be present. The mapping of value type members to the xsd:choice is always generated at translation time.

For use in references to value type instances, the following schema is defined in the corba namespace:

```
<xsd:complexType name="_VALREF"
    <xsd:attribute name="ref" type="xsd:IDREF" use="optional">
    <!-- empty attribute used for null semantics,
        i.e., value graph end nodes -->
</xsd:complexType>
```

Given the following recursive example:

```
valuetype WeightedBinaryTree {

    // state definition
    public unsigned long weight;
    public WeightedBinaryTree left;
    public WeightedBinaryTree right;

    // initializer
    factory init(in unsigned long w);

    // local operations
    WeightSeq pre_order();
    WeightSeq post_order();
};
```

The mapping would generate the following schema (assuming tns is prefix for target namespace and corba is prefix for corba mapping constructs name space):

```
<xsd:complexType name="WeightedBinaryTree" >
    <xsd:sequence>
        <xsd:element name="weight" type="xsd:integer" />
        <xsd:choice>
            <xsd:element name="left" type="tns.WeightedBinaryTree" />
            <xsd:element name="_REF_left" type="corba:_VALREF" />
        </xsd:choice>
        <xsd:choice>
            <xsd:element name="right" type="tns.WeightedBinaryTree" />
            <xsd:element name="_REF_right" type="corba:_VALREF" />
```

```
        </xsd:choice>
    </xsd:sequence>
    <xsd:attribute> name=id" type="xsd:ID" use="optional" />
        <!-- id attribute must be present if referenced by another
            node in graph -->
</xsd:complexType>
```

1.2.8 Interfaces

An IDL interface is actually three distinct constructs: a namespace for declaring types, a grouping operator for binding operations, and a type.

1.2.8.1 Interface as Namespace

Structures, Unions, and Typedefs are allowed to be constructed within an interface scope. As such, this use of the IDL interface is mapped in the same manner as the module construct.

```
// IDL
interface SomeInterface {
    typedef long Foo;
};
```

This IDL interface is mapped to the schema definition:

```
<xsd:simpleType name="SomeInterface.Foo">
    <xsd:restriction base="xsd:int"/>
</xsd:simpleType>
```

1.2.8.2 Interface as Binding Operations

The IDL interface construct also creates a grouping of operations. This use of the interface is mapped to the WSDL portType. The operations are mapped to the WSDL message type.

Each operation maps to three messages: an invocation, a response, and a fault message for potential system exceptions, except in the case of oneway operations, where there is no response message. Message names must be fully scoped (including all module names and the name of the enclosing interface). The invocation message consists of all the "in" and "inout" arguments. The response message contains the return value if one is present in the IDL (called "_return" to avoid name clashes), together with all the "out" and "inout" arguments.

System exception messages always return a CORBA system exception, which is defined as follows in WSDL:

```
<!-- WSDL -->
<simpleType name="CORBA.completion_status">
    <xsd:restriction base="xsd:string">
        <xsd:enumeration value="COMPLETED_YES"/>
```

```
        <xsd:enumeration value=”COMPLETED_NO”/>
        <xsd:enumeration value=”COMPLETED_MAYBE”/>
    </xsd:restriction>
</simpleType>

<xsd:complexType name="CORBA.SystemException">
    <xsd:sequence>
        <xsd:element
        name="minor" type="xsd:unsignedInt"
                maxOccurs="1" minOccurs="1"/>
        <xsd:element
                name="completion_status" type="CORBA.completion_status"
                maxOccurs="1" minOccurs="1"/>
    </xsd:sequence>
</xsd:complexType>
```

A System Exception message is also defined in the “CORBA” namespace, for brevity:

```
    <message name="CORBA.SystemExceptionMessage" >
        <part name="_return" type="CORBA.SystemException"/>
    </message>
```

If arrays are used in the signature of an IDL operation, then there has to be two WSDL operations defined for that IDL operation: one as specified for the simple case in “Simple form of Mapping Operations” and another tailored for use with Soap Encoding.

For IDL operations that do not use IDL sequence or array in their signatures, the simple mapping specified in “Simple form of Mapping Operations” suffices.

Simple form of Mapping Operations

By example:

```
// IDL
interface SomeInterface {
    long bar(in float pi);
};
```

is mapped to the WSDL definition:

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="SomeInterfaceExample"
        targetNamespace="http://www.omg.org/IDL-Mapped/"
        xmlns:tns="http://www.omg.org/IDL-Mapped/"
        xmlns:corba="http://www.omg.org/IDL-WSDL/1.0/"
        xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
        xmlns="http://schemas.xmlsoap.org/wsdl/"
        xmlns:xsd="http://www.w3.org/2001/XMLSchema">

    <import
        namespace="http://www.omg.org/IDL-WSDL/1.0/" location="../corba/corba.wsdl"/>

    <!-- Messages related to port: SomeInterface -->
    <message name="SomeInterface.bar">
        <part name="pi" type="xsd:float"/>
```

```
    </message>
    <message name="SomeInterface.barResponse">
        <part name="_return" type="xsd:int"/>
    </message>

    <!-- port type for SomeInterface -->
    <portType name="SomeInterface">
        <operation name="bar">
            <input message="tns:SomeInterface.bar"/>
            <output message="tns:SomeInterface.barResponse"/>
            <fault name="CORBA.SystemException"
message="corba:CORBA.SystemExceptionMessage"/>
        </operation>
    </portType>
</definitions>
```

Two message types are created. One message type is declared for the input parameters and one type created for the output parameters. A portType is also declared, in this case SomeInterface. The portType binds the input message and output message together into the contract.

Attributes are mapped to get/set accessor operations. Read-only attributes generate a single “get” operation, whereas read-write attributes generate both “get” and “set” forms. “Set” operations have a single parameter, of the same type as the attribute, and a void return type. “Get” operations have an input message with no parts, and the return type is the same as the attribute. The names of these accessor operations is generated by prefixing either “_get_” or “_set_” to the name of the attribute, as appropriate.

For example:

```
// IDL
interface MyAttrs {
    attribute string strAttr;
    readonly attribute long longAttr;
};
```

is mapped to the WSDL definition:

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="MyAttrExample"
        targetNamespace="http://www.omg.org/IDL-Mapped/"
        xmlns:tns="http://www.omg.org/IDL-Mapped/"
        xmlns:corba="http://www.omg.org/IDL-WSDL/1.0/"
        xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
        xmlns="http://schemas.xmlsoap.org/wsdl/"
        xmlns:xsd="http://www.w3.org/2001/XMLSchema">

    <import
        namespace="http://www.omg.org/IDL-WSDL/1.0/" location="../corba/corba.wsdl"/>

    <!-- Messages related to portType: MyAttrs -->
    <message name="MyAttrs._get_strAttr"/>
    <message name="MyAttrs._get_strAttrResponse" >
        <part name="_return" type="xsd:string"/>
```

```
    </message>
    <message name="MyAttrs._set_strAttr" >
        <part name="value" type="xsd:string"/>
    </message>
    <message name="MyAttrs._get_longAttr"/>
    <message name="MyAttrs._get_longAttrResponse" >
        <part name="_return" type="xsd:int"/>
    </message>

    <!-- portType for MyAttrs -->
    <portType name="MyAttrs" >
        <operation name="_get_strAttr" >
            <input message="tns:MyAttrs._get_strAttr"/>
            <output message="tns:MyAttrs._get_strAttrResponse"/>
            <fault name="CORBA.SystemException"
message="corba:CORBA.SystemExceptionMessage"/>
        </operation>
        <operation name="_set_strAttr" >
            <input message="tns:MyAttrs._set_strAttr"/>
            <fault name="CORBA.SystemException"
message="corba:CORBA.SystemExceptionMessage"/>
        </operation>
        <operation name="_get_longAttr">
            <input message="tns:MyAttrs._get_longAttr"/>
            <output message="tns:MyAttrs._get_longAttrResponse"/>
            <fault name="CORBA.SystemException"
message="corba:CORBA.SystemExceptionMessage"/>
        </operation>
    </portType>
</definitions>
```

Extra Productions for Operations having IDL sequence or array for parameters or return values

For IDL operations that have IDL sequence or array types in their signature (at any level of nesting depth in the type for an operation parameter or return value), there will need to be additional definitions corresponding to a second port type for use with soap bindings that support style=rpc and use=encoded.

For example, given the previous definition of **longSeq**

```
interface SomeInterface2 {
    longSeq bar(in float pi);
};
```

is mapped to the WSDL definition:

```
<?xml version="1.0" encoding="utf-8"?>
<w:definitions name="SomeInterface2Example"
        targetNamespace="http://www.omg.org/IDL-Mapped/"
        xmlns:tns="http://www.omg.org/IDL-Mapped/"
        xmlns:corba="http://www.omg.org/IDL-WSDL/1.0/"
        xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
        xmlns:w="http://schemas.xmlsoap.org/wsdl/"
        xmlns:xsd="http://www.w3.org/2001/XMLSchema"
        xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/">
```

```
    <w:import
        namespace="http://www.omg.org/IDL-WSDL/1.0/" location="../corba/corba.wsdl"/>

    <w:types>
        <xsd:schema targetNamespace="http://www.omg.org/IDL-Mapped/">
            <!-- schema for soap encoding -->
            <xsd:complexType name="_SE_longSeq">
                <xsd:complexContent>
                    <xsd:restriction base="soapenc:Array">
                        <xsd:sequence>
                            <xsd:element name="item" minOccurs="0"
maxOccurs="unbounded" type="xsd:int"/>
                        </xsd:sequence>
                        <!-- the following line, using the arrayType attribute from the wsdl
namespace,
                         is specified to be used by WSDL1.1 for soapEncoding Array.  It is a strange
incantation,
                        and the need to namespace qualify this attribute is why this file does not use wsdl
as the default namespace -->
                        <xsd:attribute ref="soapenc:arrayType" w:arrayType="xsd:int"/>
                    </xsd:restriction>
                </xsd:complexContent>
            <!-- schema for WS-I compliant mapping -->
            </xsd:complexType>
            <xsd:complexType name="longSeq">
                <xsd:sequence>
                    <xsd:element name="item" minOccurs="0" maxOccurs="unbounded"
type="xsd:int"/>
                </xsd:sequence>
            </xsd:complexType>
        </xsd:schema>
    </w:types>

    <!-- Messages related to port: SomeInterface2 -->
    <w:message name="SomeInterface2.bar">
        <w:part name="pi" type="xsd:float"/>
    </w:message>
    <w:message name="SomeInterface2.barResponse">
        <w:part name="_return" type="tns:longSeq"/>
    </w:message>

<!-- Messages related to port: SomeInterface2 for soap encoding -->
    <w:message name="_SE_SomeInterface2.barResponse">
        <w:part name="_return" type="tns:_SE_longSeq"/>
    </w:message>

    <!-- port for SomeInterface2 -->
    <w:portType name="SomeInterface2">
        <w:operation name="bar">
            <w:input message="tns:SomeInterface2.bar"/>
            <w:output message="tns:SomeInterface2.barResponse"/>
            <w:fault name="CORBA.SystemException"
message="corba:CORBA.SystemExceptionMessage"/>
        </w:operation>
    </w:portType>
```

```
    <!-- port for SomeInterface2 with soap encoding -->
    <w:portType name="_SE_SomeInterface2">
        <w:operation name="bar">
            <w:input message="tns:SomeInterface2.bar"/>
            <w:output message="tns:_SE_SomeInterface2.barResponse"/>
            <w:fault name="CORBA.SystemException"
message="corba:CORBA.SystemExceptionMessage"/>
        </w:operation>
    </w:portType>
</w:definitions
```

1.2.8.3 *Interface as a Type*

As a type, an interface reference is expanded identically to the mapping for **CORBA::Object**.

1.2.8.4 *Mapping of interface inheritance*

In IDL, interfaces support multiple interface inheritance. WSDL does not have this construct and therefore interface inheritance is mapped as repetition of the operations declared in the parenting interfaces. Types declared within the parent interface scope are not repeated as that type space is available to the derived interfaces.

Thus:

```
// IDL
interface BaseInterface {
    typedef long Foo;
    long bar(in Foo pi);
};

interface DerivedInterface : BaseInterface {
    long baz(in Foo po);
};
```

is mapped to:

```
<?xml version="1.0" encoding="utf-8"?>
<definitions name="inheritanceExample"
        targetNamespace="http://www.omg.org/IDL-Mapped/"
        xmlns:tns="http://www.omg.org/IDL-Mapped/"
        xmlns:corba="http://www.omg.org/IDL-WSDL/1.0/"
        xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
        xmlns="http://schemas.xmlsoap.org/wsdl/"
        xmlns:xsd="http://www.w3.org/2001/XMLSchema">

    <import
        namespace=" http://www.omg.org/IDL-WSDL/1.0/" location="../corba/corba.wsdl"/>

    <types>
        <xsd:schema targetNamespace="http://www.omg.org/IDL-Mapped/">
            <xsd:simpleType name="BaseInterface.Foo">
                <xsd:restriction base="xsd:float"/>
```

```
            </xsd:simpleType>
        </xsd:schema>
    </types>

    <!-- Messages related to port: BaseInterface -->
    <message name="BaseInterface.bar">
        <part name="pi" type="tns:BaseInterface.Foo"/>
    </message>
    <message name="BaseInterface.barResponse">
        <part name="_return" type="xsd:int"/>
    </message>

    <!-- Messages related to port: DerivedInterface -->
    <message name="DerivedInterface.baz">
        <part name="po" type="tns:BaseInterface.Foo"/>
    </message>
    <message name="DerivedInterface.bazResponse">
        <part name="_return" type="xsd:int"/>
    </message>

    <!-- port for BaseInterface -->
    <portType name="BaseInterface">
        <operation name="bar">
            <input message="tns:BaseInterface.bar"/>
            <output message="tns:BaseInterface.barResponse"/>
            <fault name="CORBA.SystemException"
message="corba:CORBA.SystemExceptionMessage"/>
        </operation>
    </portType>

    <!-- port for DerivedInterface -->
    <portType name="DerivedInterface">
        <operation name="bar">
            <input message="tns:BaseInterface.bar"/>
            <output message="tns:BaseInterface.barResponse"/>
            <fault name="CORBA.SystemException"
message="corba:CORBA.SystemExceptionMessage"/>
        </operation>
        <operation name="DerivedInterface.baz">
            <input message="tns:DerivedInterface.baz"/>
            <output message="tns:DerivedInterface.bazResponse"/>
            <fault name="CORBA.SystemException"
message="corba:CORBA.SystemExceptionMessage"/>
        </operation>
    </portType>
</definitions>
```

1.2.8.5 Exceptions

IDL exceptions are mapped as constructed types, like structs. However, in IDL it can only be used in a raises clause of an operation (i.e., you cannot pass an exception as a parameter, or use it as a type elsewhere).

For each IDL operation with a “raises” clause, a corresponding fault message is generated for each exception listed. These fault messages are named after the exception (fully qualified, as any other message), and consist of a single element, named “exception,” which is of the same type as the mapped complex type corresponding to the exception definition.

For example, consider the following IDL:

```
// IDL
module Example {
    exception UnknownError {};
    exception BadRecord {
        string why;
    };
    exception RottenApple {
        long numberOfWorms;
    };
    interface SomeInterface {
        long bar(in float pi) raises (BadRecord, UnknownError);
    };
};
```

This is mapped to:

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="exceptionsExample"
        targetNamespace="http://www.omg.org/IDL-Mapped/"
        xmlns:tns="http://www.omg.org/IDL-Mapped/"
        xmlns:corba="http://www.omg.org/IDL-WSDL/1.0/"
        xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
        xmlns="http://schemas.xmlsoap.org/wsdl/"
        xmlns:xsd="http://www.w3.org/2001/XMLSchema">

    <import
        namespace=" http://www.omg.org/IDL-WSDL/1.0/" location="../corba/corba.wsdl"/>

    <types>

<!-- Exception type definitions -->
        <xsd:schema targetNamespace="http://www.omg.org/IDL-Mapped/">
            <!-- Exception definitions -->
            <xsd:complexType name="Example.BadRecord">
                <xsd:sequence>
                    <xsd:element name="why" type="xsd:string" maxOccurs="1" minOccurs="1"/>
                </xsd:sequence>
            </xsd:complexType>
            <xsd:complexType name="Example.RottenApple">
                <xsd:sequence>
                    <xsd:element name="numberOfWorms" type="xsd:int" maxOccurs="1"
minOccurs="1"/>
                </xsd:sequence>
            </xsd:complexType>
            <xsd:complexType name="Example.UnknownError">
                <xsd:sequence>
                    </xsd:sequence>
```

```
            </xsd:complexType>
        </xsd:schema>
    </types>

    <!-- Messages related to interface Example.SomeInterface -->
    <message name="Example.SomeInterface.bar">
        <part name="pi" type="xsd:float"/>
    </message>
    <message name="Example.SomeInterface.barResponse">
        <part name="_return" type="xsd:int"/>
    </message>
    <message name="_exception.Example.BadRecord">
        <part name="exception" type="tns:Example.BadRecord"/>
    </message>
    <message name="_exception.Example.UnknownError">
        <part name="exception" type="tns:Example.UnknownError"/>
    </message>

    <!-- portType for Example.SomeInterface -->
    <portType name="Example.SomeInterface">
        <operation name="bar">
            <input message="tns:Example.SomeInterface.bar"/>
            <output message="tns:Example.SomeInterface.barResponse"/>
            <fault name="Example.BadRecord" message="tns:_exception.Example.BadRecord"/>
            <fault name="Example.UnknownError"
message="tns:_exception.Example.UnknownError"/>
            <fault name="CORBA.SystemException"
message="corba:CORBA.SystemExceptionMessage"/>
        </operation>
    </portType>
</definitions>
```

1.2.9 SOAP Bindings

Having specified all the data types, messages, and ports used in a web service, it is then necessary to specify the binding to SOAP. This mapping specification requires generation of two forms of soap binding:

- SOAP RPC-style binding (style=rpc, use=encoded) to be used when WS-I conformance is not required. The name of the binding will consist of the fully-qualified name of the interface (as used in the portType), together with the prefix "_SE_."
- WS-I Basic profile conformant: (style=rpc, use=literal) to be used when WS-I conformance is required.

While both forms of soap binding must be generated by the mapping, run-time conformance to these separate binding styles form two separate conformance classes for this specification.

For example, consider the following IDL:

```
interface foo {
    void query(in string s);
};
```

This would have the following two WSDL bindings (shown in one wsdl document);

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="fooBindingExample"
        xmlns="http://schemas.xmlsoap.org/wsdl/"
        xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
        xmlns:http="http://schemas.xmlsoap.org/wsdl/http/"
        xmlns:xsd="http://www.w3.org/2001/XMLSchema"
        xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
        xmlns:mime="http://schemas.xmlsoap.org/wsdl/mime/"
        xmlns:corba="http://www.omg.org/IDL-WSDL/1.0/"
        xmlns:tns="http://www.omg.org/IDL-Mapped/"
        targetNamespace="http://www.omg.org/IDL-Mapped/" >

    <import
        namespace="http://www.omg.org/IDL-WSDL/1.0/" location="../corba/corba.wsdl"/>

<!-- messages for foo --
<message name="foo.query">
        <part name="s" type="xsd:string"/>
    </message>
    <message name="foo.queryResponse">
    </message>

    <!-- portType for foo -->
    <portType name="foo">
        <operation name="query">
            <input message="tns:foo.query"/>
            <output message="tns:foo.queryResponse"/>
            <fault name="CORBA.SystemException"
message="corba:CORBA.SystemExceptionMessage"/>
        </operation>
    </portType>

<!-- rpc encoded binding for foo -->
    <binding name="_SE_fooBinding" type="tns:foo">
        <soap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http"/>
        <operation name="query">
            <soap:operation soapAction="foo#query"/>
            <input>
                <soap:body use="encoded"
encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
namespace="http://www.omg.org/IDL-WSDL/1.0/"/>
            </input>
            <output>
                <soap:body use="encoded"
encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
namespace="http://www.omg.org/IDL-WSDL/1.0/"/>
            </output>
            <fault name="CORBA.SystemException"/>
        </operation>
    </binding>

<!-- WS-I compliant binding for foo -->
    <binding name="fooBinding" type="tns:foo">
        <soap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http"/>
        <operation name="query">
```

```
            <soap:operation soapAction="foo#query"/>
            <input>
                <soap:body use="literal" namespace="http://www.omg.org/IDL-WSDL/1.0/"/>
            </input>
            <output>
                <soap:body use="literal" namespace="http://www.omg.org/IDL-WSDL/1.0/"/>
            </output>
            <fault name="CORBA.SystemException"/>
        </operation>
    </binding>
</definitions>
```

Another example, which has an operation with **longSeq** return value:

```
interface SomeInterface2 {
    longSeq bar(in float pi);
};
```

This would have the following two WSDL bindings (note the port type for the soap encoding binding in this example has prefix "_SE_"):

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="SomeInterface2BindingExample"
xmlns="http://schemas.xmlsoap.org/wsdl/" xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
xmlns:http="http://schemas.xmlsoap.org/wsdl/http/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:mime="http://schemas.xmlsoap.org/wsdl/mime/" xmlns:corba="http://www.omg.org/IDL-
WSDL/1.0/" xmlns:tns="http://www.omg.org/IDL-Mapped/"
targetNamespace="http://www.omg.org/IDL-Mapped/" >

    <import namespace="http://www.omg.org/IDL-Mapped/"
location="../tns/SomeInterface2Example.wsdl"/>

<!-- rpc encoded binding or SomeInterface2 -->
    <binding name="_SE_SomeInterface2Binding" type="tns:_SE_SomeInterface2">
        <soap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http"/>
        <operation name="bar">
            <soap:operation soapAction="SomeInterface2#bar"/>
            <input>
                <soap:body use="encoded"
encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
namespace="http://www.omg.org/IDL-WSDL/1.0/"/>
            </input>
            <output>
                <soap:body use="encoded"
encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
namespace="http://www.omg.org/IDL-WSDL/1.0/"/>
            </output>
            <fault name="CORBA.SystemException"/>
        </operation>
    </binding>

    <!-- WS-I compliant binding for SomeInterface2
    <binding name="SomeInterface2Binding" type="tns:SomeInterface2">
        <soap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http"/>
```

```
        <operation name="bar">
            <soap:operation soapAction="SomeInterface2#foo"/>
            <input>
                <soap:body use="literal" namespace="http://www.omg.org/IDL-WSDL/1.0/"/>
            </input>
            <output>
                <soap:body use="literal" namespace="http://www.omg.org/IDL-WSDL/1.0/"/>
            </output>
            <fault name="CORBA.SystemException"/>
        </operation>
    </binding>
</definitions>
```

1.2.10 Service Endpoints

The web service endpoints will vary from implementation to implementation, and will not be discussed here.

This depends on the topology of the web service/CORBA bridge and the underlying CORBA servers, which is not something that can be captured from the IDL. Consequently, this is all outside of the scope of this specification.

1.2.11 WSDL namespace for CORBA

The following is a wsdl description, which is a normative description of the CORBA namespace.

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

    WSDL for IDL to WSDL CORBA Namespace

   Name: corba.wsdl
-->
<definitions name="corba"
        xmlns="http://schemas.xmlsoap.org/wsdl/"
        xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
        xmlns:xsd="http://www.w3.org/2001/XMLSchema"
        xmlns:corba="http://www.omg.org/IDL-WSDL/1.0/"
        targetNamespace="http://www.omg.org/IDL-WSDL/1.0/" >

    <types>
        <xsd:schema targetNamespace="http://www.omg.org/IDL-WSDL/1.0/">
            <xsd:element name="SourceIDL" >
                <xsd:annotation>
                    <xsd:documentation>IDL/WSDL Mapping Info</xsd:documentation>
                </xsd:annotation>
                <xsd:complexType>
                    <xsd:sequence>
                        <xsd:element name="source" type="xsd:string" minOccurs="1"
maxOccurs="1"/>
                        <xsd:element name="version" type="xsd:string" minOccurs="1"
maxOccurs="1"/>
                    </xsd:sequence>
```

```
            </xsd:complexType>
        </xsd:element>

        <xsd:element name="SourceRepositoryID" >
            <xsd:annotation>
                <xsd:documentation>IDL Mapped Repository ID </xsd:documentation>
            </xsd:annotation>
            <xsd:complexType>
                <xsd:sequence>
                    <xsd:element name="repositoryID" type="xsd:string" minOccurs="1"
maxOccurs="1"/>
                    <xsd:element name="version" type="xsd:string" minOccurs="1"
maxOccurs="1"/>
                </xsd:sequence>
            </xsd:complexType>
        </xsd:element>

        <xsd:complexType name="ObjectReference">
            <xsd:sequence>
                <xsd:element name="url" type="xsd:anyURI"  minOccurs="1"
maxOccurs="unbounded"/>
            </xsd:sequence>
        </xsd:complexType>

        <xsd:complexType name="CORBA.TypeCode">
            <xsd:sequence>
                <xsd:element name="definition" type="xsd:anyURI" maxOccurs="1"
minOccurs="1"/>
                <xsd:element name="typename" type="xsd:string" maxOccurs="1"
minOccurs="1"/>
            </xsd:sequence>
        </xsd:complexType>
        <xsd:complexType name="CORBA.Any">
            <xsd:sequence>
                <xsd:element name="type" type="corba:CORBA.TypeCode" maxOccurs="1"
minOccurs="1"/>
                <xsd:element name="value" type="xsd:anyType" maxOccurs="1"
minOccurs="1"/>
            </xsd:sequence>
        </xsd:complexType>
        <xsd:simpleType name="CORBA.completion_status">
            <xsd:restriction base="xsd:string">
                <xsd:enumeration value="COMPLETED_YES"/>
                <xsd:enumeration value="COMPLETED_NO"/>
                <xsd:enumeration value="COMPLETED_MAYBE"/>
            </xsd:restriction>
        </xsd:simpleType>
        <xsd:complexType name="CORBA.SystemException">
            <xsd:sequence>
                <xsd:element name="minor" type="xsd:unsignedInt" maxOccurs="1"
minOccurs="1"/>
                <xsd:element name="completion_status"
type="corba:CORBA.completion_status" maxOccurs="1" minOccurs="1"/>
            </xsd:sequence>
        </xsd:complexType>
```

```
            <xsd:complexType name=”_VALREF”
                <xsd:attribute name=”ref” type=”xsd:IDREF” use=”optional”>
                <!-- empty attribute used for null semantics, i.e., value graph end nodes -->
            </xsd:complexType>
        </xsd:schema>
    </types>
    <message name="CORBA.SystemExceptionMessage">
        <part name="_return" type="corba:CORBA.SystemException"/>
    </message>
</definitions>
```