The gSOAP DOM Parser (Beta)

Robert van Engelen
Genivia inc.
engelen@genivia.com

January 5, 2004

Contents

1 Introduction

The gSOAP DOM parser is the first DOM parser of its kind that introduces a new concept, namely the ability to create a DOM structure with both generic DOM nodes and nodes that consist of native C/C++ application data structures. The gSOAP XML parser is integrated in the DOM parser to deserialize native C/C++ application data types from the contents of an XML document to populate the DOM with nodes that are application-specific. This unique DOM parsing approach enables developers to automatically extract application data from any XML document. In contrast, other DOM parsers provide a generic DOM tree representation only. SAX parsing offers another approach to application-driven data processing. While SAX parsing requires application developers to write the data extraction methods, this DOM parser is fully automatic.

The DOM parser is a relatively simple parser that supports the XML 1.0 standards including XML namespaces. It has been specifically designed to integrate with gSOAP. The DOM parser can be used with gSOAP to support the exchange of generic XML documents in SOAP/XML for example.

The gSOAP compiler-generated XML parsers are validating. However, this DOM parser does not attempt to validate documents against schemas. Additional features are planned, see Section 8.

A C++ implementation of the DOM parser is available as well as a pure C version. The C++ implementation offers more convenient DOM parsing with iostream operator overloading, DOM constructors, and DOM tree iterators.

2 Example 1: Extracting Application Data From an XML Document

The following example illustrates the gSOAP DOM parser's capabilities to automatically extract application data from an XML document.

Suppose our goal is to extract the contents of `<product>` elements from certain XML documents and store this data in C++ class instances that somehow match the `<product>` element schema layout. We assume that the `<product>` element has the following sub-elements according to its schema: `name` of type `xsd:QName` (qualified name), `manufacturer` of type `xsd:string`, `SKU` of type `xsd:int`, `price` of type `xsd:double`, and `description` of type `xsd:anyType` (i.e. a generic XML document structure). The `<product>` element also has an optional attribute `Id` of type `xsd:ID` to enable cross referencing. The following schema describes the `<product>` element and type, where the namespace of `<product>` is assumed to be `http://domain/schemas/product.xsd`:

```
<schema targetNamespace="http://domain/schemas/product.xsd"
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:ns="http://domain/schemas/product.xsd"
  xmlns="http://www.w3.org/2001/XMLSchema"
  elementFormDefault="unqualified"
  attributeFormDefault="unqualified">
  <element name="product" type="ns:product">
  <complexType name="product">
    <sequence>
      <element name="name" type="xsd:QName" minOccurs="1" maxOccurs="1"
```

```
nillable="true"/>
     <element name="manufacturer" type="xsd:string" minOccurs="0" max-
Occurs="1" nillable="true"/>
     <element name="SKU" type="xsd:int" minOccurs="1" maxOccurs="1"/>
     <element name="price" type="xsd:double" minOccurs="1" maxOccurs="1"/>
     <element name="description" type="xsd:anyType" minOccurs="0" max-
Occurs="1" nillable="true"/>
   </sequence>
   <attribute name="Id" type="xsd:ID" use="optional"/>
  </complexType>
</schema>
```

The class ns__product declaration is:

```
#import "dom++.h" // import DOM definitions (defines xsd:anyType)
typedef char *_xsd__QName; // define xsd:QName
typedef char *_xsd__string; // define xsd:string
typedef int _xsd__int; // define xsd:int
typedef double _xsd__double; // define xsd:double
typedef char *_xsd__ID; // define xsd:ID
//gsoap ns schema namespace: http://domain/schemas/product.xsd
class ns__product
{ public:
  _xsd__QName name;
  _xsd__string manufacturer 0;
  _xsd__int SKU;
  _xsd__double price;
  xsd__anyType *description 0;
  @_xsd__ID Id 0;
  ns__product();
  ~ns__product();
  void unlink(struct soap *soap);
};
```

The class declaration can be obtained with the WSDL importer from a WSDL with a schema for the `<product>` type (we plan to enhance the WSDL importer to support the parsing of schemas without WSDL).

The ns__product class reflects the XML schema layout and properties of the `<product>` element and type. The built-in schema types `xsd:QName`, `xsd:string`, etc., are declared in gSOAP with **typedef**s.

We invoke the gSOAP compiler from the command line to generate the ns__product class (de)serializers (we assume that the class is declared in product.h):

```
soapcpp2 product.h
```

This produces the files soapStub.h, soapH.h, and soapC.cpp (among other files) with the ns__product (de)serializers. These files are needed to build our customized DOM parser. In addition, the file ns.xsd is generated which contains the XML schema of `ns:product`. This may come in handy.

Suppose we have an XML document that contains one or more `<product>` elements (within the appropriate XML namespace) and we want to extract these elements from the document in an application-specific format, i.e. as ns__product class instances. The following code parses the document from the standard input stream and then iterates over the DOM thereby printing the price of the deserialized ns__product instances:

```
#include "soapH.h"
#include <iostream.h>
int main()
{
  soap_dom_element document(soap_new()); // create a DOM with a new soap environment
  soap_set_imode(document.soap, SOAP_DOM_NODE); // DOM w/ application data nodes
  cin >> document; // parse XML
  for (soap_dom_iterator walker = document.find(SOAP_TYPE_ns__product); walker != docu-
ment.end(); ++walker)
  {
    ns__product *product = (ns__product*)(*walker).node;
    cout << product->name << " price=" << product->price << endl;
  }
  soap_destroy(document.soap); // delete deserialized DOM parts
  soap_end(document.soap); // clean up
  soap_done(document.soap); // detach soap environment
  free(document.soap); // free soap environment
  return 0;
}
struct Namespace namespaces[] =
{
  {"SOAP-ENV", "http://schemas.xmlsoap.org/soap/envelope/"},
  {"SOAP-ENC", "http://schemas.xmlsoap.org/soap/encoding/"},
  {"xsi", "http://www.w3.org/2001/XMLSchema-instance"},
  {"xsd", "http://www.w3.org/2001/XMLSchema"},
  {"ns", "http://domain/schemas/product.xsd"}, // the namespace of products
  {NULL, NULL}
};
```

The code is compiled and linked with dom++.cpp, stdsoap2.cpp, and soapC.cpp. The input-mode flag SOAP_DOM_NODE is used to force the DOM parser to use application-specific data type deserializers to parse the DOM and populate the nodes with C/C++ application types. (Note: when no flag is used, an appropriate deserializer MAY be used by gSOAP when an element contains an `id` attribute and gSOAP can determine the type from the `id` attribute reference and/or the `xsi:type` attribute of an element.) When we invoked the gSOAP compiler from the command line, it generated a deserializer for the ns__product class which is used to populate the DOM with instances. The type identifier of this class is SOAP_TYPE_ns__product (gSOAP generates type identifiers of the form SOAP_TYPE__*x* where *x* is the name of a type). The forward iterator document.find(SOAP_TYPE_ns__product) traverses the DOM and selects the ns__product nodes which we access with (ns__product*)(*walker).node.

As an example suppose we want to print the price of the products in the following XML document:

```
<?xml version="1.0" encoding="UTF-8"?>
<document
  xmlns:ns="http://domain/schemas/product.xsd"
  <ns:product Id="Z">
    <name>ns:Zoe</name>
    <manufacturer>Sesame Street</manufacturer>
    <SKU>123</SKU>
    <price>9.95</price>
  </ns:product>
  <m:product xmlns:m="http://domain/schemas/product.xsd">
    <name>m:Pluto</name>
    <SKU>567</SKU>
    <price>19.95</price>
    <description>This <i>lovely</i> doll is a <b>every</b> child's wish</description>
    <x:value xmlns:x="http://domain/schemas/value.xsd">12</x:value>
  </m:product>
  <product xmlns="http://otherdomain">
    <name>Gadget</name>
    <content xsi:type="y:value" xmlns:y="http://domain/schemas/value.xsd">3<content>
  </product>
</document>
```

Note that the last product `<product xmlns="http://otherdomain">` is not in the namespace of our products.

The program fragment shown above parses the document and then prints:

```
ns:Zoe price=9.95
ns:Pluto price=19.95
```

Note that the QName `m:Pluto` is in our namespace and therefore the namespace prefix `ns` replaces `m` in the document. Also note that certain elements are optional (`minOccurs="0"` in the schema and the `0`'s used in the class declaration).

The deserialized data of the DOM is removed with the soap_destroy() and soap_end() calls. To retain class instances and their data, you have to unlink the data references from gSOAP's deallocation chain with soap_unlink(soap, *pointer*), where *pointer* points to a class instance. Also the pointer-based data members need to be unlinked if you want to preserve their values. You can do this by adding an appropriate method to the class which calls soap_unlink(soap, this) and also calls soap_unlink() on all its pointer-based data members. For example:

```
ns__product::unlink(struct soap *soap)
{
  soap_unlink(soap, this);
  soap_unlink(soap, name);
  soap_unlink(soap, manufacturer);
  if (description)
    description->unlink();
  soap_unlink(soap, Id);
}
```

To deserialize primitive type values, you can use a typedef to define the element name and type without or without a namespace qualifier. The following gSOAP header file declares a `<value>` element and type:

```
#import "dom++.h"
typedef int v__value;
```

This assumes that all `<v:value>` elements in a document are integer valued. This header file is processed with the gSOAP compiler to produce the (de)serializers.

The code fragment below prints the integer contents of the `<v:value>` elements of a document:

```
#include "soapH.h"
#include <iostream.h>
int main()
{
   soap_dom_element document(soap_new()); // create a DOM with a new soap environment
   soap_set_imode(document.soap, SOAP_DOM_NODE); // DOM w/ application data
   cin >> document; // parse XML
   for (soap_dom_iterator walker = document.find(SOAP_TYPE_v__value); walker != document.end();
++walker)
      cout << *(v__value*)(*walker).node << endl;
   soap_destroy(document.soap); // delete DOM
   soap_end(document.soap); // clean up
   soap_done(document.soap); // detach soap environment
   free(document.soap); // free soap environment
   return 0;
}
struct Namespace namespaces[] =
{
   {"SOAP-ENV", "http://schemas.xmlsoap.org/soap/envelope/"},
   {"SOAP-ENC", "http://schemas.xmlsoap.org/soap/encoding/"},
   {"xsi", "http://www.w3.org/2001/XMLSchema-instance"},
   {"xsd", "http://www.w3.org/2001/XMLSchema"},
   {"ns", "http://domain/schemas/product.xsd"}, // the namespace of products
   {"v", "http://domain/schemas/value.xsd}, // the namespace of values
   {NULL, NULL}
};
```

The program fragment parses the example XML document shown in this section and then prints:

```
12
3
```

The first value was retrieved from the `<x:value>` element (with matching namespace) and the second value was retrieved from the `<content xsi:type="y:value">` element in the third product element, because the type `y:value` matches the `v:value` schema type.

To create a DOM that does not contain application-specific data structures is simple. Consider for example the following code that parses a DOM from the standard input stream and then copies it to the standard output stream:

```
...
soap_dom_element document(soap_new()); // create a DOM with a new soap environment
soap_set_imode(document.soap, SOAP_DOM_TREE); // DOM tree w/o application data
cin >> document; // parse
cout << document; // print it
soap_destroy(document.soap); // delete entire DOM
soap_end(document.soap); // clean up
soap_done(document.soap); // detach the soap environment
```

```
free(document.soap); // free the soap environment
...
```

In this example the input-mode flag SOAP_DOM_TREE forces the parser to construct a DOM only and ignore any application-specific data elements in the XML document.

The leading underscore (_) in the names of the _xsd__ types defined in the header file has a special meaning. When names of C/C++ types (i.e. **typedef**s, **struct**s, **class**es, **enum**s) are defined with a leading underscore, the XML elements that are defined with these types will not carry the xsi:type attribute in the XML document. That is, it makes the XML document untyped. The gSOAP DOM output will therefore omit the xsi:type attributes in the XML document.

You can eliminate the ns__ namespace prefix from the ns__product class, but doing so will force the XML parser to deserialize all

`<product>` elements from an XML document without namespace validation.

3 Example 2: Embedding Application Data in XML Documents

The following example illustrates the embedding of application data in XML documents. We use the ns__product class of Example 1 to create an XML document with a `<product>` element. The following code constructs an instance and binds it to the appropriate place in the DOM representation of the XML document:

```
#include "soapH.h"
#include <iostream.h>
int main()
{
  ns__product product;
  product.name = "ns:Zoe";
  product.manufacturer = "Sesame Street";
  product.SKU = 123;
  product.price = 9.95;
  product.description = NULL;
  product.Id = "Z";
  struct soap *soap = soap_new();
  soap_dom_element document(soap_new(), "urn:test", "myDocument");
  soap_dom_attribute myAttribute(document.soap, NULL, "myAttribute", "Y");
  soap_dom_element myElement(document.soap, NULL, "myElement", "X");
  document.add(myAttribute);
  document.add(myElement);
  document.add("http://domain/schemas/product.xsd", "product", product, SOAP_TYPE_ns__product);
  cout << document;
  soap_destroy(document.soap); // delete DOM
  soap_end(document.soap); // clean up
  soap_done(document.soap); // detach soap environment
  free(document.soap); // free soap environment
  return 0;
}
struct Namespace namespaces[] =
{
  {"SOAP-ENV", "http://schemas.xmlsoap.org/soap/envelope/"},
  {"SOAP-ENC", "http://schemas.xmlsoap.org/soap/encoding/"},
  {"xsi", "http://www.w3.org/2001/XMLSchema-instance"},
  {"xsd", "http://www.w3.org/2001/XMLSchema"},
  {"ns", "http://domain/schemas/product.xsd"}, // the namespace of products
  {NULL, NULL}
}
```

The code is linked with dom++.cpp, stdsoap2.cpp, and soapC.cpp. The program prints:

```
<?xml version="1.0" encoding="UTF-8"?>
<SOAP-DOM0:myDocument
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:ns="http://domain/schemas/product.xsd"
  xmlns:SOAP-DOM0="urn:test"
  myAttribute="Y">
  <myElement>X</myElement>
  <ns:product Id="Z">
    <name>ns:Zoe"</name>
    <manufacturer>Sesame Street</manufacturer>
    <SKU>123</SKU>
    <price>9.95</price>
  </ns:product>
</SOAP-DOM0:myDocument>
```

The root of the DOM is `<myDocument>` with namespace `urn:test`. The root has two sub elements: `<myElement>` and `<product>` with namespace `http://domain/schemas/product.xsd`. It also has an attribute `myAttribute="Y"`.

The global namespace mapping table namespaces[] contains the namespace bindings that we intend to use in our application. That is, it should contain the standard namespaces for SOAP/XML and XML schemas.

You can also create non-global tables and assign them to the gSOAP environment when the need arises:

```
struct soap *soap = soap_new();
soap->namespaces = myNamespaces;
```

where myNamespaces should be a namespace mapping table.

To eliminate the use of these tables, use:

```
struct soap *soap = soap_new();
soap->namespaces = NULL;
```

And compile with -DWITH_NONAMESPACES. However, doing so will change the behavior of the DOM parser when parsing documents. Firstly, `xsi:type` information is ignored, so application data cannot be retrieved from elements with `xsi:types`. In addition, QNames cannot be handled. Normally the parser maps QNames to the appropriate namespace prefixes where the mapping is defined in the table. Without the table with bindings to resolve the QName values are of the form `"namespaceURI":name`.

4 Example 3: SOAP/XML

The gSOAP DOM parser can be used to implement SOAP/XML clients and services that support SOAP document encoding. The following gSOAP header file uses the xsd__anyType (the external DOM XML serializer) to exchange generic XML documents as SOAP/XML service parameters:

```
#import "dom++.h"
//gsoap ns service name: docu
//gsoap ns service namespace: http://domain/services/docu.wsdl
//gsoap ns service encoding: literal
//gsoap ns schema namespace: urn:docu
int ns__docuXchange(xsd__anyType in, xsd__anyType *out);
```

The client-side and server-side codes are straightforward implementations of the usual gSOAP proxy/stub and skeleton implementations for SOAP/XML Web services (not shown).

The following example illustrates the use of the DOM parser to construct an entire SOAP/XML message to interact with the XMethods Delayed Stock Quote Service. The gSOAP header file imports the DOM definitions and declares a `xsd__float` type which we will use to extract float values from a SOAP/XML response:

```
#import "dom++.h"
typedef float xsd__float;
```

The code is:

```
#include "soapH.h"
#include <iostream.h>
int main()
{
  struct soap *soap = soap_new();
  soap_set_imode(soap, SOAP_DOM_NODE);
  soap_dom_element envelope(soap, "http://schemas.xmlsoap.org/soap/envelope/", "Envelope");
  soap_dom_element body(soap, "http://schemas.xmlsoap.org/soap/envelope/", "Body");
  soap_dom_attribute encodingStyle(soap, NULL, "encodingStyle", "http://schemas.xmlsoap.org/soap/encoding/");
  soap_dom_element request(soap, "urn:xmethods-delayed-quotes", "getQuote");
  soap_dom_element symbol(soap, NULL, "symbol", "IBM");
  soap_dom_element response(soap);
  envelope.add(body);
  body.add(encodingStyle);
  body.add(request);
  request.add(symbol);
  if (soap_connect(soap, "http://services.xmethods.net/soap", "") // = SOAP_OK when success-
ful
      || soap_put_xsd__anyType(soap, &envelope, NULL, NULL) // = SOAP_OK when successful
      || soap_end_send(soap) // = SOAP_OK when successful
      || soap_begin_recv(soap) // = SOAP_OK when successful
      || !soap_get_xsd__anyType(soap, &response, NULL, NULL) // = NULL when not successful
      || soap_end_recv(soap) // = SOAP_OK when successful
      || soap_closesock(soap)) // = SOAP_OK when successful
    soap_print_fault(soap, stderr);
  else
```

```
      cout << response << endl;
   for (soap_dom_iterator walker = response.find(SOAP_TYPE_xsd__float); walker != response.end();
++walker)
      cout << "Quote = " << *(xsd__float*)(*walker).node << endl;
   soap_destroy(soap);
   soap_end(soap);
   soap_done(soap);
   free(soap);
   return 0;
}
struct Namespace namespaces[] =
{
   {"SOAP-ENV", "http://schemas.xmlsoap.org/soap/envelope/"},
   {"SOAP-ENC", "http://schemas.xmlsoap.org/soap/encoding/"},
   {"xsi", "http://www.w3.org/2001/XMLSchema-instance"},
   {"xsd", "http://www.w3.org/2001/XMLSchema"},
   {NULL, NULL}
};
```

Compile the code and link it with dom++.cpp, stdsoap2.cpp, and soapC.cpp. The program prints the SOAP/XML response and a floating point value.

5 Controlling the DOM Parser with Flags

The following input-mode flags (soap_imode() flags) can be used to control the DOM parser:

Flag	Description
SOAP_DOM_TREE	parse XML as a DOM only w/o embedding application data nodes
SOAP_DOM_NODE	parse XML as a DOM and attempt to embed application data nodes when possible
SOAP_C_UTFSTRING	parse XML cdata in UTF-8 form into the xsd__anyType.data string field

The flags are disabled by default. This means that application data nodes are embedded in the DOM only when necessary (this requirement is applicable to SOAP/XML messages that combine the gSOAP SOAP/XML with the DOM) and cdata is parsed into the xsd__anyType.wide field.

The following output-mode flags (soap_omode() flags) can be used to control the DOM parser output:

Flag	Description
SOAP_XML_CANONICAL	produce canonical XML
SOAP_ENC_ZLIB	compress output with Zlib (see gSOAP documentation)
SOAP_C_UTFSTRING	input/output data string field encoded in UTF8

6 DOM Class

The DOM class definition (current beta form). Note: xsd__anyType is an alias of soap_dom_element. It defines the serializers of the DOM in gSOAP.

```
struct soap_dom_element
{
   soap_dom_element *next; // next element (sibling) in sequence (not used at the document root)
   soap_dom_element *prnt; // parent node
   soap_dom_element *elts; // optional element children (data, wide, node must be NULL)
   soap_dom_attribute *atts; // optional element attributes
   const char *nstr; // optional namespace name string (URI)
   char *name; // element name with optional prefix
   char *data; // optional element CDATA value
   wchar_t *wide; // optional element CDATA value (wide char string)
   int type; // optional type of data pointed to (SOAP_TYPE_X)
   void *node; // and the optional pointer to serializable data node
   struct soap *soap; // gSOAP soap struct that manages this node
   soap_dom_element();
   soap_dom_element(struct soap *soap);
   soap_dom_element(struct soap *soap, const char *nstr, const char *name);
   soap_dom_element(struct soap *soap, const char *nstr, const char *name, const char*data);
   soap_dom_element(struct soap *soap, const char *nstr, const char *name, void *node, int type);
   ~soap_dom_element();
   soap_dom_element &set(const char *nstr, const char *name);
   soap_dom_element &set(const char *data);
   soap_dom_element &set(void *node, int type);
   soap_dom_element &add(soap_dom_element *elt);
   soap_dom_element &add(soap_dom_element &elt);
   soap_dom_element &add(soap_dom_attribute *att);
   soap_dom_element &add(soap_dom_attribute &att);
   soap_dom_iterator begin();
   soap_dom_iterator end();
   soap_dom_iterator find(const char *nstr, const char *name);
   soap_dom_iterator find(int type);
   void unlink();
};
struct soap_dom_attribute
{
   soap_dom_attribute *next; // next attribute in sequence
   const char *nstr; // optional attribute namespace name string (URI)
   char *name; // attribute name
   char *data; // optional attribute CDATA value
   wchar_t *wide; // optional attribute CDATA value (not used)
   struct soap *soap; // gSOAP soap struct that manages this instance
   soap_dom_attribute();
   soap_dom_attribute(struct soap *soap);
   soap_dom_attribute(struct soap *soap, const char *nstr, const char *name, const char *data);
   ~soap_dom_attribute();
   soap_dom_attribute &set(const char *nstr, const char *name);
```

```
  soap_dom_attribute &set(const char *data);
  void unlink();
};
class soap_dom_iterator
{ public:
  soap_dom_element *elt;
  const char *nstr;
  const char *name;
  int type;
  soap_dom_iterator();
  soap_dom_iterator(soap_dom_element *elt);
  ~soap_dom_iterator();
  bool operator==(const soap_dom_iterator &iter) const;
  bool operator!=(const soap_dom_iterator &iter) const;
  soap_dom_element &operator*() const;
  soap_dom_iterator &operator++();
};
```

6.1 DOM Constructors

Note: xsd__anyType is an alias of soap_dom_element.

soap_dom_element **constructors**
soap_dom_element(**struct** soap *soap) Creates a document DOM node and binds it to a gSOAP environment for memory management and I/O
soap_dom_element(**struct** soap *soap, **const char** *nstr, **const char** *name) Creates a document DOM node with namespace URI and element name
soap_dom_element(**struct** soap *soap, **const char** *nstr, **const char** *name, **const char** *data) Creates a document DOM node with namespace URI, element name, and element content
soap_dom_element(**struct** soap *soap, **const char** *nstr, **const char** *name, **void** *node, **int** type) Creates a document DOM node containing an application-specific data structure

soap_dom_attribute **constructors**
soap_dom_attribute(**struct** soap *soap) Creates a DOM node attribute and binds it to a gSOAP environment for memory management and I/O
soap_dom_attribute(**struct** soap *soap, **const char** *nstr, **const char** *name, **const char** *data) Creates a DOM node attribute with namespace URI, name, and content

soap_dom_element **methods**
soap_dom_element &set(**const char** *nstr, **const char** *name)
Set the namespace and name of a DOM node
soap_dom_element &set(**const char** *data)
Set the DOM node element content
soap_dom_element &set(**void** *node, **int** type)
Set the DOM node to contain an application-specific data structure
soap_dom_element &add(soap_dom_element *elt)
Add a new sub element (child) to the DOM node
soap_dom_element &add(soap_dom_element &elt)
Add a new sub element (child) to the DOM node
soap_dom_element &add(soap_dom_attribute *att)
Add a new element attribute to the DOM node
soap_dom_element &add(soap_dom_attribute &att)
Add a new element attribute to the DOM node
void unlink()
Unlink the DOM node, attributes, children, and content from the gSOAP deallocation chain

6.2 DOM Inspection

soap_dom_element **data members**
const char *nstr
Namespace name string (URI)
char *name
Element name with optional prefix
char *data
Element CDATA content value (DOM output only, set by parser only with SOAP_C_UTFSTRING flag set)
wchar_t *wide
Element CDATA content value (wide char string)
int type
The type (SOAP_TYPE_*x* identifier) of the data pointed to by node
void *node Pointer to serializable data type node
soap_dom_element *elts
Children of a node (data, wide, and node must be NULL)
soap_dom_attribute *atts
Element node attributes

Note: all data members are public. The class provides setter methods, but no getter methods. DOM node values can be inspected directly. This access policy may change depending on user feedback on this beta release with regard to getter methods versus direct access.

6.3 DOM Iterators

The find method returns a DOM iterator. The iterators of xsd__anyType are:

soap_dom_element **iterator methods**
soap_dom_iterator begin() Iterates over all DOM nodes
soap_dom_iterator end() Points past last node (empty iterator)
soap_dom_iterator find(**const char** *nstr, **const char** *name) Iterates over nodes with namespacec name (URI) and element tag name. The namespace and element tag names are patterns that may include the wilcards '*' (multi-character wildcard) and '-' (single-character wildcard).
soap_dom_iterator find(**int** type) Iterates over nodes with type (a SOAP_TYPE_*x* type identifier)

All iterators are forward iterators that traverse the tree in order.

The nstr and name parameters of the find method specify the namespace name (URI) and XML element tag name of the DOM nodes, respectively. These parameters MAY contain wildcards: an asterisk denotes a multi-character wildcard and a dash denotes a single character wildcard. For example, document.find("*", "product") iterates over all `<product>` nodes in any namespace.

A type parameter is a type identifier SOAP_TYPE_*x*, where *x* is the name of the type. The type MUST have a definition in a gSOAP header file, so the gSOAP compiler can generate its (de)serializers and the type identifier. The type *x* is the name of a type defined with a **typedef**, **struct**, **class**, or **enum**, or it is the name of a primitive type such as int.

7 XML Parsing and Streams

A DOM can be parsed from a stream using the >> stream operator. A DOM can be written to a stream using the << stream operator. It is important that the DOM is bound to a gSOAP environment (soap struct) to handle the memory management and I/O operations. The binding also enables the use of various gSOAP settings to control XML parsing and generation such as compression.

8 Planned Features

The DOM parser is a beta release. Future additions will include:

- Support for DOM level-2 (NodeWalker, NodeIterator, and NodeFilter) [http://www.w3.org/TR/DOM-Level-2-Traversal-Range/traversal.html#Iterator-overview]
- XPath access (?)
- Handle mixed element content differently (?) Currently stores mixed content in string form.
- ...