A Framework for Service-Oriented Computing with C and C++ Web Service Components

ROBERT A. VAN ENGELEN
Florida State University

Service-oriented architectures use loosely coupled software services to support the requirements of business processes and software users. Several software engineering challenges have to be overcome to expose legacy C and C++ applications and specialized system resources as XML-based software services. It is critical to devise effective bindings between XML and C/C++ data to efficiently interoperate with other XML-based services. Binding application data to XML has many software solutions, ranging from generic document object models to idiosyncratic type mappings. However, a safe binding must conform to XML validation constraints, guarantee type safety, and should preserve the structural integrity of communicated application data. By contrast, tight XML bindings impose mapping constraints that can hamper interoperability between services. This paper presents a framework for constructing loosely coupled C/C++ services based on a programming model that integrates XML bindings into the C and C++ syntax. The concepts behind the bindings are generic, which makes the approach applicable to other programming languages.

Categories and Subject Descriptors: C.2.4 [**Computer-Communication Networks**]: Distributed Systems; D.2.6 [**Software Engineering**]: Programming Environments; D.2.11 [**Software Engineering**]: Software Architectures; D.2.12 [**Software Engineering**]: Interoperability; D.3.4 [**Programming Languages**]: Processors

General Terms: Algorithms, Performance

Additional Key Words and Phrases: Service-oriented computing, Web services standards.

1. INTRODUCTION

A service-oriented architecture (SOA) is a collection of network-accessible, loosely-coupled services that are assembled into composite applications to support the requirements of software users. Services in a SOA are discovered, described, and invoked via XML messages [Erl 2005], typically using UDDI [UDDI Organization 2005] for discovery, WSDL [W3 Consortium 2001] for service interface bindings, and SOAP [W3 Consortium 2000] or REST [Fielding 2000] for message transport.

The integration of legacy C and C++ applications as services in a SOA, and the

This work was partially is supported by the US Department of Energy Early Career Principal Investigator award under contract number DEFG02-02ER25543.
Authors's address: R.A. van Engelen, Department of Computer Science, 253 James J. Love building, Florida State University, Tallahassee, FL 32306-4530.

integration of system resources that are primarily accessible via C or C++ application programming interfaces (APIs), presents several software engineering challenges. Firstly, there is no one-to-one mapping between data types of common programming languages, such as C and C++, and XSD types (the types defined by XML schema root components). Hence, a mapping mechanism must be devised to safely and consistently bind programming-language specific application data to XML, see e.g. [Hericko et al. 2003; Loughran and Smith 2005; Meijer et al. 2003; Thomas 2003; van Engelen et al. 2006]. Secondly, loose couplings of services require significant flexibility of XML bindings, since a tight binding imposes mapping constraints that can hamper interoperability. Thus, it is critical to use effective bindings between XML and C/C++ data to safely and transparently exchange data with other XML-based services. Thirdly, efficiency is paramount to speed up service response times with low-latency message processing methods. XML processing often incurs significant overhead in time and space for composing and decomposing XML into application data. A mapping that is based on early binding times, e.g. using source-to-source compiler techniques, mitigates the efficiency loss of XML messaging through compiler-generated XML processing routines.

This paper presents a framework built on the open source gSOAP toolkit [van Engelen 2001]. The framework promotes ease-of-use and offers tools to accelerate the deployment of existing and new C/C++ applications as high-performance interoperable services. The ease-of-use and performance goals are achieved with source-to-source compiler-based techniques, which effectively implement XML bindings directly into the C and C++ syntax. As a result, remote service operations have the simplicity and look-and-feel of local function invocation.

The development and deployment stages of a C/C++ Web service are shown in Figure 1. The stages are broken down into three layers: the *service definition layer*, the *application integration layer*, and the *transport layer*. The gSOAP tools target each layer with the *wsdl2h* parser, the *soapcpp2* stub/skeleton compiler, and the *stdsoap2* run-time engine, respectively.

Consider the following example SOA deployment scenario. Suppose a user is given a collection of WSDLs to implement services and/or service consumers in C or C++. She processes the WSDLs with *wsdl2h* **(A)** to produce the binding code **(B)** for the application integration layer. She then invokes the *soapcpp2* compiler to generate the front-end transport-level bindings and stubs and skeletons **(C)** for the *client proxy* or *service dispatcher*. The proxy and dispatcher use the *stdsoap2* engine **(D)** for SOAP messaging over HTTP, using SOAP1.1/1.2 RPC or document/literal formats. The back-end application **(E)** handles the SOAP/XML request and response messages via object method invocations (for C++ code) or function calls (for C code) through the RPC interface. The proxy is an object with member functions to invoke remote service operations, where C/C++ type arguments are marshaled in XML using efficient auto-generated serialization routines. Service objects can be deployed in Web server containers or as stand-alone daemons.

The tools also enable the user to directly deploy existing C/C++ code, such as legacy applications, as services by invoking the *soapcpp2* compiler on a header file **(B)** containing the back-end service API definitions. This exposes the application as a service via the generated binding code **(C)**. In addition, WSDLs and schemas

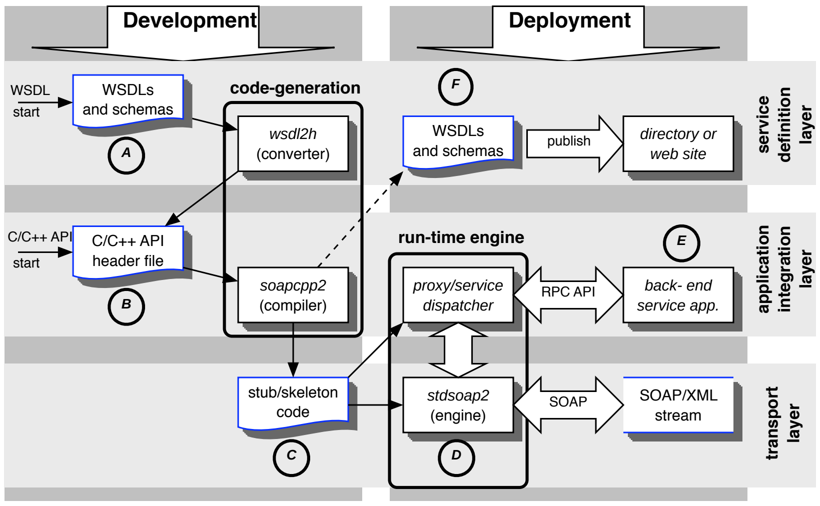


Fig. 1. Service development and deployment stages with gSOAP.

(F) are generated and those can be published to create service consumers.

The remainder of this paper is organized as follows. Section 2 compares the framework to related work. Section 3 gives an architectural overview of the framework. The XML binding concepts for C/C++ are introduced in Section 4, and Section 5 discusses the compiler implementation of the type mappings and auto-generated XML serialization routines. An evaluation of the performance and memory requirements is given in Section 6, using a number of benchmark service applications. Finally, the framework implementation and results are summarized in Section 7.

2. RELATED WORK

The early generation SOAP toolkits were limited to SOAP RPC encoding, an XML format for common data types in programming languages, such as primitive types, structs, and arrays. Example open-source toolkits that fall in this category are Apache Axis versions 1.x [Apache Foundation 2002] for Java and C++, SOAP Lite [Kulchenko 2003] for Perl, and NuSOAP [Ayala 2002] for PHP. Most of these systems provide a translation mechanism of WSDL service descriptions to program source code using auto-generation techniques. The early toolkits provided limited or no support for the translation of source code to WSDL.

The latest generation of toolkits support XML more or less natively, which is required to enable SOAP document/literal messaging with loosely coupled services. Newer generation toolkits also support the emerging set of WS-* protocols. Examples of open-source projects are Apache Axis 2.x [Apache Foundation 2002], gSOAP [van Engelen 2001], Mono [de Icaza 2004], and commercial products such as Borland JBuilder, IBM WebSphere, Rogue Wave LEIF, Systinet WASP, and Microsoft .NET.

The recently released *Service Component Architecture* (SCA) [Open SOA Collaboration 2006] specification for C and C++ is most closely related to the work presented in this paper. SCA proposes an XML-to-programming language binding by utilizing annotations and syntax extensions to support XML Web services natively. In general, SCA promotes the standardization of programming models for building SOA applications and systems. The specifications standardize a rich set of annotations for various programming and script languages. However, the specifications are at a very early stage of development and no open-source implementations are available yet.

Efforts to increase the performance of XML processing include differential XML serialization [Abu-Ghazaleh et al. 2004] and deserialization [Abu-Ghazaleh and Lewis 2005], encoding scientific data in SOAP/XML [van Engelen 2003], schema-specific parsing [Chiu 2003; van Engelen 2004; Zhang and van Engelen 2006]. Other techniques are more generic and attempt to merge parsing, validation, and deserialization stages [Kostoulas et al. 2006], which was effectively implemented in early versions of the gSOAP toolkit [van Engelen and Gallivan 2002].

3. FRAMEWORK OVERVIEW

A schematic overview of the gSOAP framework is shown in Figure 2. The *wsdl2h* tool (top-left shaded box) converts a set of WSDLs and XML schemas into a service specification represented in an annotated C/C++ header file. This header file has an IDL-like format by declaring service operations and the associated data types in familiar C or C++ syntax, which simplifies the interface to a back-end application.

The type annotations, class hierarchies, service operations, and documentation is visualized with Doxygen [van Heesch 1997]. The visual rendition includes documentation copied from the original WSDLs and schemas, service operations, parameters, and data types. It also includes instructions on how to invoke the operations via C stubs and C++ proxy objects. The service operation parameters are standard C/C++ types extracted from the XML schema types. These are automatically

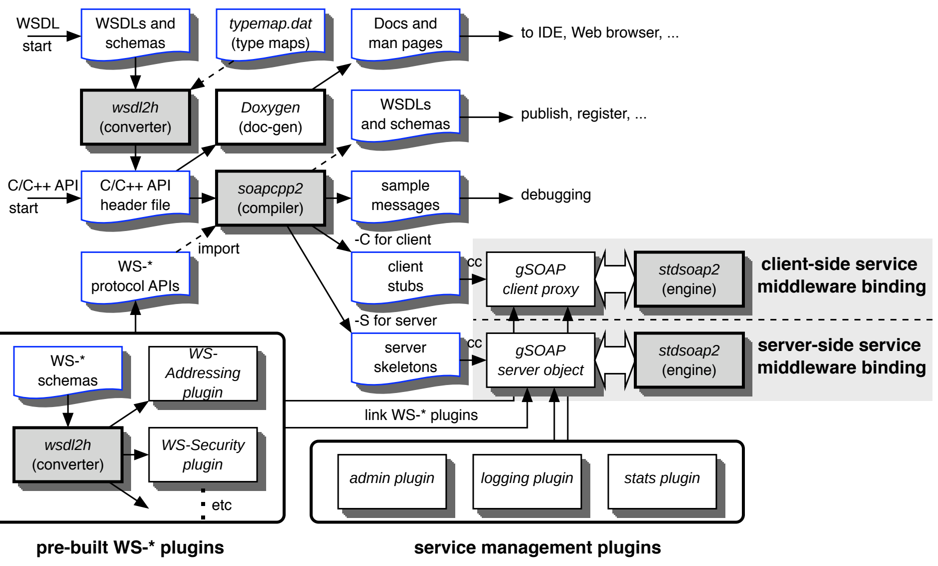


Fig. 2. Overview of the gSOAP framework.

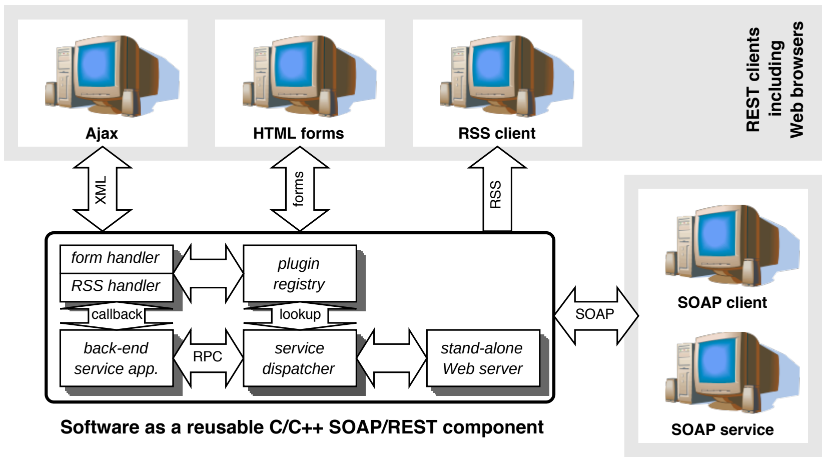


Fig. 3. Exposing several SOAP/REST ports to reuse software as a service.

mapped to C/C++ types. The mapping can be customized with alternate XSD type bindings specified in the *typemap.dat* file. Details are discussed in Section 4.

At the center of the framework is a stub/skeleton compiler *soapcpp2* shown in Figure 2 (center shaded box) to generate the actual binding code from the C/C++ header file specification. The compiler generates stub and proxy code for clients and skeleton code for servers. When developing a service interface for an existing application, this is the starting point where the tool generates WSDL definitions and binding code directly from a C/C++ specification of the back-end API functions.

The *wsdl2h* tool also serves as a WS-* protocol library builder in the framework (bottom-left shaded box in Figure 2). The tool is used to construct XML processors for WS-* standards, such as WS-Security, WS-Addressing, and WS-Management. These meta-protocols are implemented in plugins using the *wsdl2h*-generated interfaces and XML processors, combined with the protocol logic. To use these plugins in applications, the protocol definitions are passed through the *soapcpp2* compiler via #import statements in the header file of the service application, which was also automatically generated by *wsdl2h* from a WSDL that specifies the use of WS-* protocols. This is essential to bind the WS-* protocol structures to the service operations, e.g. to define SOAP Header elements for SOAP messaging with meta information in SOAP headers, e.g. WS-Addressing and WS-Security headers.

The *wsdl2h* tool is self-generating. More specifically, the WSDL 1.1 schema and the XML schema-of-schema definitions are put in a gSOAP header file with the help of *wsdl2h* to build the *wsdl2h* tool itself using auto-generation of the WSDL and XML schema parsers.

Plugins are registered with the *stdsoap2* engine, as shown in Figure 2 (right shaded boxes). The engine implements the HTTP(S) and SOAP/XML stacks. It also includes a run-time environment to manage network connections, allocate memory (Section 5.3), and to keep track of the HTTP/XML processing state. Plugins implement support for REST-based services and provide APIs for WS-* protocols such as WS-Addressing and WS-Security. The REST plugins enable application data to be made accessible with HTTP methods via callbacks to application-specific handlers, as shown in Figure 3. The SOAP/XML operations are dispatched independently via the auto-generated dispatcher, while plugins handle the REST methods.

4. THE PROGRAMMING MODEL

Binding application data to XML has many software solutions, ranging from generic document object models to idiosyncratic type mappings. The gSOAP framework implements a convenient RPC-based programming model for service operations using integrated XML bindings in the C/C++ syntax. The syntax extensions, notational conventions, and code annotations for service operations and XML parameter types are described first, followed by a presentation of the mapping of WSDL and XSD types to C or C++.

4.1 C/C++ Language Extensions, Notational Conventions, and Annotations

To expose an existing C/C++ application as the back-end of a service, a standard C/C++ header file with the application's API suffices. SOAP service operations are expressed as function prototypes of the API and native C/C++ parameter types can be given. The functions and parameters are mapped to SOAP/XML by the *soapcpp2* compiler. The compiler also generates a WSDL description of the front-end service API, the stubs and skeletons for the SOAP operations, and XML serialization code for the C/C++ types.

Annotations with "*//gsoap* directives" are used to customize the stub and skeleton code output and to complete the WSDL bindings. The annotations include definitions for:

Service binding names. Used to generate a named WSDL and C++ proxy class.

Service port. Defines the endpoint address of the service.

SOAP operation styles and encodings. RPC-encoding and document/literal style.

Schema namespace bindings. Associates namespace prefixes with URIs.

Schema element and attribute form defaults. Qualified or unqualified.

An annotation essentially associates information with a particular Web service namespace. A namespace is identified by a simple, user-definable token, such as ns1, ns2, xyz, or mysrv. These namespace tokens are used to associate the service operations and data types with a service namespace. As a notational convention, C/C++ names for functions, data types, and class member names are prefixed using the format *ns__name* to bind *name* to a namespace identified by *ns*.

Because XML namespaces [W3 Consortium 2006] have no one-to-one mapping to C++ namespaces, the prefix notation is an effective mechanism for namespace resolution in C and C++. It also avoids name clashes between service operations, data types, and data members. In general, programming language namespaces and packages are not compatible with XML schema namespaces [Loughran and Smith 2005; van Engelen et al. 2006], and other naming conventions must be used.

Consider for example the C header definitions shown in Figure 4, which declares the delayed stock quote service *getQuote* operation from XMethods [XMethods 2004], an open repository of Web services. The annotations are automatically generated by *wsdl2h* from the WSDL of the XMethods service. In this example, the annotations refer to the ns namespace prefix of the service, where the service

```
//gsoap ns service name:       quote
//gsoap ns service port:       http://services.xmethods.net/soap
//gsoap ns service style:      rpc
//gsoap ns service encoding:   encoded
//gsoap ns schema namespace:   urn:xmethods-delayed-quotes
int ns__getQuote(char *symbol, float *Result);
```

Fig. 4. Annotated service operation of the XMethods' delayed stock quote service in C.

```
int main()
{ return soap_serve(soap_new()); } // dispatch SOAP request to ns__getQuote

int ns__getQuote(struct soap *ctx, char *symbol, float *Result)
{ *Result = get_delayed_stock_quote(symbol); // get the quote from a database
  return SOAP_OK;
}
```

Fig. 5. The XMethods' stock quote service in C.

operation *getQuote* is bound to the namespace-qualified ns__getQuote. An RPC stub routine for the service API function is generated by *soapcpp2*. XML marshaling and serialization routines are generated for the parameters, where the XML messages are SOAP1.1 RPC-encoding compliant.

All but the last parameter of a service API function are in-mode arguments. The last parameter is an out-mode parameter, declared as pointer type in C or reference type in C++. The general format of a service API function is:

$$\text{int } ns\_\_funcName(inArg_1,\ inArg_2,\ \ldots,\ inArg_n,\ \ldots,\ outArg);$$

Multiple out-mode parameters are collected in a "response" struct, where the data members are the named arguments. The reason behind this design choice is that the SOAP standard does not require a unique type for a named in-out mode parameter, thus making it more difficult to use in-out parameters.

The stub function returns an integer status code for success or failure of the invocation. When building a service interface for an existing back-end application, some changes in the argument signatures may be required. A simple wrapper function can be used instead. For example, a CGI-based implementation of the service is shown in Figure 5. This example also illustrates the use of the gSOAP engine context struct soap, allocated with soap_new(). The thread-local context is used for service state management, see Section 5.3.

To assist the automatic marshaling of parameter data in XML, data types are annotated as follows:

—The @ qualifier defines XML attribute serialization of a class/struct data member.

—Occurrence constraints on class/struct data members are defined with $min : max$, e.g. optional (0:1), mandatory (1:1), or prohibited (0:0). Occurrence constraints are enforced by the XML deserializer of the class or struct.

—Class and struct data members can be assigned default values, which are used in XML deserialization of missing data (e.g. optional XML attributes and elements).

—Types declared as volatile are considered non-mutable and are not redefined by the *soapcpp2* code output. This allows the serialization of specialized data types,

```
//gsoap s service name:         prober
//gsoap s schema namespace: urn:sensor
class s__readout
{ public:
    @enum status {ON,OFF} state 1:1;
    double value = 0.0             0:1;
};
int s__probe(char *sens, s__readout& r);
```

```
<SOAP-ENV:Envelope...>
 <SOAP-ENV:Body>
  <s:probe
     xmlns:s="urn:sensor">
   <sens>temp-3</sens>
  </s:probe>
 </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

```
<SOAP-ENV:Envelope...>
 <SOAP-ENV:Body>
  <s:readout
     xmlns:s="urn:sensor"
     state="ON">
   <value>89.4</value>
  </s:readout>
 </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Fig. 6. Example annotated service definition and SOAP request-response messages.

e.g. types used by C/C++ libraries, since these are externally declared.

The example shown in Figure 6 illustrates these annotations. The s__probe operation acquires the status and sensor readout when the sensor is ON. The state value is serialized as an XML attribute (@) that is mandatory (1:1), while the value element may be absent (0:1) in the XML content, upon which 0.0 is assigned.

4.2 Mapping XSD Types to C/C++ Types

The annotations and C/C++ syntax conventions form the basis of a generic XML serialization algorithm, discussed in Section 5. This section describes how the C/C++ annotations and types are generated by *wsdl2h* from WSDLs and XML schemas.

The *wsdl2h* tool maps the XSD type system to the C/C++ type system. Mapping XSD type constructs to programming language types is a challenge, sometimes referred to as the "X/O impedance mismatch" [Loughran and Smith 2005; van Engelen et al. 2006], i.e. stating the incompatibility between XML (parented trees) and rich objects (unparented object graphs).

Because SOAP document/literal style messages encoding is a significant departure from RPC encoding, mappings must be defined for XML schema components rather than the limited repertoire of SOAP-encoded types. The translation rules used by *wsdl2h* for the most common XML schema components are listed in Figure 7, using a notation borrowed from denotational semantics. Mappings are defined on syntactic structures, i.e. XML schema structures, denoted within $[\![\ ]\!]$ brackets. The $D[\![X]\!]\,ns\,n$ rules take a XML schema component X, the schema target namespace ns, a component name n, and returns the C++ type representation of X. The local $L[\![X]\!]\,ns$ rules take a local component X, the schema target namespace ns and returns the C++ type representation of X. The *wsdl2h* tool also supports any, group, attributeGroup, element substitutions, simpleType/list (bitmasks). Occurrence constraints minOccurs, maxOccurs, minLength, maxLength are translated into annotations.

By applying the rules recursively, the XML schema components are mapped to (nested) C/C++ types. Figure 8 shows an example translation of a schema element definition into a C++ class. These type declarations are analyzed by *soapcpp2* to generate XML serializers that are consistent with the XML schema definitions and constraints. Figure 9 shows an XML instance deserialized into a corresponding C++ data structure. The data structure is isomorphic with the XML instance type shown in Figure 8.

Top-level element and atribute translations

$D\llbracket\langle\textsf{element name='}n\textsf{'}\rangle X\langle/\textsf{element}\rangle\rrbracket\ ns\ \_ \quad = D\llbracket X\rrbracket\ ns\ E(ns:n)$

$D\llbracket\langle\textsf{element name='}n\textsf{' type='}t\textsf{'/}\rangle\rrbracket\ ns\ \_ \quad = \textbf{typedef}\ T(t)\ E(ns:n)$

$D\llbracket\langle\textsf{attribute name='}n\textsf{'}\rangle X\langle/\textsf{attribute}\rangle\rrbracket\ ns\ \_ \quad = D\llbracket X\rrbracket\ ns\ E(ns:n)$

$D\llbracket\langle\textsf{attribute name='}n\textsf{' type='}t\textsf{'/}\rangle\rrbracket\ ns\ \_ \quad = \textbf{typedef}\ T(t)\ E(ns:n)$

Top-level simpleType translations

$D\llbracket\langle\textsf{simpleType name='}n\textsf{'}\rangle X\langle/\textsf{simpleType}\rangle\rrbracket\ ns\ \_ \quad = D\llbracket\langle\textsf{simpleType}\rangle X\langle/\textsf{simpleType}\rangle\rrbracket\ ns\ n$

$D\llbracket\langle\textsf{simpleType}\rangle\langle\textsf{restriction base='}t\textsf{'/}\rangle\langle/\textsf{simpleType}\rangle\rrbracket\ ns\ n = \textbf{typedef}\ T(t)\ T(ns:n)$

$D\llbracket\langle\textsf{simpleType}\rangle\langle\textsf{restriction base='}t\textsf{'}\rangle\langle\textsf{enumeration value='}v_1\textsf{'/}\rangle\ldots\langle/\textsf{restriction}\rangle\langle/\textsf{simpleType}\rangle\rrbracket\ ns\ n$
$\quad = \textbf{enum}\ T(ns:n)\ \{\ S(v_1),\ \ldots,\ S(v_n)\ \}$

Top-level complexType translations

$D\llbracket\langle\textsf{complexType name='}n\textsf{'}\rangle X_1\ldots X_n\langle/\textsf{complexType}\rangle\rrbracket\ ns\ \_$
$\quad = D\llbracket\langle\textsf{complexType}\rangle X_1\ldots X_n\langle/\textsf{complexType}\rangle\rrbracket\ ns\ n$

$D\llbracket\langle\textsf{complexType}\rangle\langle\textsf{simpleContent}\rangle\langle\textsf{extension base='}t\textsf{'}\rangle X_1\ldots X_n\langle\textsf{extension}\rangle\langle/\textsf{simpleContent}\rangle$
$\quad \langle/\textsf{complexType}\rangle\rrbracket\ ns\ n$
$\quad = \textbf{class}\ T(ns:n)\ \{\ T(t)\ \_\_\textsf{item};\ (L\llbracket X_1\rrbracket\ ns)\ \ldots\ (L\llbracket X_n\rrbracket\ ns)\ \}$

$D\llbracket\langle\textsf{complexType}\rangle\langle\textsf{complexContent}\rangle\langle\textsf{restriction base='}t\textsf{'}\rangle X_1\ldots X_n\langle/\textsf{restriction}\rangle\langle/\textsf{complexContent}\rangle$
$\quad \langle/\textsf{complexType}\rangle\rrbracket\ ns\ n$
$\quad = \textbf{class}\ T(ns:n)\ \{\ (L\llbracket X_1\rrbracket\ ns)\ \ldots\ (L\llbracket X_n\rrbracket\ ns)\ \}$

$D\llbracket\langle\textsf{complexType}\rangle\langle\textsf{complexContent}\rangle\langle\textsf{extension base='}t\textsf{'}\rangle X_1\ldots X_n\langle\textsf{extension}\rangle\langle/\textsf{complexContent}\rangle$
$\quad \langle/\textsf{complexType}\rangle\rrbracket\ ns\ n$
$\quad = \textbf{class}\ T(ns:n)\ :\ \textbf{public}\ T(t)\ \{\ (L\llbracket X_1\rrbracket\ ns)\ \ldots\ (L\llbracket X_n\rrbracket\ ns)\ \}$

$D\llbracket\langle\textsf{complexType}\rangle X_1\ldots X_n\langle/\textsf{complexType}\rangle\rrbracket\ ns\ n$
$\quad = \textbf{class}\ T(ns:n)\ \{\ (L\llbracket X_1\rrbracket\ ns)\ \ldots\ (L\llbracket X_n\rrbracket\ ns)\ \}$

Local component translations

$L\llbracket\langle\textsf{sequence}\rangle X_1\ldots X_n\langle/\textsf{sequence}\rangle\rrbracket\ ns \quad = (L\llbracket X_1\rrbracket\ ns)\ \ldots\ (L\llbracket X_n\rrbracket\ ns)$

$L\llbracket\langle\textsf{all}\rangle X_1\ldots X_n\langle/\textsf{all}\rangle\rrbracket\ ns \quad = (L\llbracket X_1\rrbracket\ ns)\ \ldots\ (L\llbracket X_n\rrbracket\ ns)$

$L\llbracket\langle\textsf{choice}\rangle X_1\ldots X_n\langle/\textsf{choice}\rangle\rrbracket\ ns$
$\quad = \textbf{int}\ \_\_\textsf{union};\ \textbf{union}\ \{\ (L\llbracket X_1\rrbracket\ ns)\ \ldots\ (L\llbracket X_n\rrbracket\ ns)\ \}\ \textsf{choice};$

$L\llbracket\langle \mathit{component}\ \textsf{maxOccurs='unbounded'}\rangle X_1\ldots X_n\langle/\mathit{component}\rangle\rrbracket\ ns$
$\quad = \textbf{int}\ \_\_\textsf{size};\ \textbf{struct}\ \{\ L\llbracket\langle\mathit{component}\rangle X_1\ldots X_n\langle/\mathit{component}\rangle\rrbracket\ \}\ *\_\_\textsf{array}$

Local element and attribute translations

$L\llbracket\langle\textsf{element name='}n\textsf{'}\rangle X\langle/\textsf{element}\rangle\rrbracket\ ns \quad = (D\llbracket X\rrbracket\ ns\ E(ns:n))\ M(ns:n);$

$L\llbracket\langle\textsf{element name='}n\textsf{' type='}t\textsf{'/}\rangle\rrbracket\ ns \quad = T(t)\ M(n);$

$L\llbracket\langle\textsf{element ref='}n\textsf{'/}\rangle\rrbracket\ ns \quad = E(n)\ M(ns:n);$

$L\llbracket\langle\textsf{element name='}n\textsf{' type='}t\textsf{' maxOccurs='unbounded'/}\rangle\rrbracket\ ns$
$\quad = \textbf{std::vector}{<}T(t){>}\ M(n);$

$L\llbracket\langle\textsf{attribute name='}n\textsf{'}\rangle X\langle/\textsf{attribute}\rangle\rrbracket\ ns \quad = \texttt{@}\ (D\llbracket X\rrbracket\ ns\ E(ns:n))\ M(ns:n);$

$L\llbracket\langle\textsf{attribute name='}n\textsf{' type='}t\textsf{'/}\rangle\rrbracket\ ns \quad = \texttt{@}\ T(t)\ M(n);$

$L\llbracket\langle\textsf{attribute ref='}n\textsf{'/}\rangle\rrbracket\ ns \quad = \texttt{@}\ E(n)\ M(ns:n);$

where	$T(ns:n)$	returns a C type name of the form "$ns\_\_n$" for the QName $ns:n$
	$E(ns:n)$	returns "$\_ + T(ns:n)$" (type name with leading underscore)
	$M(ns:m:n)$	returns class member name "n" if $ns = m$
	$M(ns:m:n)$	returns a qualified class member name "$m\_\_n$" if $ns \neq m$
	$S(v)$	returns a unique new enum symbol for v

Fig. 7. XML schema component translations to C++.

```
⟨element name="X"⟩
 ⟨complexType⟩
  ⟨sequence⟩
   ⟨element name="Y"⟩
    ⟨complexType⟩
     ⟨choice maxOccurs="unbounded"⟩
      ⟨element name="a" type="int"/⟩
      ⟨element name="b" type="boolean"/⟩
      ⟨element name="c" type="float"/⟩
     ⟨/choice⟩
    ⟨/complexType⟩
   ⟨/element⟩
   ⟨element name="Z" type="string"
     maxOccurs="unbounded"/⟩
  ⟨/sequence⟩
  ⟨attribute name="A"⟩
   ⟨simpleType⟩
    ⟨restriction base="string"⟩
     ⟨enumeration value="ON"/⟩
     ⟨enumeration value="OFF"/⟩
    ⟨/restriction⟩
   ⟨/simpleType⟩
  ⟨/attribute⟩
 ⟨/complexType⟩
⟨/element⟩
```

```
class _ns__X
{
  public:
   class _ns__Y
   {
     public:
      int __size;
      struct S
      {
        int _union;
        union U
        {
          int a;
          bool b;
          float c;
        } choice;
      } *__array
   } Y;
   std::vector<std::string> Z;
   @enum _ns__A {ON,OFF} A;
};
```

Fig. 8. Example translation of an XML schema type to a C++ type.

```
⟨X A="ON"⟩
 ⟨Y⟩
  ⟨c⟩3.1⟨/c⟩
  ⟨a⟩5⟨/a⟩
  ⟨b⟩true⟨/b⟩
 ⟨/Y⟩
 ⟨Z⟩abc⟨/Z⟩
 ⟨Z⟩def⟨/Z⟩
⟨/X⟩
```

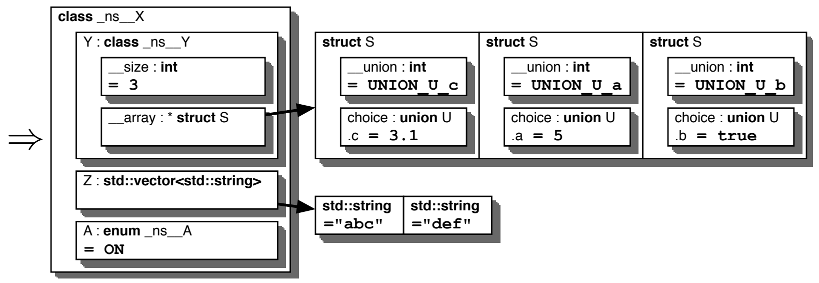


Fig. 9. An XML instance deserialized into a C++ data structure.

The mapping of primitive XSD types and simpleType restrictions to C/C++ is accomplished with typedef, which *soapcpp2* uses to binds a XSD type to a C/C++ type. For example, the xsd:NMTOKEN XSD type can be bound to a C string as follows:

typedef char ∗xsd__NMTOKEN;

Specialized mappings can be (re)defined in the *typemap.dat* file (see also Figure 2), using the following format for each mapping definition:

ns__type = *target-type-declaration* | *target-type* | *target-ptr-type*

where *target-type* is the target C/C++ type, with an optional pointer-based version. For example, the xsd:NMTOKEN type can be bound to a string as follows:

```
xsd__NMTOKEN = typedef char *xsd__NMTOKEN; | xsd__NMTOKEN | xsd__NMTOKEN
```

Other special cases are SOAP encoded arrays [W3 Consortium 2000], where SOAP arrays are type restrictions of the SOAP encoding schema array definition. The *wsdl2h* tool maps these to a special struct with size and type information, which *soapcpp2* serializes as a SOAP array. For example, a SOAP array of floats is defined by:

```
struct floatarray { float *__ptr; int __size; };
```

This declares an array of floats pointed to by __ptr with run-time size __size.

4.3 Mapping C/C++ Types to XSD Types

Translating C/C++ types into XSD types is less complex compared to the translation of XSD types into C/C++ types. The C/C++ types translation is close to SOAP RPC encoding requirements, i.e. there are only a limited number of XSD types needed to implement the binding. The framework supports all primitive C/C++ types, enumerations, structs, classes with single inheritance, STL strings and vectors, and pointer-based structures. Pointer-based structures such as lists, trees, DAGs, and even cyclic graphs are supported, but the serialization requires SOAP RPC encoding to ensure that the graph pointer structure is preserved. SOAP RPC encoding is a simple serialization format implemented with the following rules:

—Translate primitive types to built-in primitive XSD types [W3 Consortium 2004];

—Translate structs and classes to XML schema complexTypes and use a sequence of local elements for the data members;

—Translate arrays to SOAP arrays (SOAP 1.1 partial and sparse arrays);

—At run time, serialize object graphs using SOAP multi-reference encoding with id and href XML attributes;

—At run time, use dynamic binding to implement polymorphism for complexType extensions;

Table I lists the translation of primitive and compound C/C++ types to schema types.

The *soapcpp2* compiler generates serialization code and the WSDL and schema documents that define the XSD types of the mapping. XML namespaces of the schemas are used to organize related types. Both SOAP RPC encoding and the document/literal style (by default) are supported.

5. XML SERIALIZATION AND DESERIALIZATION

The XML serialization algorithms exploit type information gathered by *soapcpp2* to perform dynamic shape analysis to efficiently encode and decode native C/C++ data structures in XML. The design of the algorithms ensures object-level coherence [van Engelen et al. 2006]. Static analysis is needed, because reflection and type introspection are not generally available for C and C++.

5.1 Serializing C/C++ Data in XML

For each C/C++ type T, the *soapcpp2* compiler generates two functions to serialize data of type T:

Table I. Translation of C/C++ types to schema types for SOAP RPC encoding.

C/C++ Type	XML Schema Type (XSD Type)
bool	boolean
char	byte
short	short
int	int
long long	long
float	float
double	double
size_t	unsignedLong
time_t	dateTime
char∗	string
wchar_t∗	string
std::string	string
enum	simpleType/restriction/enumeration
typedef *T*	simpleType/extension or complexType/complexContent/extension
struct *T*	complexType/complexContent/extension
class *T*	complexType/complexContent/extension
T [*nnn*]	*SOAP-encoded array of T*
T ∗	*the schema type of T*

—soap_serialize_*T*(**struct** soap∗, **const** *T*∗). This function analyzes a structure of type *T* and its (public) data members recursively for co-referenced objects and to detect object graph cycles.

—soap_out_*T*(**struct** soap∗, **const char** ∗tag, **int** id, **const** *T*∗, **const char** ∗type). This function emits data of type *T* and its substructures, such as (public) data members, in XML based on the current encoding style (encoded or literal). The tag is a (qualified) XML element tag name, id is an optional reference identification for multi-ref encoding with id attributes (or zero), and type is the xsi:type string used for SOAP encoding style.

In case *T* is a class, the serialization (and deserialization) functions are virtual methods of that class to facilitate the serialization of polymorphic objects and containers via dynamic binding. When required, developers can also add custom serializers (and deserializers) for their data types by specifying the serialization (and deserialization) functions and methods for each custom type.

The serialization of a data structure proceeds in two phases:

(1) In the first phase the soap_serialize functions are called to analyze the data structure graph. When a data structure is composed of pointers to various substructures, this function to detect co-referenced structures and cycles. This is required to properly serialize data in the RPC multi-ref encoding style, where XML representations of pointer-based structures are to be linked with id and href attributes in XML.

(2) In the second phase, the soap_out functions emit the serialized XML stream based on the encoding style, i.e. with or without id, href, and xsi:type attributes.

To ensure object-level coherent XML serialization of object graphs, multi-ref encoding can be used to ensure that the relationships between the objects in a pointer graph are preserved [van Engelen et al. 2006]. This allows a receiver developed with gSOAP to reconstruct an exact copy of the data structure.

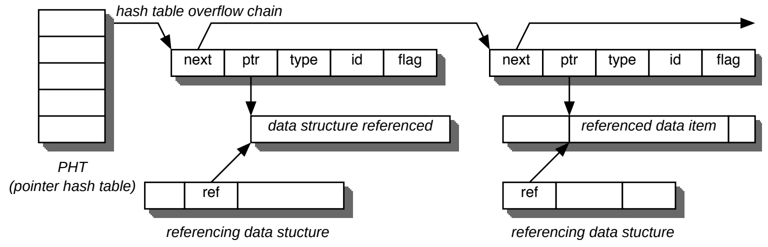

hash table overflow chain
next
ptr
type
id
flag
data structure referenced
referenced data item
ref
PHT
(pointer hash table)
referencing data stucture

Fig. 10. The PHT tracks pointer relationships within data structures for multi-ref analysis.

(a) code	(b) graph	(c) SOAP 1.1 RPC	(d) SOAP 1.2 RPC	(e) SOAP doc/lit

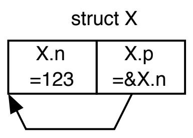
struct X
{
int n;
int ∗p;
};
		⟨X xsi:type="X"⟩
		⟨n href="#x"/⟩
		⟨p href="#x"/⟩
		⟨/X⟩
		⋮
		⟨x id="x"⟩123⟨/x⟩
			⟨X xsi:type="X"⟩
			⟨n id="x"⟩123⟨/n⟩
			⟨p ref="#x"/⟩
			⟨/X⟩
				⟨X⟩
				⟨n⟩123⟨/n⟩
				⟨p⟩123⟨/p⟩
				⟨/X⟩

Fig. 11. Example structure and SOAP/XML serializations when member p points to n.

To facilitate structure analysis, state information is maintained by the gSOAP engine in the PHT (pointer hash table) indexed with the address of an analyzed data item. The engine populates the hash table while recursively scanning for pointers to data items of type T using the soap_serialize_T functions. Figure 10 depicts the layout and state of the serializer for two pointer references. A PHT entry is created for each address value of pointers encountered in the recursive data structure analysis with soap_serialize. The PHT entries point to a chains of nodes, where each node contains the actual pointer address value ptr, the data type information (integer) on the object pointed to, an id value for multi-ref accessor generation, and a flag with value embedded or multiref to record whether the referenced data item was *embedded* within a larger structure, such as an array element or member of a struct or class.

Note that in Figure 10 the second pointer references a data item that is embedded within a data structure to be serialized, which is recorded with flag=embedded. Such data item is usually an array element or struct or class member that must be referenced *in situ*, i.e. should be identified with an id attribute to ensure isomorphic serialized forms.

Consider for example Figure 11(a). The structure contains two integers, n and p. Suppose p points to integer n as shown in Figure 11(b). The multi-ref SOAP 1.1 RPC encoded, SOAP 1.2 RPC encoded, and SOAP document style messages produced by the gSOAP serializer soap_out_X are shown in Figure 11(c), (d), and (e), respectively. In this case, the referencing data structure has a self-referenced data item which in SOAP 1.2 RPC encoding is clearly distinguished with the id attribute, based on flag=embedded.

To reduce run-time serialization cost, *soapcpp2* builds a “plausible data model” at compile time by constructing a graph of all object relationships that may exists within an application program. The model represents all possible configurations

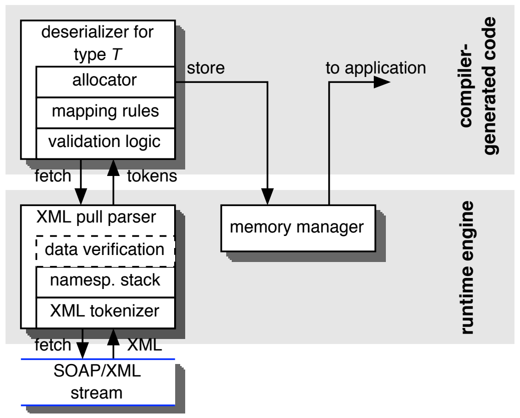


Fig. 12. Deserialization with XML parsing, validation, mapping rules, and memory allocation.

of run-time object graphs constrained only by the type information. The model is then used to generate the serialization and deserialization functions that are optimized by removing unnecessary pointer checks. Therefore, the serialization of a data structure that has no pointers does not incur any overhead at run-time.

5.2 Deserializing C/C++ Data from XML

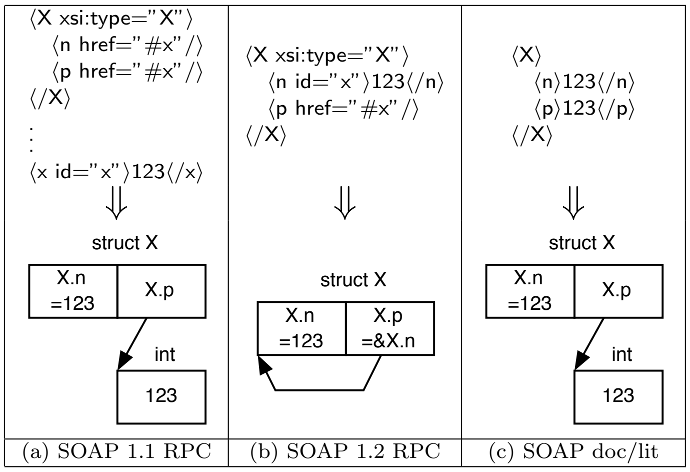
The inverse of serialization, i.e. deserialization, is a technique to decode XML representations of data structures. Combined XML parsing, validation, and data structure decoding takes place in one *soapcpp2*-generated function for each data type. Figure 12 shows an overview of the *soapcpp2*-generated deserializer and the interaction with the engine's pull parser and tokenizer. The deserializer of a type T is the controlling entity that maintains the parsing and deserialization state and fetches XML content from the stateless pull parser. The data is allocated by the memory manager and delivered to the application.

The *soapcpp2*-generated code consists of specialized XML pull parsers. The pull parsing technique is complemented with recursive descent parsing techniques, where the structure of the generated parsers is essentially derived from the XML schema structure of the XML data content. This effectively collapses the HTTP-XML stack by eliminating the need for a layered transport stack architecture. As a result, the subroutine calling depth is minimized, the amount of state information is decreased, and buffering requirements are reduced to a bare minimum.

More specifically, the *soapcpp2* compiler generates the following two functions to deserialize a type T:

—soap_default_T(**struct** soap∗, T∗). This function initializes data of type T with a default value. This function is called before deserializing the data. The initialization is important to ensure that members of structs and classes are properly initialized when XML elements and attributes are omitted. Note that the initialization of class instances can be accomplished through constructors. However, all other C/C++ types must be handled in this manner.

—soap_in_T(**struct** soap∗, **const char** ∗tag, T∗, **const char** ∗type). This function parses an inbound XML stream and populates a new structure of type T. The



```
⟨X xsi:type="X"⟩
  ⟨n href="#x"/⟩
  ⟨p href="#x"/⟩
⟨/X⟩
⋮
⟨x id="x"⟩123⟨/x⟩
```

```
⟨X xsi:type="X"⟩
  ⟨n id="x"⟩123⟨/n⟩
  ⟨p href="#x"/⟩
⟨/X⟩
```

```
⟨X⟩
  ⟨n⟩123⟨/n⟩
  ⟨p⟩123⟨/p⟩
⟨/X⟩
```

Fig. 13. Deserialized structures for different SOAP messaging styles.

function uses the stateless pull parser to fetch elements and attributes from an XML stream. Validation of structural constraints such as minOccurs, maxOccurs, and nillable occur within this function, Data is initialized with soap_default_T and struct/class members are recursively parsed using their soap_in functions and methods. Deserialized data is managed to prevent memory leaks, see Section 5.3.

In case T is a class, the deserialization functions are virtual methods of that class to facilitate the instantiation of derived classes to support XML schema type extensions. The XML content of derived instance is required to include the xsi:type="T" attribute value to distinguish it from the base class content.

The *soapcpp2* compiler generates deserializers that are specialized to reconstruct data structure graphs from multi-ref accessors in a SOAP message, i.e. when SOAP RPC 1.1/1.2 encoding with multi-ref accessors is used. To deserialize SOAP RPC encoded multi-ref accessors, the engine maintains a hash table for id and href attribute values. In SOAP 1.1 RPC encoding, most href references are forward pointing, i.e. the referenced accessor is parsed at a later point in the XML stream, usually after the method element ending tag. In SOAP 1.2 RPC encoding, XML elements can be marked with id attributed for multi-ref encoding without resorting to separate multi-ref accessors.

The deserialization process is more complex compared to the serialization process. It has to cope with differences in the encoding style (SOAP 1.1/1.2 RPC and document/literal) as well as differences in the XML representations produced by other SOAP toolkits that may or may not produce multi-ref. Consider the example shown in Figure 11(a). The serialized forms and the resulting data structures are shown in Figure 13(a), (b), and (c). Note that only SOAP 1.2 RPC encoding shown in Figure 13(b) gives an instantiation of a structurally identical copy of the original data structure.

To track the multi-ref accessors with id attributes and elements with href attributes, the gSOAP engine maintains a hash table with the id/href values. The hash table entries form chains of nodes containing the C/C++ type of the associated

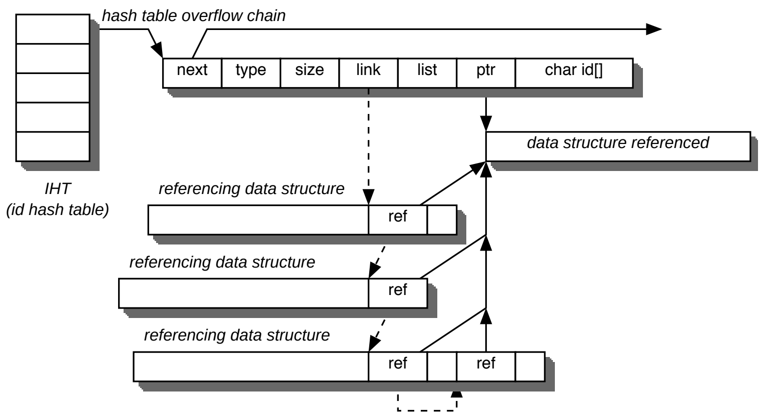

hash table overflow chain
next
type
size
link
list
ptr
char id[]
data structure referenced
IHT
(id hash table)
referencing data structure
ref
referencing data structure
ref
referencing data structure
ref
ref

Fig. 14. Pointer structures before (dashed arrows) and after (solid arrows) resolution of the forward references generated by hrefs in XML SOAP RPC-encoded messages.

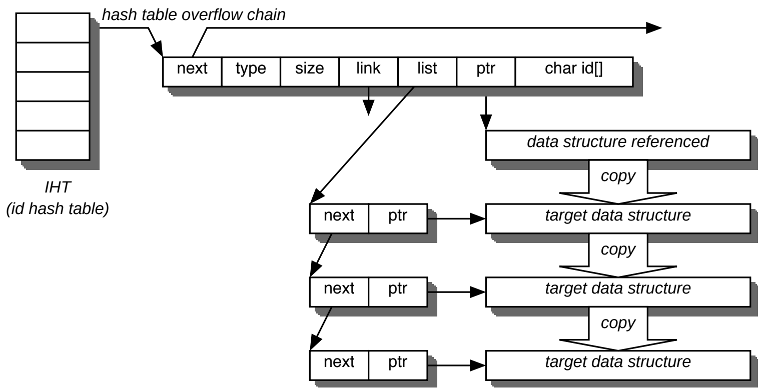


Fig. 15. Replication of a data structure from the referenced id location to hrefs.

data, the size of that data item, and the id string, see Figs. 14 and 15. The roles of the other three fields, link, list, and ptr, are described below:

—The link field is a linked list of unresolved forward pointers to the referenced data structure that is not yet instantiated, i.e. has not been parsed yet;

—The list field points to a linked list of nodes that point to memory regions into which the referenced data structure must be replicated;

—The ptr field points to the referenced data structure, when instantiated.

Figure 14 shows the changes to the linked list of pointers to resolve the references to the instantiated data structure. Figure 15 shows the replication of the referenced data structure into other structures. Note that this order is important: when replicated data structures contain pointer references, the pointer references must be resolved first before being copied.

5.3 Run-Time Memory Management

Memory management is an important aspect, especially for persistent services developed in C/C++ that cannot afford to leak memory. It is critical for any service-based application, because services must run uninterrupted for long periods of time. Therefore, memory leaks and dangling references are essential concerns that must be addressed in the overall design of a Web services framework. Newer programming languages, such as Java and C#, exploit runtime memory management techniques, such as garbage collection (GC). Certain specific development platforms for C++ also offer managed solutions. However, unmanaged memory allocation is still the most common form of memory management in C and C++ applications and typically the only form that can be adopted when programmers use pointer arithmetic or use slab allocators, e.g. in real-time systems.

To address these concerns, gSOAP's run-time engine implements an automated memory management facility that is portable, fast, and easy to use with C and C++ applications. The engine includes a managed memory space (MMS) to allocate and (semi-)automatically deallocate data. This approach has several benefits:

(1) The MMS is transparent by providing a similar interface as malloc, new, free, and delete.
(2) The decoder of a SOAP/XML message places all deserialized data in the MMS. The deserialized data can be removed at once. This prevents leaks, for example, when an application accidentally ignores (parts of) the data deserialized data[1].
(3) When an exception occurs during the deserialization process all partial data is automatically removed.
(4) Data in the MMS can be explicitly deleted when a client-server session ends (with soap_end). In that case all deserialized data is deleted at once. This is useful for servers to keep memory consumption in check. Note that the explicit use of soap_end after a session enables servers to maintain the state of deserialized data across multiple invocations.
(5) When a server or client application must retain (parts of) the deserialized data, then the data can be selectively moved out of the MMS into the user's data space with soap_unlink.
(6) The MMS interface can be used by the developer to allocate temporary data for a service response. The data is then automatically deleted by the server loop when the service operation ends.

The usefulness of the MMS is demonstrated with an illustrative example in C. Suppose a service operation returns a regular C string via one of its parameters, as shown in Figure 16. The ns__getVersion function uses soap_malloc at line 2 to allocate temporary data in the MMS of the current soap context ctx (the context is passed to the function by the engine). The memory allocated for the string is released (with soap_end) after the service operation is completed and the response was sent to the client, or when a fault occurred as shown on line 5. Note that if

[1]It is in fact quite natural to extend a complexType with any components so that additional elements can be added. Additional elements can be instantiated by the deserializer but ignored by the back-end application. Therefore, a back-end should not solely be responsible for deallocation.

```
int ns__getVersion(struct soap *ctx, char **version)
{ char *buf = soap_malloc(ctx, 100);  // allocate 100 bytes for temporary use
  int major, minor;
  if (check_release(&major, &minor))
    return soap_receiver_fault(ctx, "A problem occurred", NULL);
  sprintf(buf, "%d.%d", major, minor);
  *version = buf;
  return SOAP_OK;
}
```

Fig. 16. Memory allocation in service function with deferred memory release.

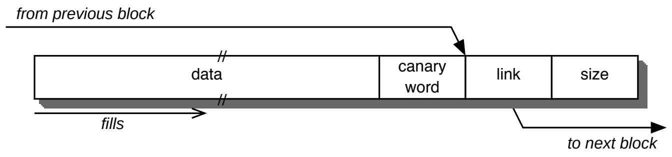


Fig. 17. Managed memory space block allocation layout.

malloc was used, the server logic would be much more complex to free the data before the next iteration of the server loop can commence (let alone the problem of passing the pointer value to free). In contrast, in C++ a service operation can be a method of a server object that manages application data, instead of a global function in C that leads to the aforementioned complications.

Note that the example does not show particularly good programming practice, since the limited buffer size may lead to buffer overruns. An additional feature of the MMS is to automatically detect overruns to avoid a crash of the server, possibly at the expense of a leak when the MMS is polluted. This design choice supports more intelligent debugging and recovery from memory problems resulting from leaks, double frees, and application-based overruns due to programming errors.

To implement these features the MMS is organized as follows. Each gSOAP run-time context maintains a pool of dynamically allocated blocks of data with additional management information. The blocks are allocated with malloc and chained together with the link field. The size field contains the size of the data segment and to find the starting position of the block. The format of a block is shown in Figure 17. There are two advantages of this layout. First, the data is located at the start of the block which facilitates the movement of the block from the MMS to the user space and eventual deletion with free. Second, the MMS guards against overruns with a “canary word”. When the canary word is changed, the MMS assumes the block is invalid.

Class instances are allocated differently in the MMS, but can also be deleted automatically when the service loop revolves to the next iteration. The *soapcpp2* compiler generates an allocator for each class to manage its instances in the MMS. In this way, temporary instances and deserialized instances can be deleted automatically.

Debugging client-server interactions between XML web services can be a complex task, especially when memory allocation issues arise. A service developed with the gSOAP framework can operate either in *production mode* for speed or in *debug mode* for testing. In production mode, the service engine is configured for performance and debugging information is not extensive. In debug mode, the messages and the engine status are logged and dynamic memory allocation tracked.

6. RESULTS

This section gives the results of the experimental evaluation of the performance and memory usage of applications developed with the gSOAP framework. The results are compared to other Web service toolkits. A performance model is presented that can be used to predict the performance of a gSOAP application based on the characteristics of a machine.

6.1 Portability and Interoperability Issues

Portability of the framework software is achieved by bundling all platform-dependent features within the *stdsoap2* engine, while keeping the generated code completely platform independent. Users need to run the tools only once to generate source code that can be deployed on a wide variety of platforms. The software has been tested for portability and performance on Windows (Win32, MS-DOS, and Cygwin), Linux, Unix (e.g. Solaris, HP-UX, FreeBSD, TRU64, Irix, QNX, and AIX), Mac OS X, and small and embedded OSes including VxWorks, WinCE, Palm OS, and Symbian devices. To address differences in the socket API or the absence of sockets, the transport layer can be replaced with a custom solution via a plugin.

The interoperability of gSOAP has been extensively tested. The gSOAP software was also checked against the WS-I Best Practices 1.0a [WS-I Organization 2003], see [van Engelen 2001].

6.2 Performance Results

In the next sections, the performance of gSOAP is compared to other toolkits and a performance prediction model for stand-alone gSOAP services for several types of machines, including 32-bit, 64-bit, single and dual core i386 and IPF CPUs.

Figure 18 shows the relative speedup of SOAP RPC messaging averaged over a large collection of Web service benchmarks on 2GHz Xeon Pentium 4 machines over a modest 100BaseT Ethernet LAN. These benchmark client-server applications exchange arrays ranging in size from 10 to 80,000 elements, see [Govindaraju et al. 2004; Head et al. 2005] for details. These arrays contain strings, integers, floats, and simple structs. The performance study shows that on average gSOAP outperformed all other toolkits tested[2].

Performance prediction allows developers to chose a machine that will give the best performance for a typical service application without requiring prior testing or simulation. A performance model is developed based on results collected for a typical Web service application by measuring the sustained performance of the message throughput on a wide selection of machines. It is assumed that a typical Web service application contains both textual and numerical data contained within XML schema complexTypes. The benchmark application exchanges SOAP RPC

[2]We had stability problems with AxisC++ which prevented us from completing the tests for arrays of more than 1000 elements.

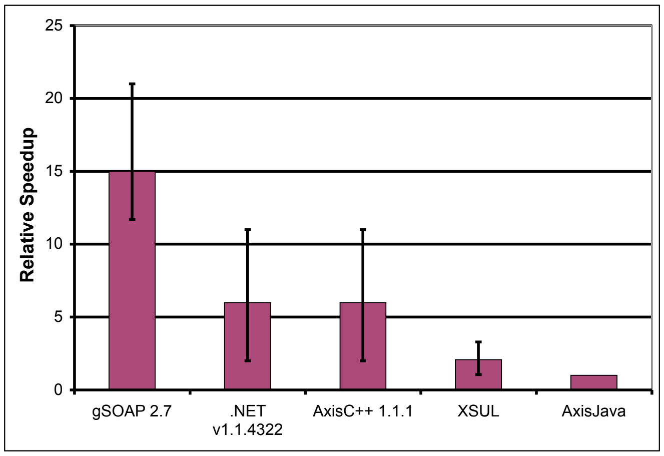


Fig. 18. Relative average speedup of Web services developed with several toolkits, compared to the base performance of AxisJava. The minimum and maximum measurements are also shown.

Table II. Throughput coefficients.

$(\alpha_{\min}, \alpha_{\max})$	32bit	64bit
i386	$(0.37, 0.45)$	$(0.72, 1.00)$
IPF	n/a	$(1.11, 1.20)$
PowerPC G4	$(0.42, 0.44)$	n/a
UltraSPARC-III	n/a	$(0.51, 0.53)$

request/response messages of 1.2KB with a struct (complexType) containing some lines of text and some numeric data. Non-persistent HTTP connections were used over a modest 100BaseT Ethernet LAN to simulate a common scenario in which a single message is fired to a server over a LAN. To determine performance prediction reliability, various configurations of machines with different Linux versions, gSOAP versions, and GCC settings were tested.

The predicted maximum throughput expressed in the number of round-trip request-response messages per second is:

$$max\_throughput = \frac{roundtrip\_messages}{sec} = \frac{\alpha_{\max} f}{c}$$

where f is the CPU clock frequency (cycles/sec) and $c = 740 \cdot 10^3$ CPU cycles per message (CPU cycles determined with Linux 2.6.5 and gcc 3.3.3 -O3).

The CPU architecture characteristics are determined by the $\alpha_{\min}$ and $\alpha_{\max}$ parameters listed in Table II, where $\alpha_{\min}$ and $\alpha_{\max}$ are the lower and upper bound for the sustained throughput, respectively. The performance prediction model was evaluated by testing the maximum performance on a set of machines shown in Figure 19. The figure shows the predicted maximum throughput and actual maximum throughput obtained.

From this figure several observations can be made:

—The relative performance on i386-compatible 64-bit systems (dark bars) is more

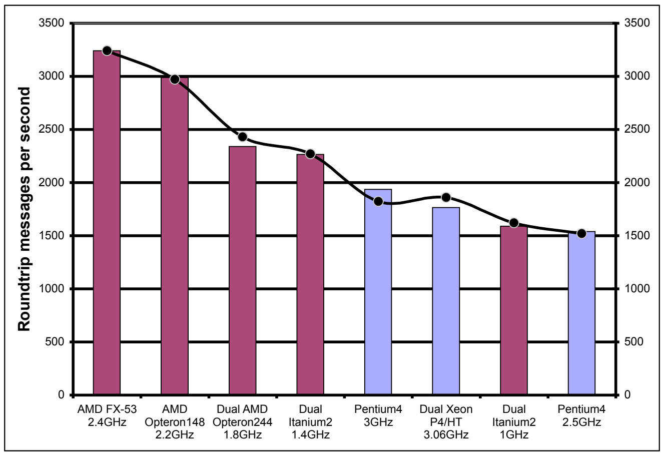


Fig. 19. Predicted maximum throughput (line) and actual maximum throughput obtained (bars) of a benchmark service on various machines.

than twice as high as on i386-compatible 32-bit systems (light bars), when the relative performance is expressed as total throughput over CPU clock frequency, i.e. the performance of 64-bit systems is twice that of 32-bit systems for the same CPU clock frequency.

—The IPF machines have a low clock frequency but their performance is competitive to i386 because the IPF machines appear to have a higher efficiency per clock tick ($1.11 \leq \alpha \leq 1.20$ as shown in the table). This means that reasonable levels of instruction-level parallelism are obtained and IPF machines are expected to outperform i386 machines for the same CPU clock frequencies.

—The range for α is wide for 32-bit i386 systems and very narrow for IPF. This means that the performance predictability is better for the latter type of machines.

—The message throughput is independent of the number of cores, which is not surprising since the benchmark tested is single threaded.

The performance model is simple and can be further refined using LAN and OS characteristics. However, SOAP/XML performance is often CPU bound [Govindaraju et al. 2004] and the model should therefore be a reasonable predictor for expected message throughput under typical circumstances.

6.3 Memory Footprint

Limited memory usage is especially important for small devices. Table III shows the total memory usage of gSOAP applications given in *static_code_size*(+*max_data_size*) for three small devices with different operating systems and CPUs. The three benchmark client applications were compiled with CodeWarrior 9.0 (opt: minimize size), MS Embedded Visual C++ 3.0 (opt: minimize size), and gcc 4.1 (opt: -O1), respectively. The code size was reduced with -DWITH_LEAN compilation flags, which reduces the total code size of the gSOAP engine up to 20%. Further code

Table III. Memory footprint statistics of gSOAP applications on selected small devices.

	getQuote	**GoogleAPI**	**WhiteMesa interop R2**
PalmOS (Motorola 68K)[a]	66K(+1.2K)	82K(+6.2K)	190K(+22K)
WinCE (ARM5)	90K(+1.2K)	114K(+6.2K)	179K(+22K)
Linux (i386)	73K(+1.2K)	122K(+5.9K)	261K(+22K)

[a] includes code for a graphical user interface, shown at the right.

size reductions of up to 30% can be obtained by eliminating attachment support using the -DWITH_LEANER compilation flag.

Applications are shown in the table with increasing complexity from left to right. The getQuote client written in C displays the value of a stock quote using the XMethods' delayed stock quote service [XMethods 2004], which has a single service operation with two parameters. The GoogleAPI client written in C retrieves Google search results, checks spelling, or retrieves a cached Web page. The memory footprint for a Google search operation is shown for 10 result entries with all details stored in dynamically allocated array of 10 structs. The WhiteMesa interop round 2 application is a C++ client to test the SOAP RPC serialization interoperability, where the set of 'A' tests covers 18 service operations with a total of 145 serializable data types, the 'B' tests cover 5 service operations with a total of 48 serializable data types, and the 'C' tests cover SOAP header processing with just one operation and a total of 25 serializable data types. The serializable data types are the parameter types of the C/C++ service operations used in the interop application, which is larger than the number of data types used in the WhiteMesa interop WSDLs due to additional uses of pointer-based types in C/C++.

From the table it can be concluded that the footprint modestly increases with increasing complexity of the gSOAP Web service client application.

7. CONCLUSIONS

The term service-oriented architecture refers to a particular aspect of computing with services as network-enabled software services that can be discovered, described, and invoked via RPC or message passing. A SOA basically supports the requirements of software users by assembling services into composite applications. The use of services as building blocks promotes software reuse and eliminates redundancy by encapsulating enterprise logic in centralized services for use by clients and other services.

Interoperability of services in a SOA is the key to its success. In this respect, effective use of XML Web service protocols is an important aspect to building a SOA. When existing C/C++ applications are exposed as XML Web service components, ultimately the middleware is responsible for handling the embedding of application-specific data structures in XML messages via XML-serialized data streams. Because programming language data types do not have a one-to-one correspondence to XSD types, considerable requirements are forced on the implementation of the middleware to support the deployment of a service, including protocol compliance, ease-of-use with programming language bindings, and performance aspects.

This paper defined the requirements and solution space for deploying C/C++ services in a SOA. An open-source framework was presented. The framework offers an intuitive programming model that extends the C and C++ languages with XML bindings. The bindings expedite the reuse of C/C++ software by exposing service operations in C/C++ as native functions and data structures. The mapping of XML to and from C/C++ data is automated using static (compile-time) and dynamic (run-time) methods. The effectiveness of the approach was evaluated with benchmark services, which showed excellent performance and other desirable properties such as a small memory footprint.

8. ACKNOWLEDGMENTS

The author would like to acknowledge Martin Kuba from the Supercomputing Center, Institute of Computer Science, Masaryk University, Brno, Czechoslovakia, for testing the performance of gSOAP v2.5–v2.7 on a range of platforms.

REFERENCES

Abu-Ghazaleh, N. and Lewis, M. 2005. Differential deserialization for optimized SOAP performance. In *proceedings of the ACM/IEEE conference on Supercomputing.* IEEE Computer Society, Los Alamitos, CA, 21–31.

Abu-Ghazaleh, N., Lewis, M., and Govindaraju, M. 2004. Differential serialization for optimized SOAP performance. In *proceedings of the IEEE International Symposium on High Performance Distributed Computing.* IEEE Computer Society, Los Alamitos, CA, 55–64.

Apache Foundation. 2002. Apache Axis project. Available from http://ws.apache.org/axis.

Ayala, D. 2002. NuSOAP for PHP. Available from http://sourceforge.net/projects/nusoap/.

Chiu, K. 2003. Compiler-based approach to schema-specific XML parsing. Tech. Rep. Computer Science Technical Report 592, Indiana University.

de Icaza, M. 2004. The Mono project. Available from www.mono-project.com.

Erl, T. 2005. *Service-Oriented Architecture: Concepts, Technology, and Design.* Prentice Hall, Upper Saddle River, NJ.

Fielding, R. T. 2000. Architectural styles and the design of network-based software architectures. Ph.D. thesis, University of California, Irvine.

Govindaraju, M., Slominski, A., Chiu, K., Liu, P., van Engelen, R., and Lewis, M. 2004. Toward characterizing the performance of SOAP toolkits. In *proceedings of the ACM/IEEE International Workshop on Grid Computing.* IEEE Computer Society, Los Alamitos, CA, 365–372.

Head, M. R., Govindaraju, M., Slominski, A., Liu, P., Abu-Ghazaleh, N., van Engelen, R., Chiu, K., and Lewis, M. J. 2005. A benchmark suite for SOAP-based communication in Grid Web services. In *proceedings of ACM/IEEE Supercomputing Conference.* IEEE Computer Society, Los Alamitos, CA.

Hericko, M., Juric, M. B., Rozman, I., Beloglavec, S., and Zivkovic, A. 2003. Object serialization analysis and comparison in Java and .NET. *ACM SIGPLAN Notices 38,* 8, 44–54.

Kostoulas, M. G., Matsa, M., Mendelsohn, N., Perkins, E., Heifets, A., and Mercaldi, M. 2006. XML screamer: an integrated approach to high performance XML parsing, validation and deserialization. In *proceedings of the International Conference on World Wide Web.* ACM Press, New York, 93–102.

Kulchenko, P. 2003. SOAP::Lite for Perl. Available from www.soaplite.com.

Loughran, S. and Smith, E. 2005. Rethinking the Java SOAP stack. In *proceedings of the IEEE International Conference on Web Services (ICWS).* IEEE Computer Society, Los Alamitos, CA, 12–15.

Meijer, E., Schulte, W., and Bierman, G. 2003. Programming with circles, triangles, and rectangles. In *proceedings of the XML Conference.* DeepX Ltd., Philadelphia, PA.

Open SOA Collaboration. 2006. Service component architecture. Available from www.osoa.org.

Thomas, D. 2003. The impedance imperative tuples + objects + infosets = too much stuff! *Journal of Object Technology 2,* 5 (October), 7–12.

UDDI Organization. 2005. The universal description, discovery, and integration (UDDI) specification. Available from www.uddi.org.

van Engelen, R. 2001. The gSOAP toolkit for C and C++ Web services. Available from http://gsoap2.sourceforge.net.

van Engelen, R. 2003. Pushing the SOAP envelope with Web services for scientific computing. In *proceedings of the International Conference on Web Services (ICWS)*. CSREA Press, Las Vegas, NV, 346–352.

van Engelen, R. and Gallivan, K. 2002. The gSOAP toolkit for web services and peer-to-peer computing networks. In *proceedings of the IEEE International Symposium on Cluster Computing and the Grid*. IEEE Computer Society, Los Alamitos, CA, 128–135.

van Engelen, R., Govindaraju, M., and Zhang, W. 2006. Exploring remote object coherence in XML Web services. In *proceedings of the IEEE International Conference on Web Services (ICWS)*. IEEE Computer Society, Los Alamitos, CA, 249–257.

van Engelen, R. A. 2004. Constructing finite state automata for high performance XML web services. In *proceedings of the International Symposium on Web Services (ISWS)*. CSREA Press, Las Vegas, NV.

van Heesch, D. 1997. Doxygen. Available from www.doxygen.org.

W3 Consortium. 2000. SOAP 1.1 and 1.2 specifications. Available from www.w3.org.

W3 Consortium. 2001. WSDL Web services description language 1.1 specification. Available from www.w3.org.

W3 Consortium. 2004. XML Schema 1.1 specification. Available from www.w3.org.

W3 Consortium. 2006. Namespaces in XML 1.1 (second edition). Available from www.w3.org.

WS-I Organization. 2003. Basic Profile BP1.0a. Available from www.ws-i.org.

XMethods. 2004. XMethods service listings. Available from www.xmethods.com.

Zhang, W. and van Engelen, R. 2006. Table-driven XML streaming - a methodology for web service performance optimization. In *proceedings of the IEEE International Conference on Web Services (ICWS)*. IEEE Computer Society, Los Alamitos, CA, 197–206.

Received December 2005; revised May 2006; accepted December 2006