XQuery at Your Web Service

Nicola Onose
Ensimag
BP 72
38402 Saint Martin d'Hères Cedex, France
nicola.onose@ensimag.imag.fr

Jérôme Siméon*
IBM Watson Research Center
19 Skyline Drive
Hawthorne, NY 10532, USA
simeon@us.ibm.com

ABSTRACT

XML messaging is at the heart of Web services, providing the flexibility required for their deployment, composition, and maintenance. Yet, current approaches to Web services development hide the messaging layer behind Java or C# APIs, preventing the application to get direct access to the underlying XML information. To address this problem, we advocate the use of a native XML language, namely XQuery, as an integral part of the Web services development infrastructure. The main contribution of the paper is a binding between WSDL, the Web Services Description Language, and XQuery. The approach enables the use of XQuery for both Web services deployment and composition. We present a simple command-line tool that can be used to automatically deploy a Web service from a given XQuery module, and extend the XQuery language itself with a statement for accessing one or more Web services. The binding provides tight-coupling between WSDL and XQuery, yielding additional benefits, notably: the ability to use WSDL as an interface language for XQuery, and the ability to perform static typing on XQuery programs that include Web service calls. Last but not least, the proposal requires only minimal changes to the existing infrastructure. We report on our experience implementing this approach in the Galax XQuery processor.

Categories and Subject Descriptors

D.2 [**Software**]: Software Engineering

General Terms

Languages, Standardization

Keywords

Web services, XML, XQuery, WSDL, Interface, Modules.

1. INTRODUCTION

Web services are considered by many to be the next revolution that will allow the Web to distribute not only documents and data, but also applications. XML lays at the heart of the Web services infrastructure. It is used to describe services themselves (using WSDL [25, 26]), as a format for the messages that are exchanged between services and applications (using SOAP [20]), and to describe the structure of those messages (using XML Schema [21, 17]). Indeed, because of its flexibility, XML greatly facilitates the deployment, as well as access, composition, and maintenance of Web services. Yet, current approaches to Web services development tend to hide the XML layer behind Java or C# APIs, preventing the application to get direct access to the original Web services description and semantics. In this paper, we propose tools and techniques which enable the use of a native XML language, namely XQuery, as an integral part of the Web services infrastructure.

To better understand where Web services development may benefit from the use of XQuery, Figure 1 shows a typical Web service architecture. The purpose of a Web service is to make some operations available to one or more target application(s). WSDL can be used in conjunction with XML Schema to describe those operations **(1)**, including their input parameters and their result. Both the service and the application are built around this description, and must implement some messaging layer **(2)** that deals with the creation and manipulation of the XML messages based on that description. Finally, the corresponding logic, both on the service side and on the application side needs to be implemented **(3)**. Note that building the service itself may also require to access information located in one or more legacy repositories **(4)**, which themselves might be another source for XML data.

Currently, existing Web services development environments use Java or C# for both the application/server logic and the messaging layer. In order to deal with the mismatch between XML Schema and object-oriented type systems, they apply some code generation from the WSDL description into OO classes. This approach has several drawbacks. First, OO classes are often not a natural representation for the original XML, making it more difficult for the application to recover the actual semantics of the messages. Second, those classes must be regenerated and recompiled whenever the structure of the messages changes. Third, the XML messages have to be encoded/decoded, and validated, at run time. Finally, there is little use of the XML Schema information at compile time to detect errors in programs, making development error-prone and programs difficult to debug. We argue for a hybrid approach where XQuery is used to handle the messaging layer (possibly along with part of the application logic), while a general purpose programming language is still available for dealing with the application.

Using XQuery to handle the messaging layer gives the application a more direct access the original Web Service content and semantics. XQuery provides features that can be used to directly access the content of messages (e.g., using XPath expressions), as well as to construct new messages (using XML construction syntax) to be passed to a Web service operation. Finally, part of the application/service logic may also benefit from some of the capabilities of XQuery. For instance, XQuery might be used to access and process the information from a back-end source. This hybrid

*The work was done while this author was at Bell Laboratories, Murray Hill, NJ 07974, USA.

WWW2004, May 17–22, 2004, New York, New York, USA.
ACM 1-58113-844-X/04/0005.

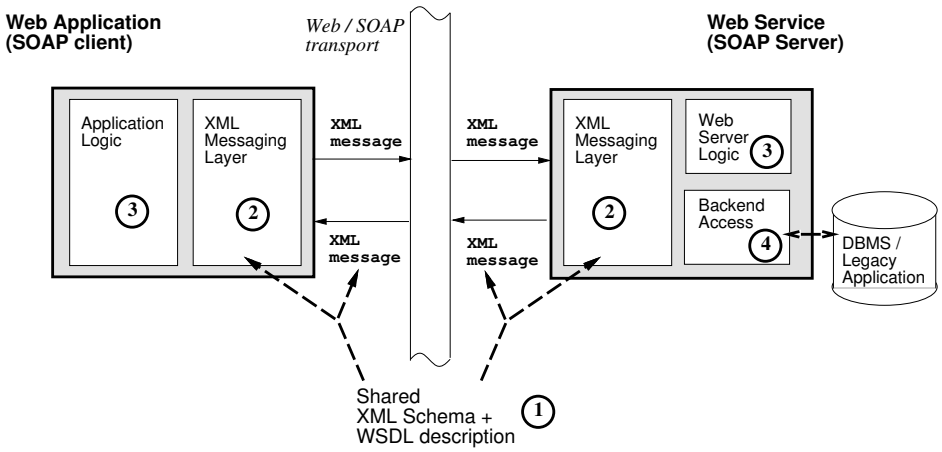

Figure 1: Archetypal Web Service Architecture

approach requires proper interfaces between WSDL, SOAP and XQuery on one hand, and XQuery and OO languages on the other hand. The latter is the topic of a number of on-going research [10], and industrial activities [19, 29] and will not be addressed here. Instead, we focus on the integration between XQuery, SOAP and WSDL. More precisely, the paper makes the following technical contributions:

- We define a binding between XQuery and WSDL, based on the relationship between XQuery modules and WSDL ports. We identify constraints that must be fulfilled by the WSDL description and the XQuery module that it is bound to.
- We extend XQuery with a Web services import statement that provides transparent access to Web services from within an XQuery program. Importing a service can be done with one line in the XQuery prolog, as in:

```
import service namespace myservice =
   "http://www.myservice.com/"
   name "MyWebService";
```

- We show how to deploy a SOAP service from a given XQuery module. We describe a simple command-line tool that can perform the necessary installation:

```
xquery2soap -installdir /var/www/html
            MyXQueryService.xq
```

- The proposed approach can be implemented with only minimal changes to the existing infrastructure. We describe our own implementation in the Galax[1] XQuery processor [7]. In particular, we show how to use XQuery itself as a stub language to implement the Web service import and deployment mechanisms.
- We illustrate additional benefits of the approach. Notably, we show how to use WSDL as an interface for XQuery modules, allowing a form of *hiding* similar to the one found in modern functional programming languages [16, 6, 15], and we explain how to use static typing in order to detect errors in XQuery programs that include Web service calls.
- Finally, we identify specific mismatches between XQuery and WSDL, and suggest some possible solutions that may further improve the coupling between the two technologies.

[1] `http://db.bell-labs.com/galax/`

Related work

Bindings between XML and object-oriented languages [13, 5] are currently the most widely used approach, but none of those bindings give the ability to manipulate the original XML information natively. Web services composition languages [27, 24, 23, 22] focus on the orchestration between services and provide only lose coupling with native XML languages such as XPath and XQuery. Full-fledged programming languages for XML such as XDuce [11] or CDuce [4] have the potential to support arbitrary XML application development but do not address the specific needs of Web services. Closer to our approach is the work on the XL language [9, 8], a native XML language for Web service development. However, XL focuses on extensions to allow XQuery to support end-to-end Web services development, and does not provide details about how to connect to the existing Web services infrastructure. Instead, we advocate an hybrid XQuery-OO approach which requires minimal changes to XQuery. We believe the proposed WSDL-XQuery binding is a simple but essential step that immediately brings the benefits of native XML languages to Web services development. Finally, the ActiveXML project at INRIA [18, 2], is a parallel project which tries and address the need of Web service development. They focus on a document-driven approach, where XML documents can be extended to contain service calls, while our approach is more a language-driven approach, which puts XQuery at the heard ot Web services development.

Organization of the paper

The rest of the paper is organized as follows. Section 2 reviews key aspects of WSDL and XQuery that are necessary for our work. Section 3 describes the import service extension to XQuery, and its implementation in Galax. Section 4 describes the `xquery2soap` tool that can be used to deploy a Web service out of an XQuery module, and its implementation in Galax. Section 5 gives a precise definition for the XQuery-WSDL binding itself, and defines the constraints that apply to it. Section 6 discusses additional benefits of the approach, and suggests some changes to XQuery and WSDL to support a tighter integration. Section 7 concludes the paper and outlines some future work.

2. XQUERY AND WSDL

The proposed binding takes advantage of the module system recently added to XQuery to provide a tight coupling between WSDL

and XQuery. In this section, we briefly review the features of XQuery and WSDL that we will use. The examples in the paper are based on a simplified application scenario for a distributed user profile management system. This scenario is inspired by the GUPster project at Lucent whose goal is to support unified user profile management across multiple service providers, based on the 3GPP GUP standard [1].

XQuery expressions and functions

XQuery [30] is the W3C XML Query language. The design of XQuery is that of a small functional language [31], based on a set of expressions that can be composed together arbitrarily. Those expressions include navigation in XML documents using XPath [28], database statements (the so-called FLWOR expressions), construction of new XML values, operations on XML Schema types, and function calls. More significant in the context of Web services is the ability for XQuery users to define their own functions. For instance, the following function computes some contact information with the name and the first phone number found for a given contact (in variable `$contactid`), within a given user profile (in variable `$ownername`).

```
declare function gup:getContact (
    $ownername as xs:string,
    $contactid as xs:integer
  ) as element(contact) {
  let $prof := //profile[@owner = $ownername],
      $c := $prof//contact[@id = $contactid]
  return
   <contact>
     <name>{ concat($c/first,$c/last) }</name>
     <tel>{ $c/phone[1] }</tel>
   </contact>
};
```

A function has a name (here `gup:getContact`), takes some input parameters, and returns a result. Each parameter is identified by a variable name and a type. XQuery can refer to existing XML Schema types using "sequence types", for instance: `element(contact)` is a sequence type that refers to the globally defined `contact` element declaration. The body of the function is composed of an XQuery expression that computes the result from the function's input parameters. In our example, the query uses XPath syntax to perform a lookup inside the profiles database (assumed here to be an XML document), extracts the appropriate contact information, then constructs a `contact` element which contains the name and first telephone number found for the identified contact. Note that element construction in XQuery has the same syntax as XML, and uses curly braces to switch back to the XQuery expression syntax.

We refer the reader to the growing literature on XQuery [12, 14] for a more gentle and complete introduction to the language.

XQuery modules

XQuery adopted a notion of module in its May 2003 working draft. Modules provide a means to regroup and identify a set of type declarations, global variables and functions as a single *unit*. For instance, the following defines a module "gup", importing the user profile schema, and defining the same function `gup:getContact` as earlier.

```
module namespace gup = "http://example.net";

import schema
  "http://example.net/UserProfile.xsd";

declare function gup:getContact ( ...
```

A module can then be *imported* from within another module, as follows:

```
import module
   namespace gup = "http://example.net";

gup:getContact("jsimeon",4)
```

Once imported, the functions defined in that module are available to the query. In our example, the imported function is used to retrieve the contact whose id is `4` from the profile of `jsimeon`.

```
<definitions
   targetNamespace="http://example.net"
   xmlns:tns="http://example.net" ...>

  <types>
   <xs:schema targetNamespace="http://example.net">
    <xs:element name="contact">
      <xs:complexType>
        <xs:sequence>
          <xs:element name="name" type="xs:string"/>
          <xs:element name="tel" type="xs:string"/>
        </xs:sequence>
      </xs:complexType>
    </xs:element>
   </xs:schema>
  </types>

  <message name="getContact">
   <part name="ownername" type="xs:string"/>
   <part name="contactid" type="xs:integer"/>
  </message>

  <message name="getContactResponse">
    <part name="result" element="tns:contact"/>
  </message>

  <portType name="UserProfilePort">
   <operation name="getContact">
    <input message="tns:getContact"/>
    <output message="tns:getContactResponse"/>
   </operation>
  </portType>

  <binding name="UserProfileSOAP"
           type="tns:UserProfilePort">
     <soap:binding style="rpc"
      transport="schemas.xmlsoap.org/..."/>
     <operation name="getContact">
      ...
     </operation>
  </binding>

   <service name="UserProfile">
      <port name="UserProfilePort"
            binding="tns:UserProfileSOAP">
        <soap:address
         location=
         "http://example.net/services/gup.xqs"/>
      </port>
  </service>
</definitions>
```

Figure 2: WSDL for a user profile service

WSDL

WSDL stands for the Web Services Description Language. Here we explain how ports and bindings work, and also what the relationship between WSDL and XML Schema is (for use in operation's signature). We used WSDL 1.1 as the basis for this work, as it appears to be the version most widely supported [3]. Note that the binding can easily be adapted to WSDL 2.0 [26], although some changes would be required (e.g., `portType`s have been renamed to `interface`s in WSDL 2.0).

Figure 2 shows the WSDL for a simplified service giving access to user profile information.

A Web service description is typically composed of the following elements:

- A set of `types` used by the service, and defined using XML Schema.
- A set of `messages`, composed of one or more `part`(s) which are the components of that message. Each `part` has a name and a type (described using XML Schema).
- A `portType` which regroups all the `operations` supported by a service. Each `operation` is described by a name, an input message and an output message.
- A `binding` element which identifies an implementation for the service (e.g., using SOAP).
- A `service` element which defines the service itself, and associates a `portType` to its `binding`.

In our example, the *UserProfile* service provides a single operation called *getContact* which takes as input the user profile owner's name (of type `xs:string`) and the id of the contact that is requested (of type `xs:integer`), and returns a `contact` element.

Connecting XQuery modules to WSDL

There is a natural parallel between XQuery modules and WSDL descriptions. The key remark is that an operation input/output messages in the WSDL `portType` are in essence similar to the function signature in the XQuery module. Each operation behaves like a function, and each input part for the operation corresponds to a parameter of that function. Also, both are using XML Schema types to describe their input parameters and their ouput. There is no equivalent in WSDL for the function's body in XQuery. Instead, WSDL allows to define a *binding*, which gives some information about how to contact the service (in our example a SOAP server). Of course, the corresponding service must effectively accept calls according to the WSDL description and implement those operations.

The idea behind our approach is to exploit this parallel to provide a tight coupling between XQuery modules and WSDL descriptions.

3. SERVICE IMPORT IN XQUERY

We now describe an extension that allows XQuery programs to access Web services.

3.1 Example

The import service statement is very simple. One just needs to identify the WSDL resource, the name of the service, and the port which must be accessed (in the case there exists several ports for the service). Optionally, the WSDL target namespace may be bound to a prefix for use within the rest of the query. For instance, the following statement can be used to import the user profile service presented in the previous section in the query.

```
import service
 namespace gup = "http://example.net"
  name "UserProfile";

let $c := gup:getContact("jsimeon",4)
return $c/tel
```

Once the service is imported, the operations from that service are available to the query *as standard XQuery functions*. In our example, the imported Web service operation is used to retrieve the contact whose id is `4` from the profile of `jsimeon`. Calling the XQuery function `gup:getContact` triggers a SOAP call to the appropriate Web service. The input parameters from the function call are passed to the corresponding Web service operation. Once the service returns a result for that call, this result is passed back to the XQuery function and the query evaluation may proceed (here by applying a simple path expression which extracts the telephone number out of the result of the service call).

It is important to note that because the values manipulated by XQuery and the SOAP messages are both in XML, the user does not need to perform any conversion of the input parameters (resp. of the result) before (resp. after) calling the service. The query can use the Web service operation as if it were an XQuery function whose type signature is the one given in the WSDL description. Also note that the similarity between the `import service` statement and XQuery's standard `import module` statement is not fortuitous. In fact, the WSDL import behaves exactly like a module import and has a similar semantics [31] as far as XQuery is concerned. We will see more in Section 5 about the relationship between XQuery modules and WSDL descriptions.

The import service statement is very concise. This facilitates both access to and composition of multiple Web services. For instance, the following XQuery program imports both the user profile service and the Google search service[2], and compose operations from the two services to retrieve Web pages containing the telephone number of the contact with id `4`.

```
import service
  namespace gup = "http://example.net"
  name "UserProfile";

import service
  namespace google = "urn:GoogleSearch"
  name "GoogleSearchService";

let $c := gup:getContact("jsimeon",4)
return google:doGoogleSearch($c/tel,...)
```

3.2 Service Import Statement

The full syntax for the proposed import service statement is as follows:

```
import service
  (namespace NCName =)?    (: namespace prefix :)
  StringLiteral            (: target namespace :)
  (at StringLiteral)?      (: location hint :)
  name StringLiteral       (: service name :)
  (port StringLiteral)?    (: port name :)
```

The Web service is identified using its target namespace. The target namespace, or an optional location hint, may be used to retrieve the corresponding WSDL description which contains the necessary information to implement the service import. The name of the service must be provided as well, and optionally the port name (in case there are multiple ports defined for that service). In case there are several ports, each of them may be imported independently. The rationale behind this choice is that each port in fact corresponds exactly to one XQuery module.

Note that from a user point of view, the import service statement behaves the same, independently of the actual binding for the service. Obviously, the actual implementation for the import must

[2] `http://www.google.com/apis/`.

take the actual binding into account. In the rest of the section, we report on our experience in implementing the `import service` to access a SOAP service.

3.3 Implementing SOAP Service Import

Implementing access to a SOAP service in Galax is done in two steps. First we add a simple low-level, generic, SOAP call to the XQuery engine. Second, we implement the `import service` statement by compiling the WSDL description and the corresponding binding information into a specific XQuery stub that is imported as a standard XQuery module. The approach makes the stub generation very simple since we can use the XQuery's operations to construct and access the necessary SOAP envelopes.

Generic SOAP call

A SOAP [20] call is typically implemented as an HTTP request that follows certain conventions concerning the encoding of XML parameters for the service operation. Usually, a SOAP call has an HTTP header beginning with the POST method, or another HTTP method mentioned in the 'verb' attribute of an *http:binding* element from the WSDL file of the web service. The header must also contains a *SoapAction* field (possibly empty) and a MIME type that tells the server that it receives XML data: *Content-Type: text/xml; charset=utf-8*. The result of the http request is then decoded to extract the appropriate XML response.

Galax already supports standard HTTP requests. HTTP requests are implemented in Galax using the `libcurl` library[3] that offers high-level routines for sending requests for the most common Internet protocols: HTTP, HTTPS, FTP, FTPS, GOPHER, LDAP etc. The programmer can also specify options such as user and password, timeout, stream buffer callbacks or the way to deal with invalid SSL certificates.

In order to support SOAP calls, we just added a new built-in function, called `glx:soap-call`, whose signature is the following:

```
declare function
  glx:soap-call(xsd:anyURI,
                xsd:string,
                xsd:string,
                item()*) as (document) external;
```

The first argument represents the URI of the server (as specified in the *soap:address* or *http://address* and *http:location* elements in the WSDL). The second argument is the string representation of the HTTP method to be used ("GET" or "POST"). The third is the concatenation of additional HTTP headers (such as *SoapAction:* ...) and the last one is the body of the SOAP message made up of untyped XML data.

Stub generation

From the WSDL description, we generate a stub which implements the XQuery interface in terms of the underlying protocol level (using the `glx:soap-call` function). Compiling the WSDL into the corresponding XQuery stub is done in four steps: (1) the WSDL file is parsed as an XML document, (2) the *WSDL loader* interprets that XML document as a WSDL description and builds a WSDL abstract syntax tree (AST), (3) the *WSDL compiler* takes the resulting AST and generates the stub as an XQuery module, finally (4) that module is imported as a standard XQuery module.

The result of the `import service` statement in our earlier example is equivalent to the import of an XQuery module containing a single function with the same type signature as the first XQuery *getContact* function we defined at the beginning of the section[4]:

```
module
namespace gup =
 "http://example.net/UserProfile#UserProfilePort";

import schema
 "http://example.net/UserProfile.xsd";

define function
gup:getContact($ownername as xs:string,
               $contactid as xs:integer)
               as element(contact)
{
 let $input :=
   <soapenv:Envelope>
     ...
     <tns:getContact>
       <ownername>{$ownername}</ownername>
       <contactid>{$contactid}</contactid>
     </tns:getContact>
     ...
   </soapenv:Envelope>
 let $output :=
   glx:soap-call(xsd:anyURI("http://example.net"),
                 "POST", "SoapAction: ", $input)
 return
  $output/soapenv:Envelope/soapenv:Body/
    tns:getContactResponse/tns:contact
};
```

Note that the module imports the schema `UserProfile.xsd` which contains the schema that corresponds to the `type` element in the WSDL description. Also note that the target namespace for the module contains both the service target namespace and the port name, in order to deal properly with services that contain multiple ports.

In practice, the generated stub is small[5]. This is due to a conjunction of two things. First, the stub can leave most of the message 'as is', and second, XQuery expressions can be used to conveniently construct and access the necessary SOAP envelope.

4. DEPLOYING AN XQUERY SERVICE

In this section, we describe how to use the XQuery-WSDL binding to deploy a Web service from a given XQuery module. The `xquery2soap` top-level application exports an XQuery module as a SOAP-apache server. Server responses are serialized by an XQuery server skeleton according to the SOAP specifications.

4.1 Example

We will illustrate these procedures by deploying the `gup` module from our first example in Section 2.

```
xquery2soap
 -installdir "/var/www/services"
 -interfacedir "/var/www/services/wsdl"
 -address "http://example.net/gup.xqs"
 gup.xq
```

As a result of this call, a WSDL file similar to that in Figure 2 is generated. Based upon that WSDL file, an XQuery SOAP server is

[3]See `http://curl.haxx.se/libcurl/`.

[4]Due to space limitations, we only show the stub built when using the WSDL RPC style. The alternative WSDL document style can be generated in a similar way. See [25] for more details about the two encoding styles.

[5]As an illustration the complete stub for the Google search service can be found at `http://db.bell-labs.com/xbutler/`.

built and copied along with *gup.xq* to the location of the apache service directory (here "`/var/www/services`"). In addition, the tool generates the appropriate stub (denoted '`gup.xqs`' in the following) between the apache server and the XQuery module. Compared to the stub generated in the `import service` case, this one plays the inverse role: it extracts the parameters from the SOAP messages and passes them to the appropriate XQuery function in the XQuery module. The result of the XQuery call is then wrapped back again as a soap message which is sent back as a result to the calling application.

4.2 The Syntax of `xquery2soap`

The complete syntax for the export application is:

```
xquery2soap
   [-wsdl WSDL]
   [-port WSDLPort] [-binding WSDLBinding]
   [-installdir Directory]
   [-interfacedir Directory]
   [-address URI]
   XQueryModule
```

The mandatory `XQueryModule` argument indicates the module to be exported and accessed via the (SOAP) server XQuery program. The optional `WSDL` argument designates the WSDL interface used for the module. If it is present, the system checks for consistency between the WSDL interface and the corresponding module. If it is absent, an interface is created automatically, respecting the same conventions of passing from XQuery functions to WSDL parameters as those for the XQuery binding. In this case, the names of the SOAP port and the SOAP binding can be given with the options `-port` and `-binding` respectively. The `-installdir` option specifies the directory in the local file system where the SOAP server will be installed. The WSDL interface can also be published with the `-interfacedir` argument. Finally, the `-address URI` is the Web address of the server and is used to build the `soap:address` element in the WSDL `port`. If absent, the name of the local host concatenated with the module name is assumed by default.

4.3 Server Implementation

The `xquery2soap` is meant to produce an XQuery program that would act as a SOAP server called through apache. Again, an important component of that SOAP server is an XQuery stub which maps generic SOAP calls into the appropriate XQuery function call.

The XQuery stub starts with the statement.

```
import module namespace gup="http://example.net"
```

The rest of the stub is dispatching the call to the appropriate function in the XQuery module. The result of that function is then wrapped into the proper SOAP response.

```
declare function
  local:make_envelope
   ($funname as xs:string,
    $nmsname as xs:string,
    $param as item()*)
    as element(soapenv:Envelope) {
 if ($nmsname="http://example.net"
      and $funname="getContact")
 then
 <soapenv:Envelope ..>
  ...
  {gup:getContact($param/p1/node(),
                  $param/p2/node())}
  ...
 </soapenv:Envelope>
 else if  ...
};
```

A SOAP client – such as one generated with the XQuery import service statement – may send a request to the Web server containing the message:

```
POST /services/gup.xqs HTTP/1.1
Host: example.net
...
SOAPAction: ...

<?xml version="1.0" encoding="utf-8"?>
<soapenv:Envelope ...>
  <soapenv:Body>
    <getContact xmlns="http://example.net">
      <ownername>jsimeon</ownername>
      <contactid>4</contactid>
    </getContact>
  </soapenv:Body>
</soapenv:Envelope>
```

The associated XQuery module extracts the XML content and the meaningful HTTP headers and, using the Galax C interface, calls the `make_envelope` function in the generated stub:

```
local:make_envelope (
  "getContact",
  "http://example.net",
  <param>
   <p1>jsimeon</p1> <p2>4</p2>
  </param>
)
```

This, in turn, calls `gup:getContact` as shown before and produces a SOAP Envelope which is sent to the client in a HTTP response:

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

 <soapenv:Envelope ...>
   <soapenv:Body>
     <tns:contact>
       <name>...</name>
       <tel>...</tel>
     </tns:contact>
   </soapenv:Body>
 </soapenv:Envelope>
```

Again, the generated stub is compact because most of the data processing is done directly in XML, and XQuery itself is used to manipulate the SOAP envelopes.

Currently, the `xquery2soap` utility in Galax is capable of producing an XQuery server skeleton for the XQuery apache module through the use of the apache API, which itself calls the Galax C API[6]. The automatic WSDL generator is still under implementation.

5. THE WSDL-XQUERY BINDING

In all our previous examples, the WSDL `portType` describes an operation with the right name and signature for the XQuery module given above. In such a case, we can use the XQuery module as an *implementation* for the service, as was described in Section 4. Obviously, not all pairs of XQuery modules and WSDL portTypes can be bound to one another. In this section, we provide a complete description of the WSDL-XQuery binding which underlies both the import service statement and the `xquery2soap` tool.

[6]The corresponding C API is only available in Galax version 0.3.1 and later.

5.1 Binding Definition

The binding between WSDL and XQuery relies on the close relationship between WSDL `portType`s and XQuery modules, which we explained informally in Section 2. We define the XQuery-WSDL binding by giving a mapping between the constructs of WSDL and XQuery statements. The mapping uses standard "normalization rules" notations (see [31, 14] for an introduction to those notations).

- A WSDL `portType` is mapped to an XQuery module:

  $$\texttt{[<portType>}Operations\texttt{</portType>]}_{\text{Bind}} == \texttt{[}Operations\texttt{]}_{\text{Bind}}$$

- A WSDL `operation` is mapped to an XQuery function:

  $$\texttt{[<operation name="QName">}\ Input\ Output\ \texttt{</operation>]}_{\text{Bind}} == \texttt{define function QName([}Input\texttt{]}_{\text{Input}}\texttt{)}\ \texttt{[}Output\texttt{]}_{\text{Output}}\texttt{external}$$

- A WSDL `input` message is mapped by mapping each of its `part`s to an XQuery variable declaration:

  $$\texttt{[<message>}Parts\texttt{</message>]}_{\text{Input}} == \texttt{[}Parts\texttt{]}_{\text{Input}}$$

  $$\texttt{[<part name="QName"}\ Type\texttt{/>]}_{\text{Input}} == \texttt{\$QName as [}Type\texttt{]}_{\text{Type}}$$

- A WSDL `ouput` message is mapped only if it has a single `part`, which is mapped to an XQuery sequence type:

  $$\texttt{[<message>}Part\texttt{</message>]}_{\text{Output}} == \texttt{[}Parts\texttt{]}_{\text{Output}}$$

  $$\texttt{[<part name="QName"}\ Type\texttt{/>]}_{\text{Output}} == \texttt{[}Type\texttt{]}_{\text{Type}}$$

- A WSDL part type is mapped depending on the method used to identify the type, into an XQuery sequence type:

  $$\texttt{[element="QName"]}_{\text{Type}} == \texttt{element(QName)}$$

  $$\texttt{[type="AtomicType"]}_{\text{Type}} == \texttt{AtomicType}$$

  $$\texttt{[type="ListType"]}_{\text{Type}} == \texttt{AtomicType*}$$

  Where $AtomicType$ is the atomic type from which the list type is derived.

  $$\texttt{[type="ComplexType"]}_{\text{Type}} == \texttt{element(*,ComplexType)}$$

In essence, the binding maps WSDL operations to XQuery function signatures. There are three important remarks on that mapping.

- One limitation of the binding is that output messages for operations must have exactly one part. This limitation is imposed to accommodate the fact that XQuery function must have exactly one return value. This is not a strong limitation in practice as a user may always include the various parts within an XML elements if necessary. For instance, the following WSDL operation cannot be mapped to XQuery.

```
<message name="getContactResponse">
  <part name="name" type="xs:string"/>
  <part name="tel" type="xs:string"/>
</message>

 <operation name="getContact">
  <input message="tns:getContact"/>
  <output message="tns:getContactResponse"/>
 </operation>
```

- The mapping does not bind XQuery global variables as there is nothing equivalent at the WSDL level. For instance, in the following XQuery module, the global variable cannot be made accessible as a service operation.

```
module "http://myexample.com";
declare variable $const { 1 };
```

- The mapping is precise for XML Schema atomic types, list types, and element types. However, it handle complex types and union types only if enclosed an element which must be generated by the system. For instance, the following operation

```
<types>
 <xs:schema
  targetNamespace="http://example.net">
  <xs:complexType name="Contact">
    <xs:sequence>
      <xs:element name="name" type="xs:string"/>
      <xs:element name="tel" type="xs:string"/>
    </xs:sequence>
  </xs:complexType>
 </xs:schema>
</types>

<message name="getContactResponse">
  <part name="result" type="Contact"/>
</message>

 <operation name="getContact">
  <input message="tns:getContact"/>
  <output message="tns:getContactResponse"/>
 </operation>
```

can only be mapped to an XQuery function which uses an element around the result information, as in:

```
declare function gup:getContact (
    $ownername as xs:string,
    $contactid as xs:integer
  ) as element(result,Contact) ...
```

This forces the user to pay special attention to those cases when using the service import and Web services deployment tool. We will propose some suggestions to avoid those specific problems in 6.3.

5.2 Semantics Constraints

The proposed binding can be used in several ways:

- To generate automatically an XQuery module out of given WSDL `portType`. This is the direction used when importing a service in XQuery (See Section 3).
- To generate automatically a WSDL port type out of a given XQuery module. This is the direction used when deploying a Web service from an XQuery module (See Section 4).
- The binding can be used to check that a given module and a given WSDL port type are consistent with each other. This

is used in the case where the `xquery2soap` tool uses the optional `-wsdl` flag, or if one wants to use WSDL as an interface language for XQuery (See Section 6.1).

In the latter case, it is necessary to check semantic constraints for the WSDL and XQuery binding to be valid. We propose to use semantic constraints which are similar to those which apply to module interfaces in standard functional programming languages [16, 6, 15]. A WSDL `portType` can be validly bound to an XQuery module, iff:

- The target namespace of the module corresponds to the target namespace of the WSDL files concatenated with the name of the service and the name of the port.
- Each WSDL `operation` has exactly one implementation as an XQuery function in that module.
- For each WSDL `operation`, there is the same number of input `part`s as the number of parameters in the function it is bound to. Each part corresponds to a parameter with the same name and appearing in the same order, and the type of the part is a subtype of the type for the corresponding function parameter.
- For each WSDL `operation`, there is exactly one output `part`. The type for that part must be a super type for the corresponding output type of the function.

As we will see in Section 6.1, those constraints not only make sure that the binding is valid, but also that it can be used to support some form of hiding during module import.

6. BENEFITS AND IMPROVEMENTS

6.1 WSDL as a Module Interface

As we have seen in the previous section, the binding requires that all WSDL operations have a corresponding function in the XQuery module it is bound to, but not the other way around. As a result, there might be some XQuery functions which are not exported through the service, and are therefore left local to the module itself. Notably this may be applied to hide part of the implementation of that module to other modules or applications.

For instance, consider the following alternative implementation for the user profile module, which uses the additional local function `local:getFullContact`.

```
declare function local:getFullContact( (
    $ownername as xs:string,
    $contactid as xs:integer
  ) as element(fullcontact) {
  let $prof := //profile[@owner = $ownername]
  return
    $prof//fullcontact[@id = $contactid]
};

declare function gup:getContact (
    $ownername as xs:string,
    $contactid as xs:integer
  ) as element(contact) {
  let $c :=
    local:getFullContact($ownername,$contactid)
  return
   <contact>
     <name>{ concat($c/first,$c/last) }</name>
     <tel>{ $c/phone[1] }</tel>
   </contact>
};
```

Now assuming we export that module using the *same* WSDL file as before, then only the registered operation `gup:getContact` will be accessible to other modules, leaving the local function hidden. This approach of hiding can bring numerous benefits for large scale application development, most notably one can change the implementation for a module without recompiling the other modules as long as the interface for that module is left unchanged. The problems and benefits of module systems has been a topic of interest in the programming language field for many years [16, 6, 15].

6.2 Static Typing on Web Service Composition

XQuery is a statically typed language [30, 31, 14]. When using static typing, the XQuery processor may check that operations are applied on the right type(s) and detect errors at compile time. Because of the tight-coupling between WSDL and XQuery, static typing on XQuery programs with service calls come for free. For instance, let us consider again the small program given in Section 3 which imports the user profile service and the Google search service. Let us assume for a moment that the user mistakingly inverted the parameters when calling the `gup` service, as in:

```
import service
 namespace gup = "http://example.net"
  name "UserProfile";

let $c := gup:getContact(4,"jsimeon")
return google:doGoogleSearch($c/tel,...)
```

If it supports static typing, the XQuery processor can detect such errors at compile time, before the service is effectively called. This feature, although maybe not very useful for such small programs, is most valuable when Web services application become large or when some changes to the underlying Web services require some maintenance to the program.

6.3 Suggestions for WSDL and XQuery

In Section 5, we identified a few specific mismatches between WSDL and XQuery modules. We finish the technical presentation with a few suggestions about how to deal with those problems.

Multiple output parts. We suggest to restrict WSDL output messages to contain only a single part, making those consistent with the functional approach used in XQuery.

Complex types in WSDL. Sequence types in XQuery do not provide all the functionalities required to describe parameters from operations in WSDL. We suggest to extend XQuery sequence type to align it with WSDL's type descriptions. Conversely, one might extend the WSDL types to support the equivalent of minOccurs, maxOccurs.

Global variables. It might be convenient to be able to support global variables through a service. Currently a user would have to wrap the content of that variable as a function. We suggest to extend WSDL with a notion of 'constant' value, which could be retrieved through a service call.

We believe those specific changes could further improve the coupling between the two technologies, without significantly changing the spirit of any of the corresponding specifications.

7. CONCLUSION AND FUTURE WORK

In this paper we have presented techniques and tools to allow the use of XQuery as an integral part of the Web services development infrastructure. Our work relies on the notion of modules

recently added to XQuery. We introduced a notion of binding between XQuery and WSDL which can be used for both Web services deployment and composition. The approach has the advantage of requiring only limited changes to both XQuery and WSDL, and to facilitate Web services development.

We believe this work is a first step that immediately brings the benefits of native XML processing to Web developers, but also opens several new research opportunities. For instance, this provides an interesting framework for distributed XQuery processing, or information integration. Also we plan to investigate several new extensions to XQuery in order to make it even more useful in the context of Web services development.

Acknowledgments. We want to address our thanks to the GUPster team at Bell Labs for interesting discussions on Web services, and to Sihem Amer-Yahia and Mary Fernández for helpful comments on previous drafts of the paper.

8. REFERENCES

[1] 3rd Generation Partnership Project. 3GPP generic user profile - architecture. Technical report, Technical Specification Group Services and System Aspects, 2003.

[2] S. Abiteboul, O. Benjelloun, I. Manolescu, T. Milo, and R. Weber. Active xml: Peer-to-peer data and web services integration. In *Proceedings of International Conference on Very Large Databases (VLDB)*, pages 1087–1090, Hong Kong, China, Aug. 2002.

[3] Axis user's guide. `http://ws.apache.org/axis/`.

[4] V. Benzaken, G. Castagna, and A. Frisch. CDuce: An XML-centric general-purpose language. In *Proceedings of the ACM International Conference on Functional Programming*, Uppsala, Sweden, Aug. 2003.

[5] Castor. `http://castor.exolab.org/`.

[6] P. Curtis and J. Rauen. A module system for Scheme. In *Proceedings of the ACM conference on LISP and Functional Programming*, pages 13–19, Nice, France, 1990.

[7] M. Fernandez, J. Siméon, B. Choi, A. Marian, and G. Sur. Implementing XQuery 1.0: The Galax experience. In *Proceedings of International Conference on Very Large Databases (VLDB)*, pages 1077–1080, Berlin, Germany, Sept. 2003.

[8] D. Florescu, A. Grünhagen, and D. Kossmann. XL: A platform for Web services. In *Conference on Innovative Data Systems Research (CIDR)*, Asilomar, CA, Jan. 2003.

[9] D. Florescu, A. Grünhagen, and D. Kossmann. XL: An XML programming language for Web service specification and composition. In *Proceedings of International World Wide Web Conference*, pages 65–76, May 2002.

[10] V. Gapeyev and B. C. Pierce. Regular object types. In *Proceedings of the 17th European Conference on Object-Oriented Programming (ECOOP'2003)*, pages 151–175, Darmstadt, Germany, July 2003.

[11] H. Hosoya and B. C. Pierce. XDuce: an XML processing language. In *International Workshop on the Web and Databases (WebDB'2000)*, Dallas, Texas, May 2000.

[12] P. B. James McGovern, K. Cagle, J. Linn, and V. Nagarajan. *XQuery Kick Start*. SAMS, 2003.

[13] Java architecture for XML binding (JAXB). `http://java.sun.com/xml/jaxb/`.

[14] H. Katz, editor. *XQuery from the Experts: A Guide to the W3C XML Query Language*. Addison-Wesley, 2003.

[15] X. Leroy. Manifest types, modules, and separate compilation. In *Proceedings of ACM Symposium on Principles of Programming Languages (POPL)*, pages 109–122, Portland, Oregon, Jan. 1994.

[16] D. MacQueen. An implementation of standard ML modules. In *Proceedings of the ACM Conference on LISP and Functional Programming*, pages 212–223, Snowbird, Utah, 1988.

[17] M. Maloney and A. Malhotra. XML schema part 2: Datatypes. W3C Recommendation, May 2001.

[18] T. Milo, S. Abiteboul, B. Amann, O. Benjelloun, and F. D. Ngoc. Exchanging intensional xml data. In *Proceedings of ACM Conference on Management of Data (SIGMOD)*, pages 289–300, San Diego, California, June 2003.

[19] Ojxqi - the oracle java xquery api. `http://otn.oracle.com/sample_code/tech/xml/xmldb/jxqi.html`.

[20] SOAP version 1.2 part 0: Messaging framework. W3C Recommendation, June 2003.

[21] H. S. Thompson, D. Beech, M. Maloney, and N. Mendelsohn. XML schema part 1: Structures. W3C Recommendation, May 2001.

[22] WebL. compaq's Web language. `http://www.research.compaq.com/SRC/WebL`.

[23] WSCI. Web service choreography interface. `http://wwws.sun.com/software/xml/developers/wsci`, 2002.

[24] WSCL. Web services conversation language. `http://www.w3.org/TR/wscl10`, Mar. 2002.

[25] Web services description language (wsdl) 1.1. W3C Note, Mar. 2001.

[26] Web services description language (WSDL) version 2.0 part 1: Core. W3C Working Draft, Mar. 2003.

[27] WSFL. Web services flow language. `http://www.ibm.com/software/solutions/webservices/pdf/WSFL.pdf`.

[28] XPath 2.0. W3C Working Draft, Nov. 2003.

[29] XQuery API for Java (XQJ). `http://jcp.org/en/jsr/detail?id=225`.

[30] XQuery 1.0: An XML query language. W3C Working Draft, Nov. 2003.

[31] XQuery 1.0 and XPath 2.0 formal semantics. W3C Working Draft, Feb. 2004.