STUDIA UNIV. BABEŞ–BOLYAI, INFORMATICA, Volume **LVIII**, Number 3, 2013

WEB SERVICE MATCHING

FLORIN M. BOIAN[(1)], ADINA PLOSCAR[(1)], AND RAREŞ F. BOIAN[(1)]

Abstract. The paper addresses the heterogeneity of implementing web services of different types on various platforms, and the need for a way to determine whether two web services are matching. Starting with an analysis of the implementation differences and theoretical base for determining matching automatically, the paper proposes a novel architecture. The architecture offers a unified way of designing and creating web services, which eliminates the existing differences plaguing the available frameworks. It also offers support for automatic formal detection of web service matching.

1. Introduction

Web pages were, until recently, the main form in which data was offered on the World Wide Web for human consumption. Therefore the Web is mostly used for browsing using a Web browser to read news/articles, to buy goods and services, to manage on-line accounts and so on.

From a publishing perspective, this is realized by transforming the information from a database, for example, into HTML or similar language so that it can be rendered in a user readable format. Many Web sites put together information extracted by other sites via Web pages, which is an inconsistent process involving decoding and parsing human-readable information not intended for machine processing at all.

Received by the editors: April 17, 2013.

2010 *Mathematics Subject Classification.* 68N25, 68U35.

1998 *CR Categories and Descriptors.* C.2.4 [**Computer Systems Organization**]: Computer-Communication Networks – *Distributed systems*; D.2.12 [**Software**]: Software Engineering – *Interoperability*; H.3.5 [**Information Systems**]: Information Storage and Retrieval – *On-line Information Services*; I.2.2 [**Computing Methodologies**]: Artificial Intelligence – *Automatic Programming.*

Key words and phrases. web services, client, matching, XML-RPC, SOAP, REST, XRDL, WSDL, WADL.

This paper has been presented at the International Conference KEPT2013: Knowledge Engineering Principles and Techniques, organized by Babeş-Bolyai University, Cluj-Napoca, July 5-7 2013.

This scenario works well in many cases but it does not support software interactions very well. What is really needed is a mechanism through which the raw data from a database can be accessed in a similar fashion by machines as humans read Web pages now. To reach this goal we need a specialized client that knows how to perform true machine-to-machine communication thus creating a Web that is machine process-able.

In the last 15 years, three models of web services were studied:

- XML-RPC model
- SOAP (+ WSDL and UDDI) model
- REST (RESTfully) model

For all three models, there are many implementations, open source as well as commercial. In this context, four languages came to our attention: Java, C♯, Python and PHP.

2. The Purpose of XRDL, WSDL, WADL

In [4] we presented a method for generating uniform web services and clients. We discussed in that paper the WSWrapper package and how it has as its central component the descriptor generator: XRDL - for XML-RPC services, WSDL for SOAP services, and WADL for REST services. We also proposed there a method for generating client proxies from those descriptors.

A major problem related to Web Services is matching them. There are many web services available on the Internet and APIs offering functionalities that are identical, similar or very close to each other. This leads to a natural attempt to automatically detect such similar services. Studies on the similarity problem are practically impossible to perform without formal descriptors of the services.

Khorasgani et. al. studies in [9] the matching problem for RESTful services. They propose the SFM (Semantic Flow Matching) method for studying the equivalence of two services. Unfortunately, the results are far from being applicable. Their analysis is practically impossible without the use of a WADL.

The WADL[19] purpose is to create a contract between the method exporter (i.e. the service) and the calling clients. However, such a contract is not mandatory [14]. WADL is not a standard and it is not maintained since 2009 but is important in the matching problem. If the service is to be integrated in a complex system, extremely precise communication contracts are mandatory, which turn the WADL in a necessary annoyance.

The matching analysis for SOAP web services is a very complex problem because equivalent services can generate a wide variety of WSDL descriptors, with significant differences between them.

XML-RPC web services are much simpler than SOAP, yet the matching analysis is still very complex. The lack of a formal descriptor makes the matching analysis inapproachable. For the time being, the only proposed descriptor format is XRDL. Unfortunately, the adoption and pro/cons opinions on this format are divided evenly among experts [16].

Our experience in Web service matching is the proposal in [2] of twelve equivalent services: four XML-RPC services, four SOAP services, and four REST services. Every such group of services was implemented in C♯, Java, PHP, and Python. For every group and every language we implemented clients for accessing the services. In [5] we published an extended version of this work that includes Android clients as well.

All web services are identified with their URLs. These have the format: `http://host:port/path` from which `port` and/or `path` may be missing. Working with web services, one quickly realizes that there is a wide variety of constraints and restrictions regarding these URLs. In some cases the `http://` must be present, while in other it doesnt need to be there. In some situations `port` is mandatory, while in others it is not required or even forbidden. In some cases `path` must end with /, in other cases / is forbidden, and so on. Basically, every web service distribution comes with its own conventions.

We will detail this analysis in the next sections.

3. XML-RPC Case Study

The XML-RPC web service used in [2, 5] as a practical example for our work was implemented in every of the four programming languages listed above. The figure 1 shows the XRDL descriptor of this service.

```
<?xml version="1.0" encoding="UTF-8"?>
<service name="Exec" ns="ro.ubbcluj.cs.Exec"
         url="www.scs.ubbcluj.ro:1001">
  <methods>
    <method name="ping" result="string"/>
    <method name="upcase" result="string">
      <param type="string">s</param>
    </method>
    <method name="add" result="i4">
      <param type="i4">a</param>
      <param type="i4">b</param>
    </method>
  </methods>
</service>
```

Figure 1. XRDL descriptor for the Exec web service

The four implementations use very similar XRDLs as the one above, differing only in the values of attributes name, ns, and url of the service element.

The four implementations are:

- C♯ using **XmlRpcCS-1.2**
- Java using **apache-xmlrpc-3.1.3**
- PHP using **xmlrpc-2.2.2** deployed on a web server running a PHP 5 engine
- Python using **xmlrpclib** which is packaged by default in Python 2.7

The localization of these services is relatively consistent. The PHP service is identified by the host of the web service, while the other three create their own web server which listens on the port specified in the URL. The Python implementation requires the specification of both a port and a host address.

The registration of the exported methods is done differently in the implementations. C♯ and Java simply provide the object which implements the methods along with a name under which the clients see this object. In PHP and Python each method is registered along with an associated name to be used by the client.

The PHP implementation is done at the lowest level of all the four languages. Here one needs to specify both on the client and the server the encoding and decoding functions needed for converting the parameters and the messages exchanged between the client and the service.

The clients were implemented on five platforms: C♯, Java, PHP, Python, and Android. Each client gets the URL of the web service which includes the name with which the client identifies it (the name under which the methods are exported). The method calls are then done consistently and without issues.

Although these web services were implemented as equivalent, there is no practical way to automatically prove their equivalence. Consequently, it is necessary to automatically generate the XRDL descriptors which then can be compared in order to prove that the web services match.

4. SOAP Case Study

In the work presented in [5] we used a single web service implemented in multiple programing languages. The methods exported by the web service are the same as those presented in the previous section. The descriptions are similar with those in fig. 1 using the WSDL standard, but instead of `param s` and `param a` we use `param arg0`, and instead of `param b` we use `param arg1`. In the source code of the three web services we changed the definition of method `upcase` to use parameter `arg0` instead of `s`, and in method `add` the two parameters are named `arg0` and `arg1` instead of `a` and `b`. This was necessary due to constraints imposed by some of the clients calling the web

service. The C♯, Python and Android clients, and some PHP clients access the parameters using their *name* while some Java and PHP clients use implicitly declared names such as: *arg0*, *arg1*, *arg2* , and so on. The four services we implemented are:

- C♯ using a file of type ***.asmx** on an IIS web server[2].
- Java using apache-cxf-2.2.7 of type **JAX-WS2**.
- PHP using **nusoap** 0.9.5 with the web service deployed on a PHP 5 engine.
- Python using **spyne**. In [2] we used **jkp-soaplib** which could not be accessed from C♯ and Java.

The localization of these services is relatively consistent. The C♯ and PHP web services are identified by their host web servers. Java publishes an object Endpoint which is configured with the service URL and a reference to the object which exports the methods. PHP and Python require the developer to specify the `targetNamespace` parameter and also an application name.

The method registration is done differently among the implementations. In C♯ and Java the service object is annotated as `WebService`, and the exported methods are annotated as `WebMethod`. The web service Python extends the standard `spyne.service.ServiceBase` object, and the methods are prefixed with the `@rpc` decorator. PHP requires that each method be registered individually along with its prototype and namespace.

The client needs to be treated independently for each service to be accessed.

For each C♯ or Java client that accesses a web service we must generate the source code of a static proxy that accesses the web service. Thus there are four C♯ clients and four Java clients created for every of the web services involved in our analysis. The proxies are generated in a consistent manner. In C♯ this is done using the WSDL utility on the web service URL. The utility generates the C♯ code of the client and compiles it into a DLL which is integrated in the client.

The client sources differ usually only through the line that defines the proxy object named by us service. The definition of these proxies for each web service is given below:

- `Ss1Serv service = new Ss1Serv(); // for C♯`
- `Sj1ServService service = new Sj1ServService(); // for Java`
- `Sh1Serv service = new Sh1Serv(); // for PHP`
- `Sy1Serv service = new Sy1Serv(); // for Python`

This proves that our proposal in [1, 4] to use a single `WebServiceClient` object for any platform is feasible.

The Java clients are of two types: JAX-WS for the C♯, Java and Python web services; and AXIS1 for PHP. This is because `nusoap` does not support (yet) JAX-WS. The JAX-WS proxies are generated using the `wsimport` utility based on the web service URL. The generated sources are compiled and built into a JAR which is then integrated in the client. For PHP, the proxy generation is done using the `Wsdl2Java` utility. As for C♯, the clients differ only in the proxy definition line:

- `Ss1ServSoap service =`
  `(new Ss1Serv()).getSs1ServSoap(); //for C♯`
- `Sjj1Proxy.Sj1Serv service =`
  `(new Sj1ServService()).getSj1ServPort(); //for Java`
- `Sh1ServPortType service =`
  `(new Sh1ServLocator()).getSh1ServPort(); //for PHP`
- `Application service =`
  `(new Sy1Serv()).getApplication();// for Python`

These similarities reinforce the feasibility of our [4] proposal for using a unified client object for any platform.

There are two types of PHP clients. The **nusoap** clients can only access **nusoap** web services. The PHP 5 distribution supports the creation of SOAP clients using the PHP SOAP module which is part of the core PHP distribution. This library is good enough for building clients, but not sufficiently strong yet for building web service. The proxy definition lines are:

- `$service = new nusoap_client($urlServ . "?wsdl", true);`
  `//for PHP nusoap`
- `$service = new SoapClient($urlServ . "?wsdl");`
  `//for any service`

The SOAP PHP clients calling services other than PHP **nusoap** require a special parameters transmission approach: the developer must create a PHP object containing the parameters as properties and these are then accessed either by name or by value.

The SOAP Python clients are the simplest of all. The suds project is a recently released Python distribution specialized on web service clients. The suds library creates dynamically a proxy object starting from the WSDL, without generating any source code. This aspect simplifies significantly the client implementation.

For writing Android web service clients, the only library available is ksoap2. Essentially the client requires the WSDL and gets from there the `targetNamespace` parameter. If this parameter is not specified, then the client goes directly to the web service, gets the WSSDL from there and extracts the `targetNamespace` from it. This peculiar behavior is necessary because there is a problem in

the connection to the Python web service. The connection fails if the client fetches the WSDL from the web service, extracts the `targetNamespace` parameter and then starts calling methods. However everything works fine if the `targetNamespace` is specified directly to the client. The explanation is that the Python web service enforces stateless access, which means that a WSDL delivery closes the connection.

5. REST Case Study

The service used in [2, 5] as example implemented in several programming languages is used here as well. The fig. 2 presents WADL for describing this service.

The four web services implement the same functionalities as those in the previous sections. To support a wider range of examples we choose to make the `upcase` method accessible through both GET and POST calls, and the `add` method through both PUT and DELETE calls. The implemented web services are:

- C♯ using a ***.ashx** file deployed in an IIS web server.
- Java using the JAX-RS **jersey**[17] library deployed in a servlet container.
- PHP using the **Da Silva distribution** [6] which offers two objects: `RestServer.php` and `RestClient.php` which support method mapping using URL regular expressions.
- Python using the **CherryPy** [18] library.

The localization of these services is variable, depending on the method used to pass the parameters: as a query string at the end of a GET method, in the body of a POST or DELETE method, or as variables in the URL path.

The method registration is done differently among the web service implementations. In C♯ the method is given control upon the completion of an analysis of the path in the `HttpContext` URL. In Java the methods to be exported are annotated as such. In PHP the mapping of each method is specified in the `RestServer` object. In Python the methods are annotated using the `@cherrypy.expose` decorator.

The clients were implemented in: C♯, Java, PHP, Python, and Android. Each of them gets the service URL and then the service is accessed simply over URL connections: `HttpWebRequest` and `HttpWebResponse` for C♯, `HttpUrlConnection` in Java, using CURL and possibly the Da Silva `RestClient` for PHP, and the standard `urllib` and `httplib` modules of Python.

Although these services were built to be equivalent, there is no practical way to demonstrate their equivalence automatically. To do so, we need to generate the WADL descriptors and compare them to prove the equivalence.

```
<?xml version="1.0"?>
<application xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://wadl.dev.java.net/2009/02 wadl.xsd"
   xmlns:tns="urn:yahoo:yn" xmlns:xsd="http://www.w3.org/2001/XMLSchema"
   xmlns:yn="urn:yahoo:yn" xmlns:ya="urn:yahoo:api"
   xmlns="http://wadl.dev.java.net/2009/02">

   <resources base="http://www.scs.ubbcluj.ro/Exec">
      <resource path="Rj1Serv/ping">
         <method name="GET" id="ping">
            <request />
            <response status="200">
               <representation mediaType="application/xml"
                  element="xsd:string" />
            </response>
         </method>
      </resource>
      <resource path="Rj1Serv/upcase">
         <method name="GET" id="upcaseQ">
            <param name="s" type="xsd:string" style="query"
               required="true" />
            <response status="200">
               <representation mediaType="application/xml"
                  element="xsd:string" />
            </response>
         </method>
      </resource>
      <resource path="Rj1Serv/upcase/{s}">
         <method name="POST" id="upcaseP">
            <param name="s" type="xsd:string" style="path"
               required="true" />
            <response status="200">
               <representation mediaType="application/xml"
                  element="xsd:string" />
            </response>
         </method>
      </resource>
      <resource path="Rj1Serv/add">
         <method name="PUT" id="addB">
            <param name="a" type="xsd:int" style="body"
               required="true" />
            <param name="a" type="xsd:int" style="body"
               required="true" />
            <response status="200">
               <representation mediaType="application/xml"
                  element="xsd:int" />
            </response>
         </method>
      </resource>
      <resource path="Rj1Serv/add/{a}/{b}">
         <method name="DELETE" id="addP">
            <param name="a" type="xsd:int" style="body"
               required="true" />
            <param name="a" type="xsd:int" style="body"
               required="true" />
            <response status="200">
               <representation mediaType="application/xml"
                  element="xsd:int" />
            </response>
         </method>
      </resource>
   </resources>
</application>
```

FIGURE 2. WADL descriptor of the Exec web service

6. Our Proposal

Considering the aspects presented above we propose to extend the WSWrapper published in [1], as shown in the fig. 3.

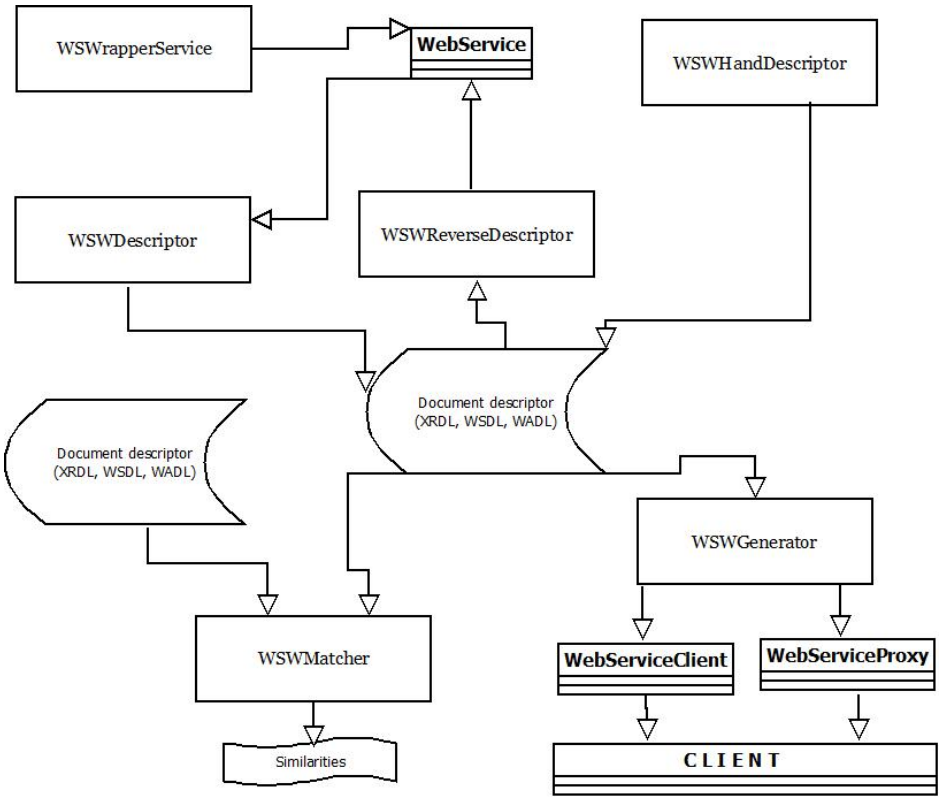

Figure 3. Extended WSWrapper architecture

The main components of this new architecture are:

WSWrapperService processes the **WebDescriptorService** descriptors and generates the actual web service for the desired platform.

WSWDescriptor processes WebService objects using reflection (inspection, introspection) and, depending on the web service type, generates a descriptor of type XRDL, WSDL, or WADL. This is possible because all the languages involved support reflection, and specifically introspection, which is essential to the extraction of meta-data, exported methods, and calling paths, from the web service objects. [12, 13, 15].

WSWHandDescriptor is an auxiliary manual tool which allows the designer to customize the generated descriptor as necessary.

WSWReverseDescriptor generates source code from a given descriptor of any of the three supported types: XRDL, WSDL, or WADL. The generated source code will contain the method prototypes for the web service on a desired platform.

WSWGenerator [8] generates a static proxy based on a web service descriptor. The proxy will be specific to the web service it was generated for and to the platform being used. The proxy is then packaged in a library (i.e. **dll**, **jar**, **phar**, **zip**) which will become part of the client.

WSWMatcher takes as input two web service descriptors and detects the similarities between them, trying to determine whether they match or not.

As web services of different types have different descriptions and even various WSDL files may eventually have (and often do) different descriptions for equivalent services we have the following proposition. We describe each service regardless of the type of the web service in an abstract format that we will call WSAD (Web Service Description Abstract). This WSAD will describe only web method name, parameters whose types must comply with XML types and return type of the method with the same restrictions as for parameters. Because it is an abstract description of the service, the location of the web service is not needed.

WSWMatcher takes as input two WSAD files describing two web services and will compare them syntactically. The matching that we propose is at a syntactic level, and at this point in our research we do not consider the semantic. If two services match in a syntactically way we can continue their matching semantically.

7. Conclusions

The paper presents a detailed analysis of the differences between web service implementations on various platforms and the theoretical possibilities for automatically determining whether web services are matching. The proposed architecture unifies web service development and eliminates the differences which get in the way when using the existing frameworks. It also provides support for automatically detecting web service matching.

References

[1] Boian F., Chinces D., Ciupeiu D., Homorodean D., Jancso B., Ploscar A., *WSWrapper - A Universal Web Service Generator*, Studia Universitatis Babes-Bolyai Series Informatica, Volum LV, nr. 4, 2010, pp 59-69, ISSN: 2065-9601

[2] Boian F.M., *Servicii web; modele, platforme, aplicaii*, Ed. Albastra, Cluj, 2011

[3] Boian F.M., *A uniform approach to define and implement the web services; case studies for indexing huge file systems*, ZAC2012, pp 85-90

[4] Boian F.M., Jancso B., *Uniform solutions for web services*, Studia Universitatis Babes-Bolyai Series Informatica, Volum LVII, nr. 3, 2012, pp 13-23
[5] Boian F.M., http://www.cs.ubbcluj.ro/ florin/books/SWMPA/*New.zip, 2013
[6] DaSilva S.D., *PHP Classes*, http://diogok.users.phpclasses.org/browse/author/529977.html
[7] Jancso B., *RESTful Web Services*, ZAC2010, pp. 158-163
[8] Jancso B., *Web Service proxy Generator*, ZAC2012, pp 124-129
[9] Khorasgani R.R., Stroulia E., Zaiane O.R., *Web service Matching for RESTful Web Services*, http://webdocs.cs.ualberta.ca/ zaiane/postscript/wse2011.pdf
[10] Ploscar A., *A Java Implementation for REST-style web service*, ZAC2010, pp 140-146
[11] Takase T. Makino S. Kawanaka S. Ueno C.F. Ryman A. *Definition Languages for RESTful Web Services: WADL vs. WSDL 2.0.*, http://www.ibm.com/developerworks/library/specification/ws-wadlwsdl/index.html
[12] * * * http://scripts.incutio.com/xmlrpc/introspection.html
[13] * * * http://xmlrpc-c.sourceforge.net/introspection.html
[14] * * * http://stackoverflow.com/questions/1312087/what-is-the-reason-for-using-wadl/1314357♯1314357
[15] * * * http://www.codeproject.com/Articles/235269/Using-Introspection-in-Java, C♯ reflection Java reflection PHP reflection Python inspect
[16] * * * XRDL: XML-RPC Description Language. http://code.google.com/p/xrdl/
[17] * * * http://jersey.java.net/
[18] * * * http://www.cherrypy.org/
[19] * * * http://www.w3.org/Submission/wadl/

(1) Babeş-Bolyai University, Department of Computer Science, Cluj-Napoca, Romania
E-mail address: florin@cs.ubbcluj.ro
E-mail address: adina_ploscar@yahoo.com
E-mail address: rares@cs.ubbcluj.ro