Improving the Predictable Assembly of Service-Oriented Architectures

Brian A. Malloy, Nicholas A. Kraft, Jason O. Hallstrom
Computer Science Department
Clemson University
Clemson, SC 29634, USA
{malloy,nkraft,jasonoh}@cs.clemson.edu

Jeffrey M. Voas
Director, Systems Assurance
Science Applications International Corporation
Arlington, Virginia
jeffrey.m.voas@saic.com

Abstract

In this paper, we propose an intermediate approach to web service specification. In particular, our approach balances the ease of expressivity offered by informal documentation with the mathematical rigor provided by traditional formalisms. We present a technique that integrates the use of regular expressions in WSDL specifications to constrain the format of input and output values to and from web services. This approach provides the basis for automating the generation of both client- and server-side checking wrappers.

1. Introduction

The development and wide-adoption of object technology has improved the modularity, extensibility, and reusability of software applications. The object-oriented approach is especially successful in achieving a high degree of reuse, with the development of countless numbers of highly accessible libraries and frameworks and their concomitant availability. Some popular examples of this reuse include graphical user interface libraries such as Microsoft Foundation Classes and Java Swing, as well as container and algorithm classes such as the C++ standard library. This reuse is complemented by the development and growing popularity of *design patterns*, providing a common vocabulary for object-oriented software design and development [2].

However, reusing object libraries and frameworks requires skill in object-oriented programming and an underlying knowledge of the classes, as well as the services that they provide. An approach that is complementary to object-oriented software reuse entails the use of web services and service oriented architectures (SOAs) [1]. In fact, the popularity of web services will almost certainly expand as we move closer to realizing the ubiquitous computing vision, suggesting that the future of software will involve the ever expanding use of SOAs. The popularity of standards such as XML, WSDL, SOAP and UDDI facilitate the development and accessibility of web services, providing the foundation and tool set to build federated distributed applications.

However, the specifications for web services are typically informal and not well-defined. The resulting ambiguity can lead to improper use of web services, errors and a lack of knowledge of the functionality being provided. There are two commonly used approaches to web service specification. In the first approach, comments are embedded in the WSDL specification; these comments are generally vague and difficult or impossible to verify automatically. In the second approach, a more rigorous specification is mandated through the use of Turing-complete specification languages, such as BEPL4WS or WSCI, or through the use of algebraic languages [3]. However, these Turing-complete languages provide more implementation than specification and algebraic specifications can be intimidating to practitioners.
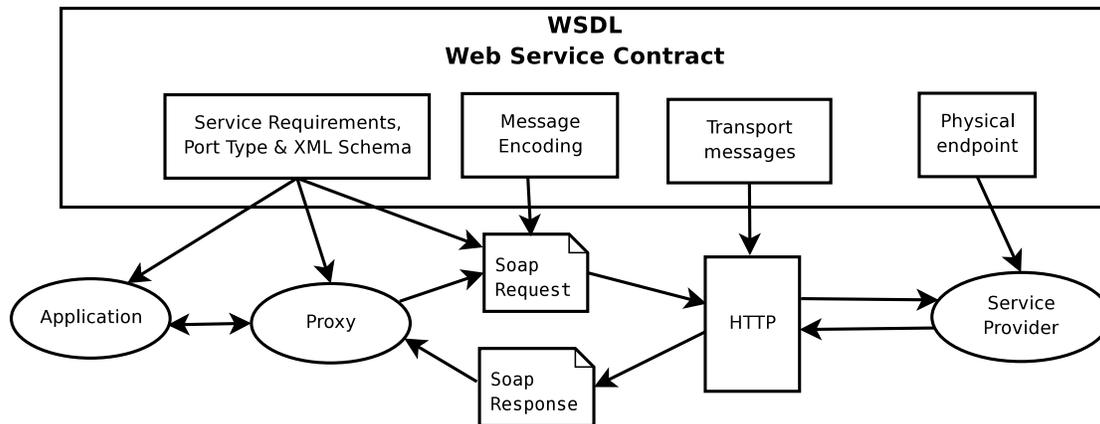
Figure 1. *Service Scenario.* **This figure illustrates the usual protocol in requesting and providing web services. All of the services are constrained by the web service contract, as expressed by the Web Service Definition Language (WSDL).**

In this paper, we propose an intermediate approach to web service specification. In particular, our approach balances the ease of expressivity offered by informal documentation with the mathematical rigor provided by traditional formalisms. Rather than attempting to capture the total behavior of each web service, our approach targets those aspects that are most relevant to practitioners but are most often lacking due to informal specifications. Our technique integrates the use of regular expressions in WSDL specifications to constrain the format of argument and return values to and from web services. This approach provides the basis for automating the generation of both client- and server-side checking wrappers.

The rest of this paper is organized as follows. In Section 2 we provide an overview of web services, WSDL and the associated tool set, as well as a weather service application that illustrates their use. In Section 2 we present our specification and checking approach and illustrate its benefits. Finally, we summarize our contributions in Section 4.

2. Web Services

Web standards such as WSDL, SOAP and UDDI are a set of open specifications that facilitate widespread adoption of service-oriented architectures (SOA) [4]. In Section 2.1, we provide an overview of the WSDL and SOAP standards and how they may be exploited to implement and access web services. In Section 2.2, we provide an example that uses web services to retrieve weather information and provides an illustrative example of the use of these standards.

2.1. Overview of Web Services

Figure 1 provides an overview of a typical web service scenario. The rectangle in the top half of the figure illustrates the functionality of the Web Service Definition Language (WSDL) that is used to specify the service contract for the particular service being provided. The bottom half of the figure illustrates the run-time execution sequence in a typical web service transaction. The WSDL service contract defines the information needed for interoperability, including detailed information such as the message format and transport details, but also abstracts the execution environment so that different platforms and systems can provide the same or similar service.

We include in the figure four components of the WSDL contract. The first component, listed in the top left box of Figure 1, is labeled Service Requirements, Port Type & XML Schema. The Service Requirements are used by (1) the application, Application, under development, shown on the bottom left of Figure 1, by (2) the generated proxy, Proxy, and (3) by the generated soap request, Soap Request, also shown on the bottom of the figure. In the next section we describe a weather application that we use to demonstrate a typical service scenario; however, the application can be any program that provides a web service.

The Proxy is usually generated automatically by a proxy generation tool that recognizes WSDL specifications. We use Microsoft .NET and C# to build our weather application and the corresponding tools to generate the associated proxy. Similarly,
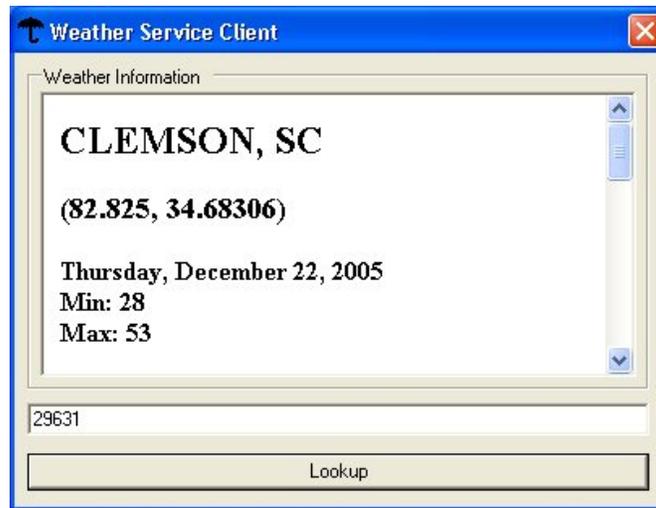
Figure 2. *Graphical Interface for The Weather Application.* **This figure shows the graphical interface for our weather application, where the user enters a zip code into the box above the** Lookup **button. When the** Lookup **button is pressed, the longitude, latitude, date and weather appear on the large canvas for the zip code area. The scroll bar in the canvas permits the user to view the weather for one week.**

```
(1)    <wsdl:portType name="WeatherForecastHttpGet">
(2)      <wsdl:operation name="GetWeatherByZipCode">
(3)        <documentation xmlns="http://schemas.xmlsoap.org/wsdl/">
(4)          Get one week weather forecast for a valid Zip Code(USA)
(5)        </documentation>
(6)        <wsdl:input message="tns:GetWeatherByZipCodeHttpGetIn" />
(7)        <wsdl:output message="tns:GetWeatherByZipCodeHttpGetOut" />
(8)      </wsdl:operation>
(9)    </wsdl:portType>
```

Figure 3. *Partial WSDL Specification for The Weather Application.*

the Soap Request is a SOAP message, also generated automatically, encoding information required to provide the service. SOAP is an messaging format, based on XML, that obviates the need for customized message passing infastructures. This common message syntax also permits reuse of functionality to parse and validate messages and facilitates interoperability among competing web service providers.

The Soap Request is passed to the server interface, shown in Figure 1 as a rectangle labeled HTTP, which then passes the request to the Service Provider, shown on the lower right corner of the figure. The Service Provider executes the requested service, and does whatever is necessary to execute the requested service, sending the computed information back to the server interface, which then builds a Soap Response message to pass back to the Proxy. Finally, the Proxy parses the Soap Response and passes the information to the Application. We provide further details of this scenario by describing our weather application in the next section.

2.2. Example of Web Services

In this section we present a weather application to demonstrate a typical web service scenario. In Section 3, we use this example to demonstrate our use of contracts to improve the predictable assembly of web services.

We use Microsoft .NET and the C# programming language to develop our weather application, including a graphical user

```
namespace WeatherServiceClient.net.webservicex.www {
   public System.IAsyncResult
      BeginGetWeatherByZipCode(string ZipCode, System.AsyncCallback
         callback, object asyncState) {
            return this.BeginInvoke("GetWeatherByZipCode",
               new object[] { ZipCode}, callback, asyncState);
   }
   public WeatherForecasts GetWeatherByZipCode(string ZipCode) {
      object[] results = this.Invoke("GetWeatherByZipCode",
                  new object[] { ZipCode});
         return ((WeatherForecasts)(results[0]));
   }
   public WeatherForecasts EndGetWeatherByZipCode(
      System.IAsyncResult asyncResult) {
            object[] results = this.EndInvoke(asyncResult);
            return ((WeatherForecasts)(results[0]));
   }
}
```

Figure 4. *Segment of C# Proxy generated from WSDL Specification for The Weather Application.*

interface (GUI) and the necessary Proxy components. The GUI is shown in Figure 2, including a scrolled canvas to permit easy viewing of the results, an input box to permit the user to enter information, and a button to permit the user to execute the application. In the figure, the user has typed a zip code of 29631, pressed the Lookup button, and the web service has responded with the requested information. The Proxy has interpreted the information and painted the scrolled canvas with the location, the city of Clemson SC, the longitude and latitude, the date, Thursday, December 22, 2005, and the minimum and maximum temperatures of Min: 28 and Max: 53. The canvas can be scrolled to reveal the date and temperature for the next six days.

In Section 2.1, we provided an overview of the basic elements of a WSDL specification. Figure 3 illustrates a portion of a WSDL specification corresponding to a service used to obtain information about the current weather conditions at various places in the United States. Line 1 in Figure 3 lists the <portType> for our weather application. The <portType> element is an important WSDL element. It describes a web service, the operations that the web service can perform, and the messages that are involved in providing the service. The <portType> element can be compared to a function library (or a module, or a class) in a traditional programming language. For our weather application, the <portType> element specifies WeatherForecastHttpGet as the name of the port that uses the GetWeatherByZipCode operation, an HTML GET operation listed on line 2, to marshal information about the weather forecast.

A WSDL <documentation> element provides informal comments about the web service captured by the specification. The <message> element defines the data elements of an operation. Each message can consist of one or more parts. The parts can be compared to the parameters of a function call in a traditional programming language. The <message> elements listed on lines 6 and 7 of Figure 3 identify the messaging structures used to communicate with the web service. Lines 8 and 9 of Figure 3 are end tags for elements operation and portType respectively.

We used a WSDL tool to automatically generate a WSDL *proxy class* in C# source code. Figure 4 includes portions of a C# proxy class generated from the WSDL specification included in Figure 3. The source code and associated artifacts for our weather application can be obtained by contacting the authors.

3. Overview of the Contract Methodology

In the previous section, we described the specification capabilities provided by WSDL, which facilitate the development of web service applications. WSDL, for example, describes the message format and endpoint bindings, but does not include sufficient information about the expected format of arguments and return values. This limitation can potentially compromise the correctness of service-oriented architectures. To address this issue, and to understand the motivation for our work, we

4

return to the weather service client, described in the previous section. In particular, consider the development difficulties that might be encountered by a design team responsible for using the weather service illustrated in Figure 3.

A designer using the WSDL specification in Figure 3 to understand how to use the weather service, benefits from by the specification in a number of ways. For example, by examining the message format referenced by the WSDL specification, it is clear that the client is responsible for providing a string argument to the service, and should expect a string argument in return. The documentation tag illustrated on lines 3 through 5 provides additional semantic information. This informal comment indicates that the client is expected to provide a *valid* U.S. zip code and should expect weather forecast information in return. But what precisely does *valid* mean? This information is not clear from the documentation. Similarly, in what format will the weather forecast be returned?

Consider the correctness compromises that might be introduced by the ambiguity present in the WSDL specification. There are two valid forms of a U.S. zip code: a string containing five digits or a string containing five digits, a hyphen, followed by four digits. A designer that expects the service to work correctly for an extended zip code will obviously not include checks that prevent the user from entering an extended format zip code. If however, the web service does not allow extended zip code queries, the client application might exhibit unexpected behaviors. Indeed, while developing our example, this very scenario was encountered. Our client application was not designed to expect a formatting exception when an extended zip code was provided, therefore the exception thrown by the server was propagated to the interface illustrated in Figure 2. We encountered similar problems when attempting to compute over the values returned by the web service. These issues are representative of a wide class of common formatting problems, and are precisely the problems we wish to address.

To make formatting expectations clear to the designer, we extend the WSDL specification syntax to include regular expressions for each argument and return value. One approach to doing this would be to use formal regular expression notation. However, we are interested in automating the run-time checking of these formatting requirements; thus, we use regular expression syntax consistent with the .NET library. This approach allows us to capture formatting requirements precisely, while simplifying the development of the corresponding monitoring code. As part of our future work, we expect to integrate our checking code generator with the proxy generator provided by Visual Studio .NET. This would provide seamless integration of specification and checking into a development process familiar to many practitioners.

Consider the benefits provided by this approach in the context of our weather service example. To avoid the ambiguity that led to a defect in the original version of our client application, the WSDL specification can be extended with a regular expression bound to the zip code argument that indicates an expectation of five numeric characters. This is, of course, a simple example. The regular expression bound to a return value that aggregates multiple data elements would be much more complex. In fact, some web services return XML strings as return values, which might require a more powerful notation than regular expressions provide, such as a context-free grammar. Regular expressions and context-free grammars are well-understood by many practitioners, making the notations immediately accessible and practically useful. The result is that designers and implementors of service-oriented systems will have a precise understanding of the data format that must be provided and the data format that will be returned.

There are additional benefits beyond the comprehension benefits provided by our approach. Even with a precise understanding of formatting requirements, errors of oversight can and do occur. The extended WSDL specifications can be used to automatically generate client-side checking wrappers, because the WSDL extensions are based on a grammar recognized by the .NET regular expression library.

Our approach not only improves the reliability of the system, but also provides a convenient opportunity for checking server-side requirements on the client side. To see the utility of this approach, recall that in the weather service example, the formatting error was detected by the server rather than the client. This round-trip to the server incurred a performance penalty on both the client and the server. Detecting the formatting error on the client-side avoids this round-trip penalty.

Client-side checking is of course valuable when the web service being accessed does not itself check formatting requirements. More surprising are the benefits received when accessing a web service that does include appropriate checking logic. Although there is a slight storage penalty for shadowing this logic on the client-side, avoiding the round-trip network exchange justifies this overhead in many cases.

4. Summary

Our work is motivated by the observation that the service-oriented paradigm is founded on an assumption of well-specified and well-understood contracts that is not realized in practice. In this paper, we introduce an approach to extending the WSDL specification language with support for argument and return format specification. Our approach brings us one step closer to realizing the assumptions upon which the paradigm is based. We believe that this work is important in reducing the adoption

barriers that have slowed the acceptance of web-services and service-oriented architectures. This is especially important as we come closer to realizing the vision of ubiquitous computing that promises transparent integration of widely distributed services.

References

[1] D. K. Barry. *Web Services and Service-Oriented Architectures*. Morgan Kaufmann, 2003.

[2] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley, 1995.

[3] L. G. Meredith and Steve Bjorg. Contracts and types. *Commun. ACM*, 46(10):41–47, 2003.

[4] E. Newcomer and G. Lomow. *Understanding SOA with Web Services*. Addison-Wesley, 2005.