Revising WSDL Documents: Why and How

Although Web service technologies promote reuse, Web Services Description Language (WSDL) documents that are supposed to describe the API that services offer often fail to do so properly. Therefore, finding services, understanding what they do, and reusing them are challenging tasks. The authors describe the most common errors they've found in real WSDL documents, explain how these errors impact service discovery, and present some guidelines for revising them.

Marco Crasso, Juan Manuel Rodriguez, Alejandro Zunino, and Marcelo Campo
Universidad Nacional del Centro de la provincia de Buenos Aires

From their origins, Web service technologies were conceived for enabling Web-based application reuse. As Ian Foster stated, "Web services have little value if others cannot discover, access, and make sense of them."[1] Service providers should clearly describe what a Web service offers (making sense of the service) and how to use it (accessing the service). Service descriptions are mostly brought down to Earth using the Web Services Description Language (WSDL), an XML dialect sponsored by the W3C (www.w3.org/TR/wsdl). WSDL documents are crucial in enabling third parties to make sense of services and access them. Such descriptions also play an important role in discovering services[2,3] because Web service search engines rely on WSDL documents to support discovery.[4,5] Third parties must be able to achieve all three functions for WSDL documents to be of use.

Despite service descriptions' importance, consensus doesn't yet exist on how to create them. For instance, following the principles of model-driven architecture, the focus should be on how the service problem domain is modeled rather than on how providers developed the service or how its offered functionality is exposed.[6] Furthermore, a dichotomy exists between describing a service before implementing it and vice versa. We see the glaring headline "code first vs. contract first" every time we look around — on the Internet, in blogs, and even in magazines. Broadly, code first, means inferring a WSDL document from a service implementation, whereas contract first means describing the service before implementing it. Both approaches have pros and cons, making it impossible to claim that one over the other will always be a better choice.

Independent of how services were created, however, service providers should always "revise" service descriptions to ensure that they meet the three criteria for third-party reuse. Although this suggestion might sound obvious, paradoxically, developers tend to create descriptions that hinder services' understandability and discoverability, as several researchers have pointed out.[2,3,7–9] This situation motivated us to survey common mistakes that service providers should avoid when creating WSDL documents, and create guidelines to correct them.

Common Mistakes in WSDL Documents

With service-oriented architecture (SOA), systems are composed of independent software components, called services, that interact with each other through remote call mechanisms. When developers implement such services using standard Web languages and protocols, we call them Web services. To access a Web service, a service consumer must obtain the associated WSDL document. Commonly, WSDL documents are made available through Web service search engines representing a crossroad in the path of service providers and consumers. A typical WSDL document is structured as sets of interfaces, called *port types*, that consist of operations with *input*, *output*, and, optionally, *fault* messages. Additionally, each port type is linked to one or more access protocols, such as SOAP or HTTP, via bindings. Operation messages have one or more parts for transporting XML data defined using the XML Schema Definition (XSD). Each element of a service description has a name and might have a textual comment associated with it. We can summarize WSDL grammar as Figure 1 shows.

```
<documentation .... />?

<types>?
    <documentation .... />?
    < schema .... />*
</types>
<message name="nmtoken">*
    <documentation .... />?
    <part name="nmtoken" element="qname"?
type="qname"?/>*
</message>
<portType name="nmtoken">*
    <documentation .... />?
    <operation name="nmtoken">*
        <documentation .... />?
        <input name="nmtoken"? message="qname">?
            <documentation .... />?
        </input>
        <output name="nmtoken"? message="qname">?
            <documentation .... />?
        </output>
        <fault name="nmtoken" message="qname">*
            <documentation .... />?
        </fault>
    </operation>
</portType>
```

*Figure 1. Web Services Description Language (WSDL) grammar version 1.1. Note that ? means optional and * means none or many.*

We've analyzed 391 WSDL documents gathered from the Internet[3] and found that the functionality of many would be hard for third-party consumers to understand. We extrapolated poor practices found in the surveyed WSDL documents from well-known bad practices for coding component interfaces, using other programming paradigms — such as structured and object-oriented ones — to analyze these bad practices' implications and suggest solutions to them in the context of Web services.

First, we noted that developers seem to take little care of the names and comments in WSDL documents. Nobody would argue that commenting source code is a good practice, but fewer than 50 percent of the documents in the analyzed dataset have some documentation. This percentage is in accord with findings from Jianchun Fan and Subbaras Kambhampati, who analyzed comments from a different set of WSDL documents in 2005.[8] Furthermore, we detected that part names are commonly related to their role in the operation or the supported communication protocols — the names we encountered most frequently were `parameters`, `body`, and `return` — but don't give an idea of what they represent. This is undesirable because, when WSDL elements are undocumented, names are the only available element description. In Amazon's AWSECommerce service (http://webservices.amazon.com/AWSECommerceService/AWSECommerceService.

wsdl), which is bound to SOAP over HTTP, all message parts are named `body`. Clearly, the lack of comments and the proliferation of cryptic names won't be helpful to potential service users. This finding is supported by results from M. Brian Blake and Michael F. Nowlan,[2] who detected name tendencies within WSDL documents.

The second identified bad practice is tying port types to concrete protocols, given that the real purpose of port types is to enable different bindings for a single type, such as SOAP/HTTP or SOAP/SMTP. However, we found that port types are tied to bindings in 60 percent of our dataset. For example, the FraudLab service (http://ws.fraudlabs.com/fraudlabswebservice.asmx?wsdl) defines the "same" port type three times but binds each one to a different protocol. Such port types usually contain a protocol reference in their names, such as `FindServiceSoap` from Microsoft's Bing Maps platform services (http://staging.mappoint.net/standard-30/mappoint.wsdl). In some way, port types are to Web services what headers are to C, so this practice is similar to redefining OpenGL headers for each implementation — clearly, a weird thing to do, which generates unnecessarily big and puzzling WSDL documents.

Another commonly found bad practice is to place semantically unrelated operations in a unique port type, although modules with high cohesion tend to be preferable in structured design. Cohesion refers to how strongly operations are functionally related within a service. The operations in these services must be highly related to one another — that is, highly cohesive.[10] For example, the Amazon Elastic Compute Cloud (EC2) service (http://s3.amazonaws.com/ec2-downloads/2009-10-31.ec2.wsdl) has 74 operations for managing images, volumes, security, instances, and snapshots, grouped in a single port type. By grouping cohesive operations within separate port types — that is, a port type for managing images, another for volumes, and so on — each port type might be more cohesive while avoiding problems similar to "God classes" (that is, a single class in charge of everything). Another problematic practice is to include operations that return information about service performance or availability within the same port type.

Another detected problem that occurs in 10 percent of our dataset is overloading output messages to transport operation results and piggyback errors. This can make the service functionally difficult for third-party developers to understand because it requires service operations to use flexible data types for conveying either output results or error data.[7] We can see a clear example in Amazon's SimpleDB service (http://sdb.amazonaws.com/doc/2009-04-15/AmazonSimpleDB.wsdl). This service offers an operation named `Select` whose output message carries a set of items when everything goes well; otherwise, it carries error information. By contrast, services within NASA's Earth Observing System (EOS) Clearing House (ECHO) project provide a good example of how to deal with operation errors — in this case, all offered operations use fault messages to transport different kinds of errors, such as `InvalidArgumentFault`, `ItemNotFoundFault`, and `AuthorizationFault` (http://api.echo.nasa.gov/echo-wsdl/v10/ExtendedServicesService.wsdl).

Finally, we found two recurrent data modeling problems: defining general-purpose data types and repeating data types. Some XSD constructors can define data types capable of exchanging any XML content. James Pasley called these constructors *wild cards*.[9] If a message is associated with a wild card, a potential user can't predict how its content will look. Although wild cards obscure the operations' domain and range, they're present in 15 percent of our dataset.

With regard to the second data modeling problem, 28 percent of the WSDL documents contained at least one repeated data type definition. We noticed that service providers commonly defined specific data types for each message, regardless of whether the messages needed to convey the same information. Repeated structures ranged from simple XSD built-in types, such as `double` or `string`, to user-defined ones such as `PayOrder`. This seems to be related to the fact that 70 percent of the documents we analyzed have the data model — that is, the XSD code — defined within them. To clarify, suppose you have a service for checking stocks when the market is open and another for when the market is closed. Both services offer an operation that retrieves market information; the only difference is when the service collects that information. To define data types, developers can repeat the data model in both WSDL documents. Alternatively, they can define

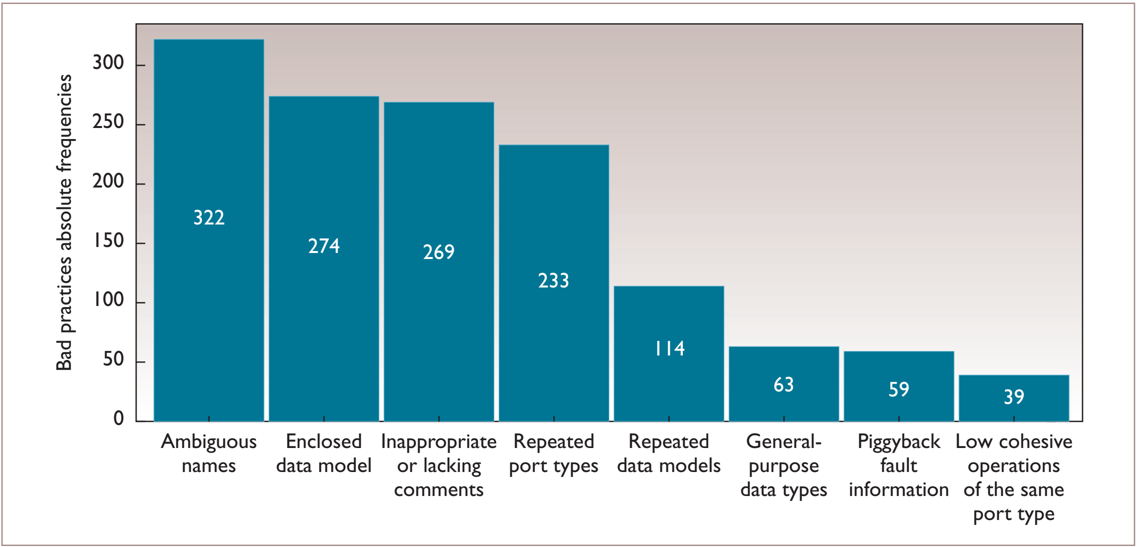


Figure 2. Identified bad practices in WSDL documents. Less than 50% of the analyzed services are well documented, while less than 20% of them have carefully selected names.

the data model in an XSD document and then import or include it from the WSDL ones. In general, the latter approach is the best alternative (see this service from the Argentinean interior ministry: http://webservices.mininterior.gov.ar/Feriados/Service.svc?wsdl).

WSDL documents aren't supposed to be big, puzzling, noncohesive, undocumented, or wrongly named, mainly because their real consumers are third-party developers. However, as Figure 2 shows, the creators of the WSDL documents we analyzed appeared to ignore ongoing service design principles and years of consensus on the right and wrong way to codify software APIs.[10,11]

Revising Your WSDL Documents

No silver bullet guarantees that potential consumers will discover, understand, and access a particular Web service. However, we believe that you can improve a WSDL document by following six steps:

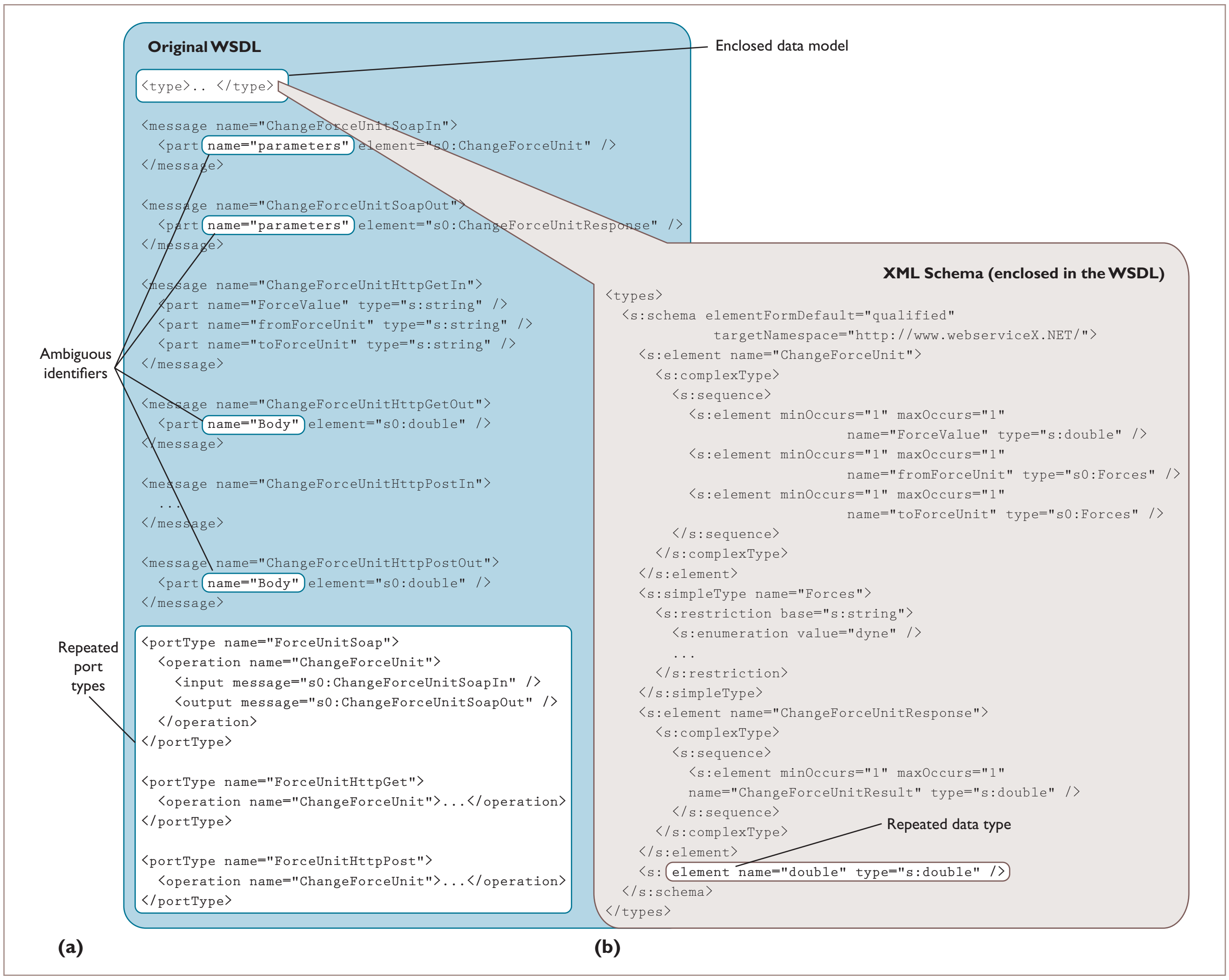
1. separating the schema from the definition of the offered operations;
2. removing repeated WSDL and XSD code;
3. putting error information within fault messages and only conveying operation results within output ones;
4. replacing WSDL element names with explanatory names if original names are cryptic;
5. moving noncohesive operations from their original port type to separate port types; and
6. documenting the operations.

The first step means moving complex data-type definitions into a separate XSD document and adding the corresponding import sentence into the WSDL document. However, when a developer isn't going to reuse data types, those types can be included in the WSDL document to make it self-contained.

The second step deals with redundant code in both the WSDL document and the schema. Repeated WSDL code might stem from port types tied to a specific protocol, whereas redundant XSD comes from data definitions bounded to a particular operation. So, you can remove repeated WSDL code by defining a protocol-independent port type. Similarly, to eliminate redundant XSD code, you should abstract repeated data types into a single type and change message part references for references to the new data type.

The third step intends to separate error information from output information. To do this, you should remove error information from output messages and place it in fault messages. Moreover, you should define fault messages to transport the different errors that the operation might throw.

The fourth step aims to improve how representative WSDL element names are by renaming nonexplanatory ones. Grammatically, an operation's name should be in the form <verb> + <noun> because an operation is an action; message, message part, or data-type names should be a noun or noun phrase because they represent the objects on which the operation executes.



```
<type>.. </type>

<message name="ChangeForceUnitSoapIn">
  <part name="parameters" element="s0:ChangeForceUnit" />
</message>

<message name="ChangeForceUnitSoapOut">
  <part name="parameters" element="s0:ChangeForceUnitResponse" />
</message>

<message name="ChangeForceUnitHttpGetIn">
  <part name="ForceValue" type="s:string" />
  <part name="fromForceUnit" type="s:string" />
  <part name="toForceUnit" type="s:string" />
</message>

<message name="ChangeForceUnitHttpGetOut">
  <part name="Body" element="s0:double" />
</message>

<message name="ChangeForceUnitHttpPostIn">
  ...
</message>

<message name="ChangeForceUnitHttpPostOut">
  <part name="Body" element="s0:double" />
</message>

<portType name="ForceUnitSoap">
  <operation name="ChangeForceUnit">
    <input message="s0:ChangeForceUnitSoapIn" />
    <output message="s0:ChangeForceUnitSoapOut" />
  </operation>
</portType>

<portType name="ForceUnitHttpGet">
  <operation name="ChangeForceUnit">...</operation>
</portType>

<portType name="ForceUnitHttpPost">
  <operation name="ChangeForceUnit">...</operation>
</portType>
```

```
<types>
  <s:schema elementFormDefault="qualified"
           targetNamespace="http://www.webserviceX.NET/">
    <s:element name="ChangeForceUnit">
      <s:complexType>
        <s:sequence>
          <s:element minOccurs="1" maxOccurs="1"
                           name="ForceValue" type="s:double" />
          <s:element minOccurs="1" maxOccurs="1"
                           name="fromForceUnit" type="s0:Forces" />
          <s:element minOccurs="1" maxOccurs="1"
                           name="toForceUnit" type="s0:Forces" />
        </s:sequence>
      </s:complexType>
    </s:element>
    <s:simpleType name="Forces">
      <s:restriction base="s:string">
        <s:enumeration value="dyne" />
        ...
      </s:restriction>
    </s:simpleType>
    <s:element name="ChangeForceUnitResponse">
      <s:complexType>
        <s:sequence>
          <s:element minOccurs="1" maxOccurs="1"
          name="ChangeForceUnitResult" type="s:double" />
        </s:sequence>
      </s:complexType>
    </s:element>
    <s:element name="double" type="s:double" />
  </s:schema>
</types>
```

Figure 3. Original WSDL document demonstrating bad practices. We can see (a) the messages and port types and (b) the enclosed XML Schema Definition code.

If message names represent actions rather than objects, the information conveyed in those messages might modify the operation's behavior, hindering the operation's purpose. Additionally, you should write names according to common notations, and they should be between three and 15 characters long,[2] because these practices facilitate automatic analyses and human reading, respectively.[4] For example, you should rewrite the name `theelementname` as `theElementName`.

The fifth step is to place operations in different port types based on their cohesion. To do this, you should divide the original port type into smaller and more cohesive port types. You should repeat this step until new port types are cohesive enough.

Finally, all operations must be well documented. We can say an operation is well documented when it has a concise and explanatory comment that describes the offered functionality. Moreover, because WSDL lets developers comment on each part of a service description separately, a good practice is to place every `<documentation>` tag in the most restrictive ambit possible. For instance, if the comment refers to a specific operation, you should place it in that operation.

Except for steps 4 and 6, the other steps might require that you modify service implementations. Moreover, as a result of applying this guide, you'll have two versions of a revised service description. Although further discussion is out of this article's scope, we'll note that some version-support technique is necessary to let clients that use the old service version continue using the service until they migrate to the new one.[12]

```
<types>
  <xsd:schema targetNamespace="...">
    <xsd:import schemaLocation="forces.xsd"
           namespace="..." />
  </xsd:schema>
</types>

<message name="ChangeForceUnitIn">
  <part name="ForceValue" type="s:double" />
  <part name="fromForceUnit" type="s0:Force" />
  <part name="toForceUnit" type="s0:Force" />
</message>

<message name="ChangeForceUnitOut">
  <part name="ForceValue" element="s:double" />
</message>

<portType name="ChangeForceUnit">
  <operation name="ChangeForceUnit">
      <documentation>This service converts a force measure
        in a force unit to the same measure
        in another force unit</documentation>
    <input message="s0:ChangeForceUnitIn" />
    <output message="s0:ChangeForceUnitOut" />
  </operation>
</portType>
```

"forces.xsd"

```
<?xml version="1.0"?>
<s:schema xmlns:s="http://www.w3.org/2001/XMLSchema">
<s:simpleType name="Forces">
  <s:restriction base="s:string">
    <s:enumeration value="dyne" />
    <s:enumeration value="gramforce" />
    <s:enumeration value="poundals" />
    <s:enumeration value="newtons" />
    <s:enumeration value="pounds" />
    <s:enumeration value="kilopondkgmforce" />
    <s:enumeration value="Kip" />
  </s:restriction>
</s:simpleType>
</s:schema>
```

Figure 4. Revised WSDL document. We refactored it using four of the six steps in our approach to improve the WSDL document's understandability.

Revising an Illustrative WSDL Document

We conducted a case study that exemplifies how to use our described approach. We selected a WSDL document from our dataset (www.webservicex.net/ConvertForec.asmx?WSDL) and, in turn, followed the steps in our guide.

The selected Web service converts a force measure given in some unit, such as dyne, gram-force, or newtons, to another unit. This service offers one operation, `ChangeForceUnit`, that receives a force value, its force unit, and the target force unit, and returns a force value. This service, albeit simple, contains several bad practices, making it an excellent candidate to illustrate the reviewing process.

Figure 3a shows the messages and port types for the selected WSDL document, whereas Figure 3b shows the enclosed XSD code. As we can see, four bad practices occur in this description: an enclosed data model, ambiguous identifiers, repeated data types and port types, and no documentation.

First, we separated the schema from the original WSDL document (step 1) and imported the resulting XSD file from the revised WSDL document. Then, we removed the repeated code in both the WSDL document and its schema (step 2) — that is, the redefinition of the type `double` and the redefinition of the same port type. Specifically, we remedied the data model problem by deleting the redefinition of `double` and replacing all references with references to the built-in `s:double` type. To handle the repeated port types, we removed all but one, after which dangling messages appeared, which we also removed. Finally, we updated all binding elements to point to the new port type.

We applied step 4 by replacing all nonexplanatory names with names that represent port-type semantics, operation semantics, and the information the messages exchange. Finally, we documented the operation (step 6).

Because the service offers only one operation, we didn't need to draw noncohesive operations (step 5). With regard to step 3, removing error information from output messages might be the most difficult step to accomplish because, if the output data type is too generic, we can't know the output message's purpose unless we also revise the service's implementation and invoke it until it fires an error or its documentation explicitly indicates this anomaly. Because we didn't have enough information to detect this problem in the case study, we omitted step 3.

Figure 4 depicts the revised WSDL document. Its first characteristic is that it's shorter than the original, but the number of descriptive words, such as `unit`, `force`, or `change`, has increased. Moreover, the revised description uses more specific words, and the port-type name is protocol-independent — that is, we removed the `Soap` term.

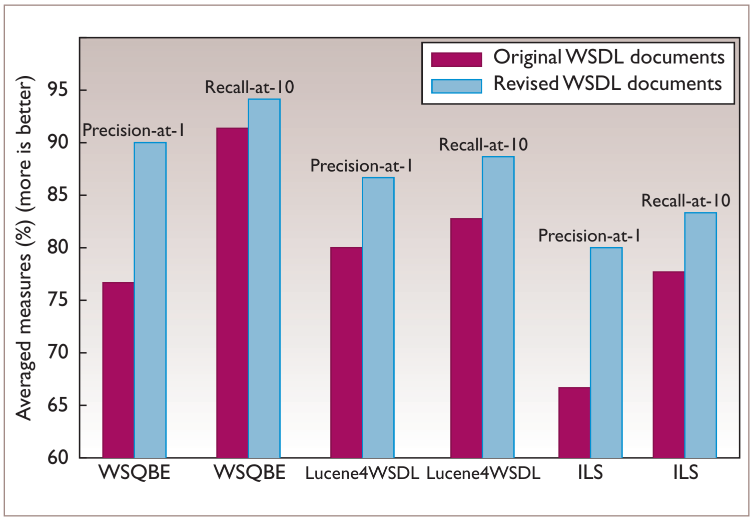


Figure 5. Impact of revising WSDL documents on Web service search engines. The better performance of search engines fed with the improved dataset implies that service will be ranked more according to the service consumer's needs.

Discovering Revised WSDL Documents

To measure whether revising WSDL documents impacts discovery, we fed both the original and revised versions of the dataset to three Web service search engines. We compared each search engine's effectiveness when using original WSDL documents against their effectiveness using revised versions.

For this comparison, we employed Lucene (http://lucene.apache.org), Web Service Query by Example (WSQBE),[4] and Eleni Stroulia and Yiqiao Wang's approach.[5] Lucene is a well-known open source search software that follows a classic information retrieval (IR) approach. Because we modified Lucene to ignore WSDL-reserved words that could negatively affect its performance, we refer to it here as Lucene-4WSDL. WSQBE combines IR techniques with a search-space reduction mechanism based on WSDL document classification. We call the final approach[5] ILS (IR + Lexical + Structural) because it combines IR techniques with term expansion based on lexical relations, such as synonyms, hypernyms, and hyponyms, and compares the retrieved candidates' structure to a WSDL specification of the desired service, which the user who performs the discovery must supply. Notwithstanding their differences, the three search engines return a ranked list of candidate services for a given query.

To ensure that our experiment was fair, we took two precautions. First, because the employed search engines' performance depends on the dataset and the queries given as inputs, we used the same 30 queries (described elsewhere[4]) with each version of the dataset. Second, to avoid influencing the results, the developers who revised the dataset didn't know the queries. The time needed to improve each WSDL document was 15 minutes per developer, on average. We didn't assess the time needed for synchronizing the improved WSDL documents and their underlying implementations. Because ILS's inquiry interface optionally accepts a functional description of the desired services using WSDL, we built a WSDL document for representing each query.

Our experimental methodology was to query the search engines, calculate whether the first ranked service was relevant (termed "precision-at-1"), and calculate how many relevant services were ranked before the 11th position ("recall-at-10"). Finally, we averaged the results over the 30 queries.

Figure 5 shows that the search engines performed better using the revised version of the dataset. Concretely, the precision-at-1 results suggest that the revised dataset retrieved more relevant services in the first position. Lucene-4WSDL obtained a gain of 6.67 points (that is, precision-at-1 was 6.67 percent higher), WSQBE gained 10 points, and ILS gained 13 points.

Recall-at-10 results indicate that the search engines retrieve more relevant services within a window of 10 candidates with the revised dataset. Specifically, we observed improvements of 2.18, 3.34, and 5.63 points with Lucene4WSDL, WSQBE, and ILS, respectively.

Although all the search engines behaved better with the revised dataset, ILS and WSQBE achieved higher improvements. If a WSDL document contains nonexplanatory terms, ILS's term expansion technique will generate a service representation that contains even more non-explanatory terms, degrading its effectiveness. Additionally, as it was reported previously,[4] WSQBE's search-space reduction mechanism performs better when WSDL documents contain domain-specific terms rather than too-general ones. Most likely, this accounts for the outstanding improvements in WSQBE, but this topic deserves a deeper analysis.

The results related to the revised dataset surpass those achieved using the original, regardless of the search engine employed, which suggests that improvements are due to the revised WSDL documents rather than the underlying search engine.

Note that when using the revised dataset, the employed search engines performed better in retrieving a relevant service at the top of the rank. Different experiments support this result: because users tend to select higher-ranked search results, even a small improvement in a rank has a great impact on discoverability. For instance, the probability that a user accesses the first ranked result is 90 percent and 60 percent that the user will access the second.[13] This further strengthens the importance of our proposed guidelines in improving the "value" (to use Foster's words[1]) of service descriptions.

The results of our research show that we can reasonably expect that removing detected bad practices will result in at least a small improvement to WSDL documents' discoverability, but developers currently don't often revise such documents. Perhaps the presented evidence will nudge them to do it.

We're now conducting research on heuristics for automatically detecting poor practices in Web service descriptions.[14] We aim to assist developers in making more representative descriptions by automatically identifying the poor practices and suggesting suitable refactorizations. Moreover, we're surveying popular tools for building WSDL documents from source code (that is, those that follow the code-first approach) because we suspect that some of the aforementioned bad practices can stem from these tools. Furthermore, we're planning to extend this study for analyzing how developers use WSDL extensions. Regarding service discovery, we'll analyze semantic WSDL (WSDL-S)[15] as a next step for improving service description quality.

Acknowledgments

We thank the associate editor, anonymous reviewers, and the content editor for their comments on this article. Also, thanks to ANPCyT for supporting this research through grants PAE-PICT 2007-02311 and PAE-PICT 2007-02312.

References

1. I. Foster, "Service-Oriented Science," *Science*, vol. 308, no. 5723, 2005, pp. 814–817.
2. M.B. Blake and M.F. Nowlan, "Taming Web Services from the Wild," *IEEE Internet Computing*, vol. 12, no. 5, 2008, pp. 62–69.
3. J.M. Rodriguez et al., "Discoverability Anti-Patterns: Frequent Ways of Making Undiscoverable Web Service Descriptions," *Proc. 10th Argentine Symp. Software Eng.*, SADIO, 2009, pp. 1–15.
4. M. Crasso, A. Zunino, and M. Campo, "Easy Web Service Discovery: A Query-by-Example Approach," *Science of Computer Programming*, vol. 71, no. 2, 2008, pp. 144–164.
5. E. Stroulia and Y. Wang, "Structural and Semantic Matching for Assessing Web Service Similarity," *Int'l J. Cooperative Information Systems*, vol. 14, no. 4, 2005, pp. 407–438.
6. T.O. Meservy and K.D. Fenstermacher, "Transforming Software Development: An MDA Road Map," *Computer*, vol. 38, no. 9, 2005, pp. 52–58.
7. J. Beaton et al., "Usability Challenges for Enterprise Service-Oriented Architecture APIs," *Proc. IEEE Symp. Visual Languages and Human-Centric Computing*, IEEE CS Press, 2008, pp. 193–196.
8. J. Fan and S. Kambhampati, "A Snapshot of Public Web Services," *ACM SIGMOD Record*, vol. 34, no. 1, 2005, pp. 24–32.
9. J. Pasley, "Avoid XML Schema Wildcards for Web Service Interfaces," *IEEE Internet Computing*, vol. 10, no. 3, 2006, pp. 72–79.
10. M.P. Papazoglou and W.-J. Van Den Heuvel, "Service-Oriented Design and Development Methodology," *Int'l J. Web Eng. Technology*, vol. 2, no. 4, 2006, pp. 412–442.
11. T. Erl, *SOA Principles of Service Design*, Prentice Hall, 2007.
12. M.B. Juric et al., "WSDL and UDDI Extensions for Version Support in Web Services," *J. Systems and Software*, vol. 82, no. 8, 2009, pp. 1326–1343.
13. E. Agichtein et al., "Learning User Interaction Models for Predicting Web Search Result Preferences," *Proc. 29th Ann. Int'l ACM SIGIR Conf. Research and Development in Information Retrieval*, ACM Press, 2006, pp. 3–10.
14. J.M. Rodriguez et al., "Automatic Web Service Discoverability Anti-Patterns Detection," *Proc. 10th IFIP Conf. E-Business, E-Services, and E-Society* (I3C 10), to appear, 2010.
15. K. Li et al., "Designing Semantic Web Processes: The WSDL-S Approach," *Semantic Web Services, Processes and Applications*, Dec. 2006, pp. 161–193.

Marco Crasso is a teaching assistant at the Universidad Nacional del Centro de la provincia de Buenos Aires (UNICEN). His research interests include the application of machine learning and data mining techniques to ease the development of service-oriented applications, and programming models for Web service consumption. Crasso has a PhD in computer science from UNICEN. He's a member of the ISISTAN Research Institute. Contact him at mcrasso@exa.unicen.edu.ar; www.exa.unicen.edu.ar/~mcrasso/.

Juan Manuel Rodriguez is a teaching assistant and PhD candidate at the Universidad Nacional del Centro de la provincia de Buenos Aires (UNICEN). His research interests are in grid computing, service-oriented computing, Web services, and mobile devices. Rodriguez has a system engineer degree from UNICEN. He's a member of the ISISTAN Research Institute. Contact him at jmrodri@exa.unicen.edu.ar; www.exa.unicen.edu.ar/~jmrodri/.

Alejandro Zunino is an adjunct professor at the Universidad Nacional del Centro de la provincia de Buenos Aires (UNICEN). His research interests are in grid computing, service-oriented computing, Web services, Semantic Web services, agent development tools, agent frameworks, mobile agents, and reactive mobility. Zunino has a PhD in computer science from UNICEN. He's a member of the ISISTAN Research Institute and Consejo Nacional de Investigaciones Científicas y Técnicas (CONICET). Contact him at azunino@conicet.gov.ar; www.exa.unicen.edu.ar/~azunino.

Marcelo Campo is an associate professor at the Universidad Nacional del Centro de la provincia de Buenos Aires (UNICEN) and the head of the ISISTAN Research Institute. His research interests include intelligent aided software engineering, software architecture and frameworks, agent technology, and software visualization. Campo has a PhD in computer science from the Universidade Federal do Rio Grande do Sul, Porto Alegre, Brazil. He's a research fellow at Consejo Nacional de Investigaciones Científicas y Técnicas (CONICET). Contact him at mcampo@exa.unicen.edu.ar; www.exa.unicen.edu.ar/~mcampo.