University of Nebraska - Lincoln
DigitalCommons@University of Nebraska - Lincoln

CSE Technical reports | Computer Science and Engineering, Department of

12-20-2007

Automated Refinement and Augmentation of Web Service Description Files

Marc Fisher II
University of Nebraska - Lincoln, fisherii@google.com

Sebastian Elbaum
University of Nebraska - Lincoln, selbaum2@unl.edu

Gregg Rothermel
University of Nebraska - Lincoln, grothermel2@unl.edu

Follow this and additional works at: http://digitalcommons.unl.edu/csetechreports

Part of the Computer Sciences Commons

Fisher II, Marc; Elbaum, Sebastian; and Rothermel, Gregg, "Automated Refinement and Augmentation of Web Service Description Files" (2007). *CSE Technical reports.* Paper 22.
http://digitalcommons.unl.edu/csetechreports/22

This Article is brought to you for free and open access by the Computer Science and Engineering, Department of at DigitalCommons@University of Nebraska - Lincoln. It has been accepted for inclusion in CSE Technical reports by an authorized administrator of DigitalCommons@University of Nebraska - Lincoln.

University of Nebraska–Lincoln, Computer Science and Engineering
Technical Report TR-UNL-CSE-2007-0026
Issued Dec. 20, 2007

Automated Refinement and Augmentation of Web Service Description Files

Marc Fisher II, Sebastian Elbaum, and Gregg Rothermel

University of Nebraska-Lincoln
{mfisher,elbaum,grother}@cse.unl.edu

Abstract. Web Service Description Language (WSDL) is being increasingly used to specify web service interfaces. Specifications of this type, however, are often incomplete or imprecise. For example, cursory examination of the WSDL file for Amazon's E-Commerce Web Service reveals that it often uses a less specific type where a more specific type is applicable, or declares that elements could be missing where other documentation indicates that they are required. Further, specifications reflecting the temporal relationships between operations are completely missing, which is not surprising since they are not supported by the current WSDL standard. These problems in WSDL specifications can cause tools that use them to perform poorly or unreliably, and can mislead developers who rely on them. To address these problems, in this paper we present an automated methodology for collecting static and dynamic information about a web service, and using this information to suggest improvements to the WSDL file as well as providing complementary information about the behavior of the web service that cannot be captured by the WSDL. Additionally, we present the results of two case studies performed on commercial web services that show our methodology can find problems in WSDL files and suggest improvements.

1 Introduction

Many businesses provide access to their services and products through a web service. For example, a wide variety of businesses that provide services to other businesses, such as shipping companies and credit card processing organizations, provide a web service for accessing those services. Further, some businesses that target consumers, such as Amazon or eBay, also provide access to their products or services via a web service. By doing so, these businesses allow parties outside of the organization to utilize or resell their services or products.

Typically, Web Service Description Language (WSDL) files are used to provide a partial specification of the interface to a web service [1]. These specifications focus on data-types of parameters and returned values of the application. However, WSDL files are often imprecisely or incorrectly specified; for example, using general types such as strings rather than more specific types like integer or decimal, or failing to correctly specify the minimum and maximum occurrences for particular elements. Additionally, WSDL files are limited in the class of specifications they can provide; specifically, they do not include support for specifying the dependencies between operations.

With better WSDL specifications, tools such as Sun's JAX-WS or Apache's Axis that produce libraries for accessing web services can potentially build higher quality code. In particular they can perform more validation of inputs on the client side and even potentially enforce temporal constraints without having to submit requests to the server, decreasing the load on the servers providing the service. Better WSDL specifications will also help developers using web services create more robust and reliable applications.

In prior work we developed the WebAppSleuth methodology for characterizing web applications [2, 3]. This methodology probes the interfaces exposed in web applications via html forms and form handlers by automatically constructing http requests, submitting them to the web server, and processing the results to generate inferences about how the underlying application behaves. In this work, to address the previously mentioned issues with web services, we have extended our WebAppSleuth methodology to (1) automatically generate sequences of requests to a web service based on it WSDL specification, and (2) use the responses to these requests to generate suggestions for refining and augmenting that WSDL specification.

The remainder of this paper presents this work, as follows. Section 2 provides background on web services and WSDL. Section 3 presents our methodology. Section 4 describes an application of our methodology to two commercial web services, Amazon's E-Commerce Service and eBay's Trading web Service. Section 5 provides details about related work, and Section 6 concludes and discusses future work.

2 Background

Figure 1 is a partial WSDL file describing an example bookstore web service that conforms to the WSDL 1.1 standard [1]. This service allows client applications to search for books, login with an existing username and password, add books to a shopping cart and change the quantities of items already in the cart.

A WSDL file consists of six kinds of definitions: types, messages, port types, bindings, ports, and services. Together the type (lines 2-75) and message definitions describe the overall structure of request and response messages. The port type definitions (lines 76-95) describe a set of abstract operations and which messages are used for the input and output of each operation, while the bindings define the concrete protocol for the abstract operations of a port type. A port definition specifies the address for a particular binding and a service is used to aggregate a set of related ports.

Although all of these definitions are important when describing a web service, in this work we focus on type and port type definitions. The type definitions in a WSDL file tend to make up the majority of the document. For example, the WSDL file describing Amazon's E-Commerce Service is 3,244 lines long, with a type definition that is 2,850 lines long,[1] while the WSDL description for eBay's Trading Web Service is 90,450 lines long with a type definition that is 87,725 lines long.[2]

[1] `http://webservices.amazon.com/AWSECommerceService/2007-07-16/AWSECommerceService.wsdl`

[2] `http://developer.ebay.com/webservices/515/eBaySvc.wsdl`

```
1.<definitions ...>
2.  <types><schema>
3.    <simpleType name="Category">
4.      <restriction base="string">
5.        <enumeration value="Databases" />
6.        <enumeration value="Web Design" />
7.        <enumeration value="Programming" />
8.      </restriction>
9.    </simpleType>

10.   <element name="StartSessionResponse">
11.     <complexType><all>
12.       <element name="sessionId" type="string" />
13.     </all></complexType>
14.   </element>

15.   <element name="SearchRequest">
16.     <complexType><all>
17.       <element name="sessionId" type="string" />
18.       <element name="category" type="Category"
19.                 minOccurs="0" />
20.       <element name="name" type="string"
21.                ' minOccurs="0" />
22.       <element name="priceMin" type="decimal"'
23.                 minOccurs="0" />
24.       <element name="priceMax" type="decimal"'
25.                 minOccurs="0" />
26.     </all></complexType>
27.   </element>

28.   <element name="SearchResponse">
29.     <complexType><all>
30.       <element name="bookDetail" minOccurs="0"'
31.                 maxOccurs="unbounded">
32.         <complexType><all>
33.           <element name="itemId" type="integer" />
34.           <element name="name" type="string" />
35.           <element name="price" type="decimal" />
36.           <element name="category"'
37.                 type="Category" />
38.         </all></complexType>
39.       </element>
40.     </all></complexType>
41.   </element>

42.   <element name="LoginRequest">
43.     <complexType><all>
44.       <element name="sessionId" type="string" />
45.       <element name="login" type="string" />
46.       <element name="password" type="string" />
47.     </all></complexType>
48.   </element>
49.   <element name="AddToCartRequest">
50.     <complexType><all>
51.       <element name="sessionId" type="string" />
52.       <element name="itemId" type="integer" />
53.       <element name="quantity" type="integer" />
54.     </all></complexType>
55.   </element>

56.   <element name="UpdateQtyInCartRequest">
57.     <complexType><all>
58.       <element name="sessionId" type="string" />
59.       <element name="orderId" type="integer" />
60.       <element name="quantity" type="integer" />
61.     </all></complexType>
62.   </element>

63.   <element name="CartResponse">
64.     <complexType><all>
65.       <element name="cartItem" minOccurs="0"'
66.                 maxOccurs="unbounded">
67.         <complexType><all>
68.           <element name="orderId" type="integer" />
69.           <element name="itemId" type="integer" />
70.           <element name="quantity" type="integer" />
71.         </all></complexType>
72.       </element>
73.     </all></complexType>
74.   </element>
75. </schema></types>

76. <portType name="BookstoreAPI">
77.   <operation name="StartSession">
78.     <output message="StartSessionResponse" />
79.   </operation>
80.   <operation name="Search">
81.     <input message="SearchRequest" />
82.     <output message="SearchResponse" />
83.   </operation>
84.   <operation name="Login">
85.     <input message="LoginRequest" />
86.     <output message="LoginResponse" />
87.   </operation>
88.   <operation name="AddToCart">
89.     <input message="AddToCartRequest" />
90.     <output message="CartResponse" />
91.   </operation>
92.   <operation name="UpdateQtyInCart">
93.     <input message="UpdateQtyInCartRequest" />
94.     <output message="CartResponse" />
95.   </operation>
96. </portType>
97.</definitions>
```

Fig. 1. A partial WSDL description for a bookstore web service (message, binding, port and service elements omitted)

In addition to constituting the largest portion of most WSDL documents, type definitions are often very complex, with highly nested structures and complex relationships between different parts of the definition. Although the WSDL specification does not specify a single standard notation for defining the types in the document, XML schemas have become the de facto standard. An XML schema defines a set of XML elements and describes the types of the XML elements using simple and complex type definitions [4]. A simple type represents a single value, while a complex type describes a composite type consisting of multiple nested xml elements and/or attributes. For example, our sample WSDL file describes an element, $SearchRequest$ (line 15), that is of an complex type with nested elements $sessionId$, $category$, $name$, $priceMin$, and $priceMax$ (lines 15-27). The type of the nested element $category$ (line 18) is defined as the simple type $Category$ which is a string with possible values "Databases", "Web Design", and "Programming" (lines 3-9) and can occur as few as 0 times (the minOccurs attribute) and as many as 1 time (the default value for the absent maxOccurs attribute).

Due to their size and complexity, type definitions often contain errors or are imprecise. In the example WSDL file, the $quantity$ element of the $AddToCartRequest$ should be defined as the more precise type positiveInteger rather than the type integer (line 53). Another common problem in the type definitions is incorrect or imprecise minOccurs on elements (in many places in both the Amazon and eBay WSDL files mentioned above elements defined as required by other documentation, implying minOccurs should be 1, have minOccurs=0).

The port type definition of the WSDL file describes the set of abstract operations for a service and specifies the types of the request and response for these operations. However, there is nothing in the WSDL specification that describes temporal dependencies between these operations. In our example bookstore service a StartSession operation must be performed before any other operation, and a Login operation must be performed before any of the cart operations. Often these temporal relationships can be inferred by considering the static type definitions (e.g. StartSession is the only operation that does not require a $sessionId$ and it returns a $sessionId$). At other times, there is nothing in the WSDL specification to indicate the temporal relationships between operations (as is the case with the Login operation in the example).

3 Methodology

Our approach builds on the WebAppSleuth methodology for characterizing web application interfaces [2, 3]. In prior work the WebAppSleuth methodology took an html form and generated inferences about how the underlying application used the variables and values in that form, identifying (for example) mandatory and optional variables and instances where certain combinations of variables were required. Several modifications were necessary, however, to adapt the methodology to the purposes of this work.

The WebAppSleuth process consists of four stages: (1) analysis of the WSDL file and solicitation of user input to determine input values for simple elements (the Analyzer); (2) generation of request sequences, including generation of the structure of the input messages for each request and assignment of input values to the simple elements

in the request (the Generator); (3) submission of requests to the server and classification of the responses (the Submitter); and (4) generation of suggested refinements to the WSDL and additional inferences (the Inferer). The following subsections provide details on each of these stages.

3.1 Determining Input Values for Simple Elements

In the WSDL file, operations are defined in the portType defintion and include input and output messages whose structure is defined in the type definition. For example, in Figure 1 lines 80-83 define the Search operation which has an input message whose structure is defined by the element $SearchRequest$ (lines 15-27) and an output message whose structure is defined by the element $SearchResponse$ (lines 28-41). The messages include *complex elements* (elements with complex types) that are compositions of other complex elements and *simple elements* (elements with simple types). In order to generate requests, we must first find suitable input values for the simple elements. The first stage of the WebAppSleuth methodology, the Analyzer is responsible for finding these values.

There are three possible sources for these input values. The WSDL file may specify a set of possible values for a simple type as an enumerated type (as in lines 4-8 of Figure 1). Alternatively, as requests are submitted to a web service, the responses to these requests may include values that can be used in other requests. Finally, in some cases it is necessary for the user of our methodology to provide input values for elements. The Analyzer identifies the elements that a user needs to provide input values for, and solicits input values from the user for these elements.

The basis for finding elements whose input values can be provided by other operations is through the use of name matching of simple elements with the same type. For example, in Figure 1, line 12 defines an element $sessionId$ of type string that is returned by the StartSession operation, and line 17 defines another $sessionId$ element of type string that is required as an input for the Search operation. Since these elements have the same name ($sessionId$) and type (string), we assume that $sessionId$s returned by the StartSession operation can be used as inputs for the Search operation.

The Analyzer attempts to minimize the number of elements for which it solicits input values from the user. Initially, it asks the user to provide input values for all simple elements used as an input to an operation that can never be produced as an output of another operation. Next the Analyzer solicits additional required input values. The function GatherUserInputs (Algorithm 1) is called with the set $operations$ containing all operations defined for the service and the set $elements$ containing all the simple elements for which the user has already provided inputs and all elements for which the WSDL file enumerates input values. The innermost loop (lines 5-10) considers each of the operations in $operations$ and for each, determines whether it is executable (it is if there is a potential source for each of its required input values, from prior user-provided inputs, enumerated values in the WSDL file, or other operations that have already been found to be executable). If an operation is executable it is removed from $operations$ and all of the elements which it could return in its output message as defined in the WSDL are added to the set $elements$. The while loop in lines 3-12 repeats until no more executable operations are found. Then, if $operations$ is not empty, the user is

Algorithm 1 GatherUserInputs(Set $operations$, Set $elements$)

```
while operations ≠ ∅ do
  boolean changed = TRUE
  while changed do
    changed = FALSE
    for all Operation op ∈ operations do
      if elements.containsAll(op.requiredElements) then
        operations.remove(op)
        elements.addAll(op.outputElements)
        changed = TRUE
      end if
    end for
  end while
  prompt user to provide inputs
end while
```

asked to provide input values for all elements of at least one operation in $operations$ that are not in $elements$. This process continues until $operations$ is empty, at which point we should have sufficient input values to generate requests for all operations.

In the case of the Bookstore example the process begins by requiring the user to provide inputs for $login$, $password$, $priceMin$ and $priceMax$ as none of these appear in the output message of any operations. The WSDL file also provides enumerated values for $category$. Thus, GatherUserInputs is called with $elements = \{login, password, priceMin, priceMax, category\}$ and $operations = \{$StartSession, Search, Login, AddToCart, UpdateQuantityInCart$\}$. The inner loop begins with the operation StartSession. This operation does not require any input values, therefore it is removed from $operations$ and $sessionId$ is added to $elements$. Next, Search is processed. The only element it requires is $sessionId$, which is already in $elements$, therefore it is removed from $operations$ and the elements $itemId$ and $name$ are added to $elements$. Login is processed similarly. At this point, no further operations in $operations$ are executable with the given inputs, so the Analyzer prompts the user to either provide inputs for $quantity$ for AddToCart or $quantity$ and $orderId$ for UpdateQuantityInCart. If the user provides input values for $quantity$, then AddToCart becomes executable and provides inputs for $orderId$ which makes UpdateQuantityInCart executable, leaving $operations$ empty and allowing the algorithm to terminate.

The set of input values for simple elements collected by the Analyzer from the WSDL file and the user are passed to the next stage of the WebAppSleuth methodology, the Generator.

3.2 Generating Request Sequences

The Generator incrementally generates sequences of requests to be submitted to the web service. These sequences of requests are generated one request at a time, with each generated request being passed to the Submitter, and the values of simple elements in the response being used for further requests in the request sequence. Requests in a

sequence are generated until a request returns an invalid response or a maximum request length specified by the user of the methodology is reached.

The generation of a single request in a sequence can be broken down into three types of decisions: selecting the operation, selecting the number of occurrences of each element nested in a complex type in the input message and selecting values for the simple elements of the input message. The Generator includes a submodule, the Chooser, that is responsible for making each of these decisions. Currently, the Chooser makes each of these decisions randomly from the range of available options.

The first type of decision, selecting the operation, begins with the Generator determining the set of executable operations. An operation is executable if for all required simple elements (simple elements minOccurs > 0 that are not nested within any complex elements with minOccurs $= 0$) we have an input value, either provided by the Analyzer or returned in an earlier request in the same sequence. The Chooser then chooses randomly from ones of these requests.

The second type of decision that must be made is determining the number of occurrences for elements nested in complex types in the input message. Each element nested in a complex type has a minimum and maximum number of occurrences defined. The Generator uses this as the range of possible number of occurrences to be passed to the Chooser to be selected from, with three possible exceptions. First, if an element has minOccurs $= 0$ then it is possible there are no currently available input values, either for that element (if it has a simple element) or for one of the elements nested inside of it with minOccurs > 0 (if it is a complex element). If this is the case, then zero occurrences of that element are included in the request. Second, since elements can have an unbounded number of maximum occurrences, it is necessary to limit the range of occurrences that can be selected from. The Generator includes a parameter that specifies the maximum range of occurrences that will be selected from and adjusts the maximum number of occurrences accordingly (e.g., if the maximum range is 3, and the minOccurs of an element is 0 and the maxOccurs is unbounded, then the Generator would select the possibilities 0, 1, and 2 occurrences for the Chooser to select from). Third, one common error that we would like to detect in web applications involves cases where the WSDL file specifies that minOccurs is greater than 0 when it should be 0. Therefore, the Generator automatically always includes 0 occurrences in the range of number of occurrences that the Chooser selects from. The Chooser then randomly selects from this set of values, and the Generator adds the appropriate number of that element to the input message of the request.

The third type of decision is determining the actual input values for any simple elements included in the input message. As each occurrence of a simple element is added to the input message by the Generator, the Generator determines the set of possible input values for that simple element from the pool of available input values, including both values from the Analyzer and values return by earlier requests in the request sequence. This set is passed to the Chooser, which randomly selects one of these values, which the Generator then inserts into the appropriate location in the input message,

After a request has been generated, it is submitted (see Section 3.3) and the simple element values returned for that request are added to the pool of input values available for use in further requests in the sequence.

To illustrate our generation methodology, we walk through the generation of a length two request sequence for the Bookstore example. To begin with, remember that the Analyzer required the user to provide inputs for $login$, $password$, $priceMin$, $priceMax$, and $quantity$ (Section 3.1) and there is a set of enumerated values for $category$ defined in the WSDL file in Figure 1. For the first request, the only operation that is executable is StartSession, so it is selected. The StartSession operation has no input message, so no further work needs to be done. This request is passed to the Submitter, which returns a value for $sessionId$. The returned value for $sessionId$ is included in the input value pool.

Now the second request needs to be generated. At this point, StartSession, Login, and Search are all executable, and the Chooser randomly selects from these three. Assume that Search is randomly selected. Consider the definition of the input message for Search in lines 15 - 27 of Figure 1. According to the WSDL file, $sessionId$ has minOccurs and maxOccurs both equal to 1. Therefore, the Chooser is asked to select between 0 and 1 (since 0 is always included in the range to be selected from). Assume that it chooses 1. According to the WSDL, all of the other elements can occur 0 or 1 times. However, since there are no input values for $name$, it must occur 0 times in the request. Since we have input values for $category$, $priceMin$, and $priceMax$ they can occur 0 or 1 times in the request. Assume the Chooser selects 1 occurrence for $category$ and 0 occurrences for $priceMin$ and $priceMax$. Finally, input values must be selected for each of the simple elements included in the input message. For $sessionId$ there is only one possible input value, the one returned by the earlier StartSession request, so it is selected. For $category$ there are three input values defined in the WSDL file, and one of these is randomly selected, and the resulting completed request is passed to the Submitter to be submitted.

Assume that $category$ is included, and the other two input values are excluded. Since we have no input values in the pool of input values for $name$, we exclude it from the input message. Finally we select input values for each of the simple elements in the input message. We have one possible value for $sessionId$, so we select that. For $category$ we have three possible values defined in lines 3 - 9 of the WSDL file that we can randomly select from. One of these is randomly selected and the resulting request is submitted.

3.3 Submitting Requests and Classifying Responses

The Submitter is responsible for actually submitting requests to the web service and for classifying the responses from the web service. To submit requests to the web service we have built submission modules that work with Sun's JAX-WS tools and with the Apache Software Foundation's Axis tools.These tools automatically build Java libraries for accessing web services from WSDL declaration files. Using Java Reflection, we are able to make use of these libraries to submit requests to the web service. The response message is then extracted into a format usable by the Generator and Inferer modules. Since the generated libraries return the responses as complex objects whose structure depends on which tool was used to generate the library, we again use Java Reflection to extract the message into a common format used by the other components.

The Classifier submodule of the Submitter is responsible for classifying responses as valid or invalid. This requires specific checks depending on the web service, as the mechanism for reporting errors varies depending on the web service implementation. For example, Amazon's e-Commerce service uses the SOAP exception model to indicate problems in requests, while eBay's service includes a list of Error elements in the response message. The classification serves two purposes. First, if a response is classified as invalid, the Generator stops generating new requests in the current request sequence, and begins a new request sequence. Second, some of the inference algorithms in the Inferer use these classifications.

3.4 Generating Inferences

The Inferer takes the set of submitted request sequences with their corresponding responses and classifications and uses this information to generate inferences. These inferences take the form of suggestions for changing the WSDL file or statements about the behavior the web service. Currently, we have three different inference algorithms implemented. The first two of these algorithms, "number of occurrences" and "simple element types", were chosen based on initial inspections of real WSDL files that showed that certain types of errors were common. The third algorithm, "precedence", was chosen as a simple temporal relationship that WSDL files do not include support for.

Number of Occurrences. Our first inference algorithm attempts to correct errors in the minOccurs of elements defined in the WSDL declaration, specifically finding elements with minOccurs $= 1$ that should be 0 or minOccurs $= 0$ that should be 1. These two cases were chosen because our initial examination of the WSDL file and html documentation for the Amazon E-Commerce service found inconsistencies in the definitions of the minimum occurrences of elements.

The inference algorithm used in this case is based on the algorithm for inferring mandatory and optional variables presented in [3]. For each element e nested in a complex type t used for an input message as defined in the WSDL specification, four boolean variables, $absentValid$, $absentInvalid$, $presentValid$, and $presentInvalid$, are created. Then for each request that is submitted, if that request included t, the appropriate boolean variable is set to true depending on whether the response was valid or invalid, and whether that request included any elements e. After processing all submitted requests, if $absentInvalid$ is true, $absentValid$ is false and $presentValid$ is true, minOccurs should be 1. If $absentValid$ is true, minOccurs should be 0. In all other cases, there is not enough information to determine what minOccurs should be. Any cases where the calculated value of minOccurs differs from the value defined in the WSDL are reported to the user as possible errors in the WSDL file.

Simple Element Types. XML schema definitions can include a variety of types for simple elements. Although the XML schema standard does not do so, these types can be arranged into a simple type hierarchy as shown in Figure 2. Due to the variety of possible types, and the fact that all types are transmitted as strings, it is often the case

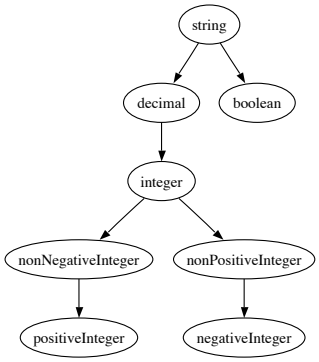

Fig. 2. (partial) XML Schema simple type hierarchy

that the WSDL file uses a more general type than possible. For example, an early version of the WSDL file for Amazon's E-Commerce service used the type "string" for nearly all simple elements. Additionally, in some cases simple elements with the same name but in different operations have different types. Although it is not always the case that these elements should have the same type, in many cases we expect that they should. Therefore we would like to find cases where the types can be improved in the WDSL file.

To do this, for each value returned in a simple element of a valid response, the most specific XML schema type that the returned value matches is determined. Since all values are returned as strings in the XML document, various Java parse functions and constructors are used to make this determination. For example, if a call to new BigInteger(value) succeeds, and the type is positive, that value has a type of positiveInteger. To facilitate the selection of the correct type, we added a psuedo-type to the hierarchy for the value zero as a subtype of both nonPositiveInteger and nonNegativeInteger. After collecting the set of all actual returned types, these types are generalized to the most specific type in the hierarchy that is a supertype of all actual returned types. For example, if an element has values "0", "1", and "50", the set of types is {zero, positiveInteger}, and this generalizes to type nonNegativeInteger. If a new value "5.27" is received for this element, then decimal is be added to the set of types, which generalizes to type decimal. The generalized type is compared to all of the declared types for all elements with the same name, and if it is a subtype of all of the declared types, it is reported as a possible improvement to the WSDL file.

Precedence. Our third inference algorithm attempts to detect dependencies between different operations. There are two classes of dependencies that we could detect. The first class are data dependencies where a value for an element that is returned by some operation is used as an input value for another operation. These dependencies can be determined by static analysis of the WSDL. The second class of dependency involves hidden dependencies, where an operation changes some state on the server that affects

Table 1. Details about objects

Object	Lines in WSDL	Operations in WSDL	Operations Analyzed	Number of Request Sequences
Amazon E-Commerce Service	3,244	19	19	70,000
eBay Trading Web Service	90,450	132	102	300,000

the behavior of a later operation. It is this class of dependencies that we are interested in here. In particular we are concerned with the case in which, for some operation to succeed (have a valid response), another operation must be successfully submitted earlier in the request sequence.

To find these precedence relationships, for each pair of operations O_i and O_j defined for the web service, four boolean variables, $precedesValid$, $doesntPrecedeValid$, $precedesInvalid$, and $doesntPrecedeInvalid$ are created. Then each time O_j is submitted, the appropriate boolean is set to true depending on whether O_i was submitted earlier in the request sequence and whether the response to O_j is valid. After submitting all requests, the relationship "O_i must precede O_j" is reported if $precedesValid$ is true, $doesntPrecedeValid$ is false and $doesntPrecedeInvalid$ is true.

4 Evaluation

To evaluate our methodology we performed two case studies on production web services, one provided by Amazon[3] and the other provided by eBay[4]. Columns 2 and 3 of Table 1 provide some basic details relating to the size of the web services. For eBay, there were a number of operations that required additional access privileges to execute. These were removed from the set of operations that we considered, leaving 102 of the 132 total operations. For each object we submitted a number of request sequences (column 5 of Table 1). The number of request sequences submitted was determined by the complexity of the web service (eBay has a much larger number of operations, therefore more request sequences are required), the rate at which we were able to submit requests (eBay processed requests significantly faster than Amazon) and the amount of time available to collect the data (we collected these requests over an approximately one week period).

There are several areas where user input is required in the methodology. First the analyzer requires the user to provide inputs for several simple elements. For many of these, the documentation for the service provided a list of suitable values (e.g. Amazon has a sort element that has a list of possible values defined in the documentation). For others, we produced values that we considered reasonable based on our knowledge of the service. Second, the generator requires us to specify the maximum sequence length and maximum range of occurrences for elements. For the maximum sequence length,

[3] `http://www.amazon.com/E-Commerce-Service-AWS-home-page/b?node=12738641`

[4] `http://developer.ebay.com/products/trading/`

Table 2. Summary of results

Inference Type	Amazon	eBay
minOccurs = 0 should be 1	7	4
minOccurs = 1 should be 0	0	6
same element name, different types	0	1
imprecise types	1	41
non-static precedences	1	0

ten was found to be sufficient to produce request sequences that included all operations. For the maximum range of occurrences for elements we chose three. Finally we need to provide a classification criteria for responses from each of the services As mentioned in Section 3.3 eBay returns error elements detailing errors in the request while Amazon uses the SOAP exception model to return errors.

Table 2 provides an overview of the results from applying WebAppSleuth to these services. On Amazon we found seven elements that were defined as having minOccurs = 0 where we were unable to submit a valid request with that element missing. Of those, five are defined in developer documentation provided by Amazon as being required, indicating that these five elements are defined incorrectly in the WSDL file. The other two cases are defined in other documentation as being required under some circumstances, but the minOccurs of these elements should not be changed. On eBay we found four elements that were defined as optional that should be mandatory. Developer documentation suggests that one of these elements is required and the minOccurs should be changed in the WSDL. The other three are defined as being required under certain conditions, but the minOccurs of these elements should not be changed.

On eBay we found six elements that were defined with minOccurs = 1 where it should have been 0. Of these, four are specified to be optional in other documentation, indicating an error in the WSDL. The other two are specified to be mandatory in other documentation, and further examination indicates that there may be missing error checks in the web service code itself.

On eBay we found one element name, $Version$, that had two different types, string and int, indicated in different portions of the WSDL. Since the value of this element was always a positiveInteger, the type of the element should be changed in the WSDL.

On Amazon we found one element, $IsValid$, that was defined as type string and should have been defined as boolean. This agrees with the possible values for the element defined in other documentation. On eBay we found nine elements that were defined with the type string that should have been defined as some type of integer. The remaining 32 elements that were found to have imprecise types were all defined as int in the WSDL and could have been defined as positiveInteger, nonNegativeInteger, or nonPositiveInteger based on the returned values. Since int represents a 32-bit integer and each of these other integer types are arbitrary precision, these elements should be defined in the WSDL as int with appropriate range constraints placed on them.

Finally, on Amazon we found one precedence relationship beyond the statically determined ones. This precedence was between a ListSearch operation that would search

for wish lists, baby registries and wedding registries based on criteria such as the name and location of the person who posted the list, and ListLookup which would look up a list based on $ListId$. This precedence represents a data dependency between ListSearch and ListLookup through the $ListId$ variable. It was not found statically because $ListId$ was defined with minOccurs $= 0$ in the ListLookup operation when it should have been defined with minOccurs $= 1$ (this was one of the minOccurs errors mentioned earlier).

Overall, we found that we were able to accurately suggest improvements to the minOccurs and types of several elements in both applications. This suggests the number of occurrences and simple element types inference algorithms can be used to help improve the WSDL files for applications. However, the single precedence we found was not particularly useful, either because these types of relationships do not exist in the web services we used or because the request sequences we submitted were not sufficient to detect these relationships.

5 Related Work

The first step of our methodology is the static generation of the relationship between input and output values of various WSDL operations that is similar to the Jungloids described in [5] or the relationships found as the first step of the WSDL-based test generation strategy proposed in [6].

Our request generation strategy is similar to random or exhaustive test generation strategies proposed for object-oriented applications [7–10]. The primary differences between these methods and our work is domain (object-oriented applications versus web services) and the intended use of the inputs (testing versus generating inferences). One methodology [11] explicitly uses automatically generated inputs to drive the process of generating properties. This methodology uses a form of bounded exhaustive generation of expressions consisting of sequences of method calls on Java objects. All of these methodologies that perform input generation for object-oriented applications are implemented for Java and take advantage of Java's type system to ease the problem of generating well-formed inputs. The XML schema based types used in WSDL files are much less rigorously defined, and making the problem of relating the types of inputs and outputs more difficult and less reliable.

XPT is a methodology for systematically generating XML inputs to applications based on a schema [12]. XPT uses a form of bounded exhaustive generation to generate a wide variety of XML documents and then uses heuristics based on category-partition testing [13] to reduce the number of test cases. It would be possible to adapt XPT for use in our methodology at the level of determining the structure of the XML requests, but this would need to be modified to take into account the set of available input values at each step.

There has been a great deal of work in developing techniques for dynamically inferring properties about programs [11, 14–21]. These methodologies work by instrumenting the program being examined. They then execute the program under some test suite, collecting traces of program events and states of interest. These traces are analyzed to produce a set of properties. These properties can be in the form of invariants on the states of variables (e.g. variable $v > 0$) or temporal relationships between events (e.g.

an open event must be followed by a close event). These methodologies differ from our approach in two major aspects. First, most assume that an engineer can instrument the code they are interested in examining. Since we are working with web services and applications that are accessed via a network, we assume that we can not instrument the system, instead we only have the input and outputs to examine. Second, all of these methods (except [11]) require that there is an existing set of inputs. Our methodology generates these inputs automatically with minimal interaction from the user.

6 Conclusion

In this work we have presented a methodology for automatically refining and augmenting WSDL files. This methodology collects static information from the WSDL file to drive a request generation process, and automatically submits large numbers of generated requests to the web service to collect dynamic information about the behavior of the web service. We have applied our methodology to two commercial web services, and found a number of improvements that could be made to the WSDL files.

In future work we intend to improve our methodology in various ways. In earlier work on web applications we have had a great deal of success using the inference generation process to guide the request generation process [3]. We intend to adapt that methodology to the process of generating requests for web services. There are also several additional kinds of inferences that we would like to generate. For example XML schemas can define an either-or relationship between elements, however this feature appears to be rarely used in actual WSDL files. We believe that we can adapt the at-least-one-of inference described in [3] to automatically detect case where these either-or relationships could be used in web services.

Additionally, we would like to perform additional studies to evaluate different aspects of our methodology. Currently we make several assumptions that can directly impact the ability of our methodology to provide meaningful suggestions to the user. For example, we assume that in all cases where two elements have the same name and same type they represent the same abstract type in the application. This can limit the ability of our methodology to generate appropriate request sequences. We also make several assumptions about the ability of the users of our methodology to make appropriate choices at several different points in the process, such as providing appropriate inputs or developing a reasonable method for differentiating valid from invalid responses. Additional controlled studies can be designed to evaluate the correctness of these assumptions. We would also like to apply our methodology to additional web services to determine how it generalizes to other applications.

References

1. Christensen, E., Curbera, F., Meredith, G., Weerawarana, S.: Web services description language (WSDL) 1.1. `http://www.w3.org/TR/wsdl` (March 2001)
2. Elbaum, S., Chilakamarri, K.R., Fisher II, M., Rothermel, G.: Web application characterization through directed requests. In: Proceedings of the 4th International Workshop on Dynamic Analysis. (May 2006) 46–56

3. Fisher II, M., Elbaum, S., Rothermel, G.: Dynamic characterization of web application interfaces. In Dwyer, M., Lopes, A., eds.: Fundamental Approaches to Software Engineering. Volume 4422/2007 of Lecture Notes in Computer Science., Springer (March 2007) 260–275
4. Fallside, D., Walmsley, P.: XML schema part 0: Primer second edition. `http://www.w3.org/TR/xmlschema-0` (October 2004)
5. Mandelin, D., Xu, L., Bodik, R., Kimmelman, D.: Jungloid mining: Helping to navigate the API jungle. In: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, ACM Press (2005)
6. Bai, X., Dong, W., Tsai, W.T., Chen, Y.: WSDL-based automatic test case generation for web services testing. In: Proceedings of the IEEE International Workshop on Service-Oriented System Engineering. (2005)
7. Visser, W., Pasareanu, C., Khurshid, S.: Test input generation with Java PathFinder. In: Proceedings of the ACM International Symposium on Software Testing and Analysis. (July 2004)
8. Pacheco, C., Ernst, M.: Eclat: Automatic generation and classification of test inputs. In: Proceedings of the European Conference on Object-Oriented Programming. (2005) 504–527
9. Xie, T., Notkin, D.: Tool-assisted unit-test generation and selection based on operational abstractions. Automated Software Engineering **13**(3) (July 2006) 345–371
10. Artzi, S., Ernst, M., Kieżun, A., Pacheco, C., Perkins, J.: Finding the needles in the haystack: Generating legal test inputs for object-oriented programs. In: Proceedings of the Workshop on Model-Based Testing and Object-Oriented Systems. (October 2006)
11. Henkel, J., Reichenbach, C., Diwan, A.: Discovering documentation for java container classes. IEEE Transactions on Software Engineering **33**(8) (August 2007) 526–543
12. Bertolino, A., Gao, J., Marchetti, E., Polini, A.: Systematic generation of xml instances to test complex software applications. In: Proceedings of the International Workshop on Rapid Integration of Software Engineering Techniques. Volume 4401. (September 2006) 114–129
13. Ostrand, T., Balcer, M.: The category-partition method for specifying and generating functional tests. Communications of the ACM **31**(6) (1988) 676–686
14. Ernst, M., Cockrell, J., Griswold, W., Notkin, D.: Dynamically discovering likely program invariants to support program evolution. In: Proceedings of the International Conference on Software Engineering. (May 1999) 213–224
15. Guo, P., Perkins, J., McCamant, S., Ernst, M.: Dynamic inference of abstract types. In: Proceedings of the ACM International Symposium on Software Testing and Analysis. (July 2006) 255–265
16. Perkins, J., Ernst, M.: Efficient incremental algorithms for dynamic detection of likely invariants. In: Proceedings of the ACM Symposium on Foundations of Software Engineering. (November 2004) 22–32
17. Ammons, G., Bodìk, R., Larus, J.: Mining specifications. In: Proceedings of the ACM Symposium on Principles of Programming Languages. (2002) 4–16
18. Whaley, J., Martin, M., Lam, M.: Automatic extraction of object-oriented component interfaces. In: Proceedings of the ACM International Symposium on Software Testing and Analysis. (2002) 218–228
19. Yang, J., Evans, D., Bhardwaj, D., Bhat, T., Das, M.: Perracotta: Mining temporal api rules from imperfect traces. In: Proceedings of the International Conference on Software Engineering. (May 2006) 282–291
20. Savage, S., Burrows, M., Nelson, G., Sobalvarro, P., Anderson, T.: Eraser: A dynamic data race detector for multithreaded programs. ACM Transactions on Computer Systems **15**(4) (November 1997) 391–411
21. Hangal, S., Lam, M.: Tracking down software bugs using automatic anomaly detection. In: Proceedings of the International Conference on Software Engineering. (May 2002) 291–301