An Automated Analysis Methodology to Detect Inconsistencies in Web Services with WSDL Interfaces

Marc Fisher II*,†, Sebastian Elbaum‡ and Gregg Rothermel§

SUMMARY

Web Service Definition Language (WSDL) is being increasingly used to specify web service interfaces. Specifications of this type, however, are often incomplete or imprecise, which can create difficulties for client developers who rely on the WSDL files. To address this problem a semi-automated methodology that probes a web service with semi-automatically generated inputs and analyzes the resulting outputs is presented. The results of the analysis are compared to the original WSDL file and differences between the observed behavior of the service and the WSDL specifications are reported to the user. The methodology is applied in two case studies involving two popular commercial (Amazon and eBay) web services. The results show that the methodology can scale, and that it can uncover problems in the WSDL files that may impact a large number of clients.

Received August 2009

KEY WORDS: Dynamic analysis; validation; WSDL; web services

1. Introduction

Many companies provide access to their services and products through web services. 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,

*Correspondence to: University of Memphis, 209 Dunn Hall, Memphis, TN 38152-3240
†Email: marc.fisher@memphis.edu
‡Email: elbaum@cse.unl.edu
§Email: grother@cse.unl.edu

Contract/grant sponsor: NSF CAREER Award 0347518
Contract/grant sponsor: The EUSES Consortium through NSF-ITR 0325273 and NSF-ITR 032486
Contract/grant sponsor: ARO DURIP award W911NF-04-1-0104

some businesses that target consumers, such as Amazon or eBay, also provide access to their products or services via web services. By doing so, these companies allow outside parties to utilize or resell their services or products. According to the companies that provide these services, web services are a significant part of their business. For example, eBay claims that third party applications account for over 25% of their listings [10] and Amazon claims that over 200,000 developers, start-ups, and Fortune 1000 companies use their technologies [1].

Typically, Web Service Definition Language (WSDL) files are used to provide a partial specification of the interface to a web service [7]. These specifications focus on the data-types of parameters and returned values of the application. Developers of clients to the web services are expected to use the information in the WSDL and in informal documentation to build applications that make use of the web services. However, the WSDL files and informal documentation often contain errors or are imprecise. For example, an early version of what became Amazon's Advertising API used strings for every element, including those such as quantities or prices that could be more precisely specified as numeric types. Additionally, recent versions of eBay's Trading Web Service include a number of elements where the informal documentation and the WSDL file disagree on whether they are required or not, indicating that one of these sources of information is incorrect.

In order to develop robust and reliable clients for web services, users of these services need accurate information about the expected inputs. However, the client developers do not have access to the underlying implementation of the web service, so identifying problems in the WSDL or documentation can be difficult. Therefore an existing analysis of web application interfaces has been extended to find anomalies in the WSDL files describing web services.

Prior work presented the WebAppSleuth methodology for characterizing *specialized search engines*, web applications with a single HTML form and corresponding form handler that allow users to perform web-based queries on structured data [11, 15]. This earlier version of WebAppSleuth performed an automated analysis that began by examining the interface described by the HTML search form. It semi-automatically constructed and submitted inputs to the form handler, and analyzed the responses to infer properties that describe the relationships between input variables and values and the corresponding responses. These inferred properties were then examined to identify problems in the analyzed web application. WebAppSleuth was applied to six production search engines and found anomalous behavior on five of these applications.

This work provides two major contributions beyond the prior work on WebAppSleuth.

- WebAppSleuth has been extended to work with web applications outside of the domain of specialized search engines. Specifically, WebAppSleuth now supports any web service described by a standard WSDL file. To support arbitrary web services, WebAppSleuth's request generation and submission components were modified and new property inference algorithms were created. The inferred properties are then compared with the WSDL specification to semi-automatically identify problems in the WSDL specification or the web service.
- WebAppSleuth is applied to two commercial web services, Amazon's Advertising web service and eBay's Trading web service. These two web services are significantly more complex than the applications used in prior work, supporting 23 and 134 operations respectively instead of the single search operation supported by specialized search engines. A variety of inconsistencies between the WSDL specifications and the behavior of the web services are presented.

In addition to helping client developers understand a web service, the information provided by WebAppSleuth can be used for other applications. For example, the problems identified by WebAppSleuth can be used to modify the WSDL specifications, which will allow tools such as Sun's JAX-WS [27] or Apache's Axis [28] to produce libraries for accessing web services with better input validation and more efficient parsing of responses. Better WSDL specifications can also help engineers understand and maintain web services by identifying previously undocumented constraints on the inputs or outputs of the service or by identifying potential problems in the service that should be resolved.

The remainder of this paper presents this work, as follows. Section 2 provides background on web services and WSDL. Section 3 presents the methodology, providing algorithms and discussion of their operation. Section 4 presents the case studies and results. Section 5 provides details about related work, and Section 6 concludes and discusses future work.

2. Background

A web service consists of a set of operations. A client executes a web service by constructing a request message for a particular operation and sending that message to the server. The server then responds with a response message that is parsed by the client to extract the information that it considers important. The Web Service Definition Language (WSDL) provides a standard for defining the set of operations and the structure of the input and output messages for these operations.

Figure 1 is a partial WSDL file describing an example bookstore web service that conforms to the WSDL 1.1 standard [7].[†] 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 2-79) and *message* definitions (omitted) describe the overall structure of request and response messages. The *port type* definition (lines 104-124) describes a set of abstract operations and identifies the messages that are used for the input and output of each operation, while the *bindings* (omitted) define the concrete protocol for the abstract operations of a port type. A *port* definition (omitted) specifies the address for a particular binding and a *service* definition is used to aggregate a set of related ports.

Although all of these definitions are important when describing a web service, in this work the focus is on type definitions as they are the largest and most complex part of the document. For example, the WSDL file describing Amazon's Advertising service[‡] is 3,244 lines long, with a type definition that

[†]While this section and the remainder of the paper uses the WSDL 1.1 standard, only minimal changes would be required to adapt the discussion to WSDL 2.0. In particular, WSDL 2.0 renames the *port type* definition to *interface*, renames the *port* definition to *endpoint* and removes the *message* definition, instead having the *interface* definition directly refer to definitions within the *types* definition.

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

```
<definitions name=“Bookstore”>
 <types><schema>
  <simpleType name=“Category”><restriction base=“string”>
   <enumeration value=“Databases”/>
   <enumeration value=“Web Design” />
   <enumeration value=“Programming” />
  </restriction></simpleType>
  <element name=“StartSessionResponse”><complexType><all>
   <element name=“sessionId” type=“string” />
  </all></complexType></element>
  <element name=“SearchRequest”><complexType><all>
   <element name=“sessionId” type=“string” />
   <element name=“category” type=“Category” minOccurs=“0” />
   <element name=“name” type=“string” minOccurs=“0” />
   <element name=“priceMin” type=“decimal” minOccurs=“0” />
   <element name=“priceMax” type=“decimal” minOccurs=“0” />
  </all></complexType></element>
          (32 lines omitted)
  <element name=“AddToCartRequest”><complexType><all>
      <element name=“sessionId” type=“string” />
      <element name=“itemId” type=“integer” />
      <element name=“quantity” type=“integer” />
  </all></complexType></element>
          (24 lines omitted)
 </schema></types>
          (24 lines omitted)
 <portType name=“BookstoreAPI”>
  <operation name=“StartSession”>
   <output message=“StartSessionResponse” />
  </operation>
  <operation name=“Search”>
   <input message=“SearchRequest” />
   <output message=“SearchResponse” />
  </operation>
          (12 lines omitted)
 </portType>
          (53 lines omitted)
</definitions>
```

Figure 1. Bookstore Web Service Description

is 2,850 lines long, 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.

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

[§]Originally downloaded from `http://developer.ebay.com/webservices/515/eBaySvc.wsdl`. Available by request from the first author.

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 [14]. A simple type represents a single value (similar to a primitive type in most programming languages), while a complex type describes a composite type consisting of multiple nested XML elements and/or attributes (similar to a structure or array depending on the min- and maxOccurs of the contained elements). For example, the sample WSDL file in Figure 1 describes an element, `SearchRequest`, that has a complex type with nested elements `sessionId`, `category`, `name`, `priceMin`, and `priceMax` (lines 11-17). The type of the nested element `category` (line 13) is defined as the simple type `Category`, which is a string with possible values "Databases", "Web Design", and "Programming" (lines 3-7) and can occur as few as zero times (the *minOccurs* attribute) and at most one time (the default value for the absent *maxOccurs* attribute).

Due to their size and complexity, type definitions often contain errors or are imprecise. These errors and sources of imprecision can directly impact the quality of code generated from the WSDL file or can mislead developers relying on the WSDL file to understand the web service. For instance, 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* attributes on elements (e.g., 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).

In general, client applications interact with a web service through a sequence of operations. For the Bookstore service, the expected first operation is a `StartSession` request to get a `sessionId`. This `sessionId` is then used in later requests so that the service can tie together a sequence of related requests.

3. Methodology

This work builds on the WebAppSleuth methodology for characterizing web application interfaces [15]. The previous version of WebAppSleuth targeted a class of web applications labeled *specialized search engines* (a search engine for structured data with a complex set of inputs, e.g., a flight search that takes departure and arrival dates and locations and returns matching flights). In this work WebAppSleuth is extended to support arbitrary web services described by WSDL files. To do this, a new request generation methodology that supports the more complex inputs possible for web services and that allows results from earlier requests to be used in later requests is developed. Additionally, a new class of properties, the type of simple elements, is added.

Figure 2 provides a high-level view of the WebAppSleuth methodology. 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 (Analyzer);
2. generation of requests, including generation of the structure of the input messages for each request and assignment of input values to the simple elements in the request (Generator and Chooser);

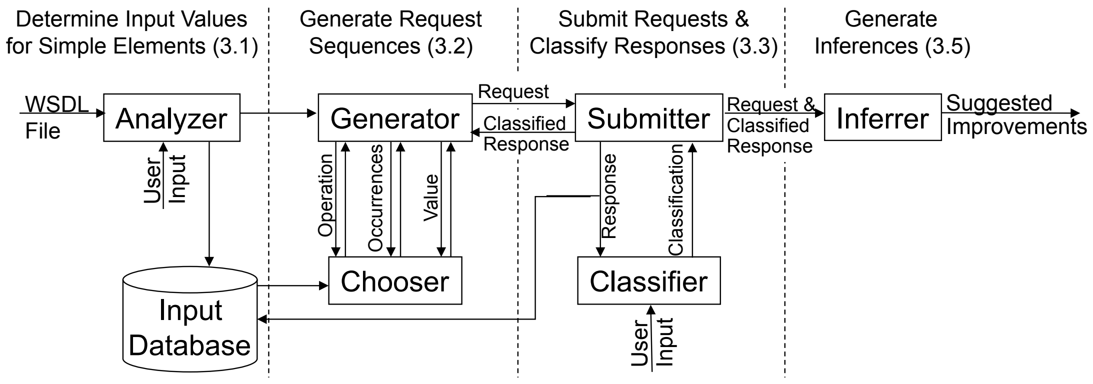

Figure 2. Architecture of the WebAppSleuth methodology

3. submission of requests to the server and classification of the responses as valid or invalid (Submitter and Classifier);
4. generation of inferred properties from the requests, responses, and classifications (Inferrer).

The following sections provide further details about these stages.

3.1. Determining Input Values for Elements

In a WSDL file, operations are defined in the portType definition and include input and output messages whose structures are defined in the type definition. For example, in Figure 1, lines 108-111 define the `Search` operation which has an input message whose structure is defined by the element `SearchRequest` (lines 11-17) and an output message whose structure is defined by the element `SearchResponse` (omitted). The messages include *complex elements* (elements with complex types) that are compositions of other complex elements and *simple elements* (elements with simple types). To generate requests, suitable input values for simple element must be identified. The first stage of the WebAppSleuth methodology, the *Analyzer*, is responsible for finding these values. While it would be possible to randomly generate input values for the simple elements, in many cases this will be ineffective. For example, in the Bookstore application, the `Login` operation requires a `username` and `password`. Random generation of strings are unlikely to successfully produce a valid `username` or `password`. Therefore the Analyzer identifies potential sources of values for different simple elements and queries the user for input values for those elements that do not have alternative input values.

In a sequence of related web service operations, operations will tend to use values output from previous operations as inputs. In general, it is impossible to tell what output values can be used as inputs, but a heuristic proposed by Bai *et al.* [4] uses the names and types of simple elements defined in the requests and responses of operations. Specifically, if an operation returns a simple element with name n and type t it is a candidate to be used as an input value for any element with name n and type t. This heuristic is applied to use outputs from earlier operations as inputs to later operations. This also decreases the number of simple input elements in the WSDL file for which the user of WebAppSleuth must provide input values.

Element	Class	Input Values
`category`	Enumerated	databases, web design, programming
`priceMin`	Missing	**0**, **20**
`priceMax`	Missing	**0**, **20**
`sessionID`	Returned	`StartSession`
`itemId`	Returned	`Search`
`quantity`	Returned	**0**, **1**, **2**, `AddToCart`, `UpdateQuantityInCart`
Bold values are provided by user, `Names` indicate that values can come from an operation		

Table I. Input values for simple elements

This heuristic, however, has two potential limitations. First, it is possible for it to identify two different simple elements as being compatible. This is most likely to occur when there exist elements with generic names such as `Id` or `Value`. Based on examination of several WSDL files, examples of this problem appear to be rare. Second, the heuristic may fail to link two simple elements that should be linked together. An example of this occurred in the eBay web service where a hierarchy of auction categories are defined. Categories can be looked up using a `CategoryId` element. Each category also includes a `CategoryParent` element that defines the id of the parent category. For cases like this mechanism is provided where the user can specify that values returned by `CategoryParent` can be used as inputs for `CategoryId`.

To determine whether the user needs to provide input values for a simple element, WebAppSleuth categorizes simple elements into three classes based on their usage in the WSDL file and using the heuristic defined above. Table I summarizes the categorization and available input values for the Bookstore service.

Enumerated input elements are elements with enumerated values defined in the WSDL file. The `category` input to `SearchRequest` is an example of an enumerated element. The WSDL Analyzer adds the values defined in the WSDL file to the set of inputs for that element.

Missing input elements are elements that are never returned by any operations. The `priceMin` and `priceMax` inputs to `SearchRequest` are examples of missing elements. The Analyzer prompts the user to provide inputs for all missing elements.

Returned input elements are elements that are returned by some operation. Some of these elements, such as `sessionId` (used by all operations except `StartSession` and returned by `StartSession`) and `itemId` (used by `AddToCart` and returned by `Search`) do not require the user to provide input values for them. Other elements, such as `quantity` (used by `AddToCart` and `UpdateQuantityInCart` and returned by the same operations), require input values to be provided because there is no way for the service to return values for these elements without first having submitted requests that include these elements. Algorithm 1 presents GetInputs, the procedure that queries the user for values for returned input elements.

GetInputs is called with the set of all operations defined in the WSDL file. The outermost loop ensures that GetInputs does not finish until sufficient input values have been provided to allow requests to be generated for all operations.

Algorithm 1 GetInputs(Set<Operation> *operations*)

```
 1: while $operations \neq \emptyset$ do
 2:    $done =$ FALSE
 3:    while $!done$ do
 4:       $done =$ TRUE
 5:       for all $o \in operations$ do
 6:          $executable =$ TRUE
 7:          for all $e \in o$.getRequiredInputElements() do
 8:             if $!e$.hasInputSource() then
 9:                $executable =$ FALSE
10:                break
11:             end if
12:          end for
13:          if $executable$ then
14:             for all $e \in o$.getOutputElements() do
15:                $e$.addInputSource($o$)
16:             end for
17:             $operations = operations - \{o\}$
18:             $done =$ FALSE
19:          end if
20:       end for
21:    end while
22:    for all $o \in operations$ do
23:       for all $e \in o$.getRequiredInputElements() do
24:          if $!e$.hasInputSource() then
25:             prompt user for input values to add as input source for $e$
26:          end if
27:       end for
28:    end for
29: end while
```

The while loop defined on lines 3-21 removes as many operations from the set *operations* as possible given the current provided input values. Since each operation that is removed from *operations* could potentially provide an input source for other operations in *operations*, it is necessary for this loop to continue until no more operations are removed from *operations*. The loop starting at line 5 iterates through each operation o in *operations*. This loop first determines if o is executable by checking to see if all of its required input elements have an input source (lines 7-12). An element has an input source if it has enumerated values, has already had input values provided by the user, or has had some operation added as an input source in line 15 of GetInputs. If o is found to be executable, o is added as an input source to all of its output elements (lines 14-16) and o is removed from *operations* (line 17).

When it is no longer possible for the loop at lines 3-21 to remove operations from *operations*, lines 22-28 of GetInputs then prompt the user for input values for input elements without input sources that are required by operations still in $operations$. The user is not obligated to provide values for all of these input elements, but must provide values for enough of them that at least one of the operations in $operations$ can be executed, at which time the outermost loop will repeat (if there are still operations in $operations$).

It should be noted that while GetInputs attempts to minimize the number of simple elements for which the user must provide inputs, the user is allowed to provide inputs for additional elements if the user desires. The set of simple input elements and their values are stored in the *Input Database*, where later steps in the process can access and update them.

3.2. Generating Requests

The Generator generates individual requests to the web service. Each generated request is passed to the Submitter, and the values of simple elements in the response (when the response is valid) are added to the input database to be used in later requests.

Algorithm 2 GenerateRequests(int $total$, Set<Operation> $allOperations$)

```
1: $count = 0$
2: repeat
3:     $operations = \emptyset$
4:     for all $o \in allOperation$ do
5:         if $o$.hasAllInputElements() then
6:             $operations = operation \cup \{o\}$
7:         end if
8:     end for
9:     $o$ = ChooseOperation($operations$)
10:    $request$ = ConstructRequest($o$.getRootElement())
11:    SubmitRequest($o$,$request$)
12:    $count = count + 1$
13: until $count \geq total$
```

Algorithm 2 presents algorithm GenerateRequests, the overall process for generating $total$ requests ($total$ is specified by the user as a parameter to WebAppSleuth) from the set of operations, $allOperations$, defined in the WSDL file. Each iteration of the outermost loop in GenerateRequests generates one request.

GenerateRequests begins by determining the set, $operations$, of operations that have inputs available for all required elements (lines 4-8). Next GenerateRequests chooses an operation, o, from the set $operations$. There are many possible heuristics for choosing an operation; the implementation currently chooses the operation randomly.

After selecting o, the input for o needs to be constructed. This occurs in the ConstructRequest procedure that is called in line 10 of GenerateRequests and presented in Algorithm 3. ConstructRequest is called with the top-level element of the input message to o. Each input element, $element$, can occur a variable number of times as defined by its minOccurs and maxOccurs attributes in the XML schema. So the first step of ConstructRequest is to choose a value for $occurs$, the number of times $element$

Algorithm 3 ConstructRequest(Element $element$)

```
1:  if $element$.hasInputValues() then
2:    $occurs$ = ChooseOccurs($element$)
3:    for $i = 1...occurs$ do
4:      if $element$.isSimpleElement() then
5:        $request$.append(ChooseValue($element$))
6:      else
7:        for all $e \in element$.getSubElements() do
8:          $request$.append(ConstructRequest($e$))
9:        end for
10:     end if
11:   end for
12: end if
13: return $request$
```

will actually occur in the request (line 2). There are several heuristics possible for choosing the value of $occurs$, but currently it is chosen randomly from the possible range of values between 0 and the smaller of the specified maxOccurs attribute and a user-provided upper bound. WebAppSleuth always uses 0 even when the minOccurs attribute is greater than 0 so that it can determine if the minOccurs attribute is actually correctly set. For the upper bound, generally small values are best as the likelihood of a request being valid tends to decrease as the size of the request increases. However, there is a trade-off involved in this decision in that larger requests may be able to find additional problems. After choosing a value for $occurs$, $element$ must be appended to the request $occurs$ times. If $element$ is a simple type, WebAppSleuth chooses a value (again randomly) from the set of available input values for $element$ (line 5). If $element$ is a complex type, ConstructRequest is called recursively on each of its sub-elements (lines 7-9). Finally this generated request is sent to the Submitter to be submitted to the web service.

Alternative Heuristics. The use of random selection within the request generation process can limit the ability of the approach to generate requests to thoroughly cover the input space for a web service which could lead to false positives or negatives in the inference generation steps of WebAppSleuth. In prior work for specialized search engines [15], two non-random heuristics, one based on covering arrays and another that used feedback to guide the generation of new requests, were used. The covering array approach is not directly applicable to the more complex structured inputs used by web services, although other systematic approaches to covering XML input spaces such as TAXI-WS [5] could be applied.

While the feedback approach was highly successful in the case of specialized search engines, it is non-trivial to extend it to the more general case. Various methods of using feedback information were tried, but these approaches have performed more poorly than the random approach presented here. Examination of the behavior of the feedback approaches show that they tend to limit the search space that is explored, leading to a deeper exploration of a narrow slice of the web service's behavior.

Additional simpler heuristics, for example a least-recently used value heuristic for selecting the values for inputs, have also been explored. The examined heuristics did not show any improvement over random, and in some cases led to poorer results than random, therefore this work focuses on the random approach to generation.

3.3. Submitting Requests and Classifying Responses

The Submitter is responsible for submitting requests to the web service and for classifying the responses from the web service. To submit requests to the web service submission modules that work with Sun's JAX-WS tools [27] and with the Apache Software Foundation's Axis tools [28] have been implemented. These tools automatically build Java libraries for accessing web services from WSDL declaration files. Using Java Reflection, WebAppSleuth is able to make use of these libraries to submit requests to the web service. Since the generated libraries return the responses as complex objects whose structure depends on which tool was used to generate the library, Java Reflection is used 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. These classifications can then be used by the various inference algorithms to be discussed in Section 3.5. Classification of individual request-response pairs raises two issues. First, it is possible that a request with bad inputs, for example leaving out a required input element, will not directly result in an invalid response, but could change the state of the server in some fashion that leads later requests to fail. If this happens, the methodology might miss some inconsistencies or incorrectly identify inconsistencies in the operation where the invalid response occurs.

Second, while it may be the case that a particular request results in an invalid response due to multiple problems, this is generally not expected to be an issue. WebAppSleuth generates large numbers of different requests which allows it to tease apart the effects of different problems within the requests.

Classification requires specific checks depending on the web service, as the mechanism for reporting errors varies depending on the web service implementation. For example, eBay's service uses the SOAP exception model to indicate problems in requests, while Amazon's service includes a list of $Error$ elements in the response message. The user is required to provide the criteria for classifying responses.

Finally, for request-response pairs that have been classified as valid, the input database is updated with the values for simple elements returned in the response.

3.4. Example Request Generation and Submission

Figure 3 shows the state of the Input Database, the executable operations, the requests and the responses as WebAppSleuth generates and submits two requests to the example bookstore service.

The table, $input_1$, shows the initial state of the input database after the Analyzer has extracted input values from the WSDL file and the user. The Generator determines the set of executable operations ($operations_1$). `StartSession` is the only executable operation since all other operations in the service require a `sessionId` and there are no values for `sessionId` in the Input Database. Since `StartSession` does not have an input message, the process of generating $request_1$ is complete, and $request_1$ can be submitted.

The bookstore service then returns $response_1$ to WebAppSleuth. This response includes one new input value for `sessionId`, "S01", which is added to the input database ($input_2$).

The process is repeated to generate the second request. Since WebAppSleuth now has a value for `sessionId` along with the other input values collected by the Analyzer from the user and the WSDL

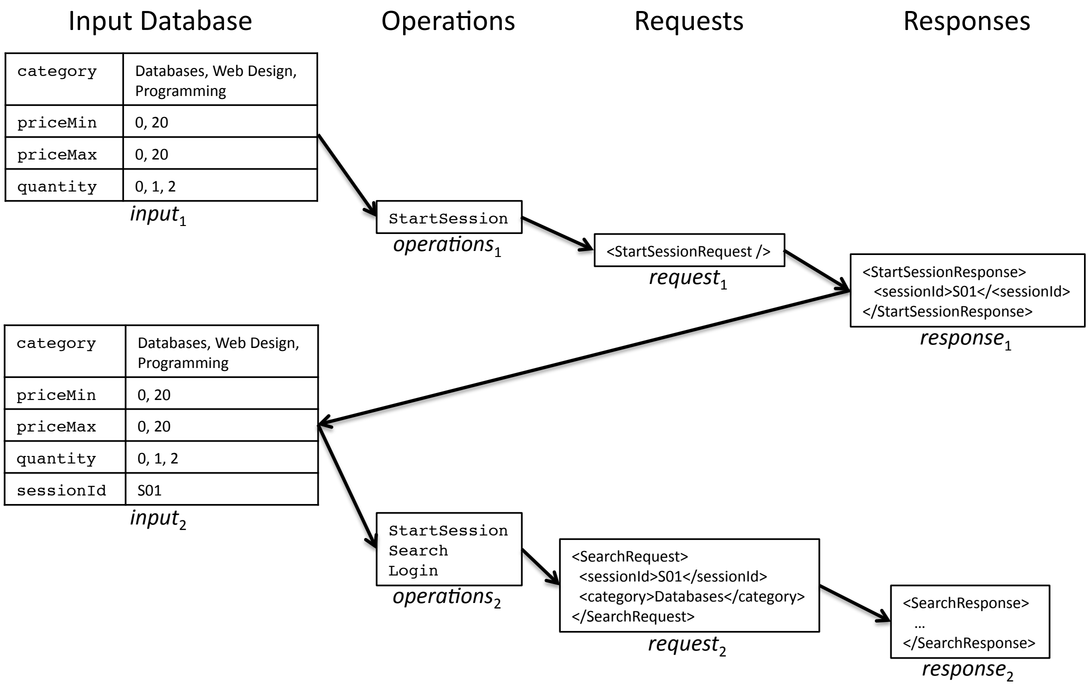
Input Database
Operations
Requests
Responses
category
Databases, Web Design, Programming
priceMin
0, 20
priceMax
0, 20
quantity
0, 1, 2
input1
StartSession
operations1
<StartSessionRequest />
request1
<StartSessionResponse> <sessionId>S01</sessionId> </StartSessionResponse>
response1
sessionId
S01
input2
StartSession Search Login
operations2
<SearchRequest> <sessionId>S01</sessionId> <category>Databases</category> </SearchRequest>
request2
<SearchResponse> … </SearchResponse>
response2

Figure 3. Example WebAppSleuth Generation and Submission

file, there are three executable operations ($operations_2$). The call to ChooseOperation randomly picks one of these operations – in this example, the `Search` operation. Then, ConstructRequest is called with the root input element of the `Search` operation, `SearchRequest`.

`SearchRequest` must occur exactly one time, so $occurs$ has the value 1 at line 2 of ConstructRequest. Since `SearchRequest` is not a simple element, the else branch of the if statement starting at line 4 is taken. ConstructRequest is called recursively for each of the sub-elements of `SearchRequest`: `sessionId`, `priceMin`, `priceMax`, `category`, and `name`. Element `sessionId` has minimum and maximum number of occurrences both equal to one, and one available value, "S01", so it will be added to the request once with that value. The four input elements, `priceMin`, `priceMax`, `category`, and `name`, have minOccurs equal to 0 and maxOccurs equal to 1. Since the `name` element has no input values, it cannot occur in the request (line 1 of ConstructRequest). For each of the other three elements $occurs$ will be either 0 or 1. Assume $occurs = 0$ is chosen for `priceMin` and `priceMax` and $occurs = 1$ is chosen for `category`. Finally a value must be chosen for the one occurrence of `category`. Assume that "databases" is that chosen value.

3.5. Generating Inferences

The Inferrer takes the set of submitted requests 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 of the web service. Currently, there are two different inference algorithms implemented, *required and optional elements* and *simple element types*. These types of inferences were chosen based on initial inspections of existing WSDL files that showed that certain types of errors were common.

3.5.1. Required and Optional Elements

The first inference algorithm attempts to correct errors in the minOccurs attribute of elements defined in the WSDL declaration, specifically finding elements with minOccurs $= 1$ (required) that should be 0 (optional) or minOccurs $= 0$ (optional) that should be 1 (required). These two cases were chosen because an initial examination of the WSDL file and HTML documentation for the Amazon Advertising 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 required and optional variables presented in prior work [15], with the primary difference being that only cases in which the inferred behavior differs from the behavior specified in the WSDL file are reported. For each element e nested in a complex type t used for an input message as defined in the WSDL specification, four boolean variables are defined as follows:

- the element e occurs 0 times in at least one request with a valid response ($absentValid$)
- the element e occurs 1 or more times in at least one request with a valid response ($presentValid$)
- the element e occurs 0 times in at least one request with an invalid response ($absentInvalid$)
- the element e occurs 1 or more times in at least one request with an invalid response ($presentInvalid$)

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 how many times element e occurred in that request. For example, in the second request (a `Search` request) in the example sequence generated in Section 3.4, the variable `sessionId` occurred once. If that request was submitted and the response was classified as valid, the $presentValid$ boolean variable for the `sessionId` element of the `SearchRequest` complex type would be set to true, and the other variables would be unchanged. After processing all submitted requests, if $absentInvalid$ is true (at least one request without the variable was attempted), $absentValid$ is false (none of the requests without the variable returned a valid response) and $presentValid$ is true (a request that included the variable and returned a valid response was submitted), then the variable is required. If $absentValid$ is true (there was a request without the variable that returned a valid response), then the variable is optional. In all other cases, there is not enough information to determine whether it should be required or optional. Any cases where the calculated required/optional status for a variable differs from the status defined in the WSDL are reported to the user as possible errors in the WSDL file.

It is important to note that a single complex type can be used as part of the input message for multiple different operations. Since the WSDL file contains only a single definition for such a complex type, it is not possible for it to define some of its nested elements as being required for some operations and optional for others, although supplemental documentation may do so. WebAppSleuth will generate only one required or optional inference for each of the nested elements within the complex type regardless of the number of operations within which the complex type is used. This may prevent WebAppSleuth from finding larger structural issues where a particular complex type is used with different assumed constraints by separate operations. A possible solution to this would be to generate separate required and optional inferences for the nested elements for each operation within which the complex type is used. This solution has not been explored.

This inference algorithm only identies the required or optional status for elements used in input messages. This is because an element is marked as required if the absence of that element appears to cause the server to produce an error message, which only makes sense when considering elements that are present or absent in input messages.

3.5.2. Simple Element Types

As mentioned in Section 2 a common problem in WSDL files is imprecise or incorrect declarations of types for simple elements. The XML schema standard provides a wide variety of types that can be used and choosing the best type for a particular simple element can be difficult. Therefore the type inference algorithm describes the different types that are accepted for a particular simple element to help client developers better understand what types of values are likely to work for that element.

The type inference algorithm focuses primarily on the variety of numeric types that are defined by the XML schema standard. These types were focused on because there are a wide variety of numeric types and selecting between the different options can be difficult.

The technique analyzes the types of every simple value used in a valid request or returned in a response. WebAppSleuth uses the declared type in the WSDL file as an upper bound on the return type. If the declared type of the element is a string type (`string`, `token`, or `normalizedString`), the value is parsed to see if it is a string representation of a more specific type. Specifically it checks to see if the value is a `boolean` (value is “true” or “false”), a number (can be parsed by constructors for java.math.BigDecimal or java.lang.Double), a `date`, `time` or `dateTime` (using parse methods of appropriate java.text.DateFormat instances), or an `anyURI` (can be parsed by a constructor for java.net.URI).

If the value is a number (either declared as a numeric type or inferred to be numeric), a further analysis is performed on the numeric types. The value is classified along three dimensions. The first and simplest dimension is the sign of the value, one of: `positive`, `negative`, `zero`, or `NaN` (if the value is a float or double NaN value).

The second dimension of classification is the size of the integer value. If the value can be expressed as an integer value without loss of precision (i.e., it has no digits to the right of the decimal point), the value is marked with the subset of the following types that are large enough to hold it: `byte` (-128 to 127), `unsignedByte` (0 to 255), `short` (-32768 to 32767), `unsignedShort` (0 to 65535), `int` (-2147483648 to 2147483647), `unsignedInt` (0 to 4294967295), `long` (-9223372036854775808 to 9223372036854775807), `unsignedLong` (0 to 18446744073709551615), or `integer` (arbitrary precision integer). For example, the value -35 would be marked as {`byte`, `short`, `int`, `long`, `integer`}, and the value 4000000000 would be marked as {`unsignedInt`, `long`, `unsignedLong`, `integer`}. For non-integer values, the value would be marked with the empty set.

Observed signs	Generalized sign
positive	positive
negative	negative
positive, zero	nonNegative
negative, zero	nonPositive
all other combinations	none

Table II. Reported generalized sign

The third dimension of classification is the precision of floating point values. The XML schema standard defines three different precisions of floating point values: float (IEEE 754-2008 binary32), double (IEEE 754-2008 binary64), and decimal (arbitrary precision decimal number). Similar to the integer size classification, we classify each value according to the set of precisions that are able to contain that value without loss of precision. For example, the value 1.5 would be marked as {float, double, decimal} and the value 1.1 would be marked as {decimal} (1.5 can be precisely represented by a float or double in binary representation, but 1.1 cannot). While it might initially seem that all values representable as a float or double could be represented as a decimal, there are three values, -INF, INF, and NaN, that have float and double representations but do not have decimal representations.

After classifying each of the values, a generalized type for each element name or element name/type pair defined in the WSDL file is generated. For any element that has at least one value with a non-numeric observed type, if it has values with two or more different observed types, it is generalized to the type string. Otherwise the observed type is reported.

For values with numeric types, the types are generalized along each of the three dimensions independently. Table II shows the rules used to generalize different combinations of observed signs for a particular element. This generalized sign can be used to select between integer, positiveInteger, negativeInteger, nonPositiveInteger, and nonNegative-Integer if the type is otherwise determined to be an integer type, or to apply a range restriction on a value for any numeric type.

To generalize the second dimension of numeric types, size of integer, the set intersection of the observed types is used. Then, the reported types will be the smallest signed type (byte, short, int, long, integer) and, if all values were non-negative, the smallest unsigned type (unsignedByte, unsignedShort, unsignedInt, unsignedLong) that remain in the intersection. These types represent the smallest integer types that can hold all of the observed values. If this set is empty then there is no appropriate integer type for the element.

To generalize the floating point dimension of numeric types, the set intersection of observed types is used. If the intersection includes float, it is reported as a usable type, otherwise, if the intersection includes double, double is reported as a usable type. Finally, regardless of whether float or double are usable types, if the intersection includes decimal, it is reported as an additional usable type.

Metric	Amazon	eBay
Version	2008-08-19	582 (JAX-WS)
Lines in WSDL	3,839	102,737
Operations in WSDL	23	134
Operations analyzed	22	112
Elements needing values	19	34
Elements with values in documentation	12	9

Table III. Details about artifacts

The reported numeric type will suggest a range of possible types that can be used to represent the observed values. Determining the correct actual type is left up to the user based on the information presented and their domain knowledge about the application.

Note that, unlike the required and optional inferences, the type inferences are only based on the set of values that the web service accepts, not on those that it fails to accept. Therefore, values that appear in output messages are included along with those that occur in input messages with valid responses to provide a broader base of values from which to infer the types.

4. Evaluation

The primary goal of the WebAppSleuth methodology is to help client developers better understand the requirements for using a web service for which they do not have the source code. Therefore, to evaluate WebAppSleuth, it has been applied to two commercial web services without any special access to the services. To measure the effectiveness of the information the returned inferences were compared to the public documentation of the service. As much as possible, further discussion into how the information provided by WebAppSleuth could help the web service's client developers is presented.

4.1. Study Infrastructure

To evaluate the WebAppSleuth methodology two case studies were performed on production web services, one provided by Amazon¶ and the other provided by eBay‖. The third and fourth rows of Table III 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 were considered, leaving 112 of the 134 total operations. On Amazon, one operation was excluded, `MultiOperation`, which provides a means for batching multiple other operations in order to improve performance. For each object 1,000,000 requests were submitted.

¶`http://aws.amazon.com/associates/`
‖`http://developer.ebay.com/products/trading/`

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, values were produced based on domain knowledge about the service. The last two rows of Table III indicate the total number of simple elements that have values provided and the number of those elements that had values defined in their corresponding HTML documentation.

Second, the Generator requires the user to specify a maximum number of occurrences for any element. For this study, up to three occurrences were allowed as this provided for the generation of requests that included no occurrences of the element, one occurrence of the element (for elements with minOccurs less than or equal to one), and more than one occurrence of the element (for elements with maxOccurs greater than one). As mentioned in Section 3.3, the choice of value for maximum number of occurrences represents a trade-off between the likelihood of producing valid requests and the ability of the methodology to find errors that are exposed only on large requests. With the value set at three, WebAppSleuth was able to produce valid requests only 7% of the time on Amazon and 48% of the time on eBay.

Finally a classification criterion for responses from each of the services is required. As stated in Section 3.3, Amazon returns error elements detailing errors in the request while eBay uses the SOAP exception model to return errors.

To get a better idea of the overall quality of the generated inferences, reports on all of the inferences are produced. These reports are then compared to both the WSDL information and to additional supporting documentation where available. For Amazon, HTML documentation that included both whether the input elements were required or optional and the expected type of the input elements was used. eBay included annotations that were embedded in the WSDL file and specified whether the elements are required, optional, or conditionally required. In many cases, the WSDL file and the additional documentation disagreed with each other.

4.2. Results

4.2.1. Generated Inputs

One important question is the ability of WebAppSleuth to generate a variety of inputs to cover the operations of the web services being analyzed. On Amazon, WebAppSlueth was able to generate requests for 19 of the 22 operations considered, and on eBay requests were generated for 94 of the 112 operations considered. This indicates that WebAppSleuth was unable to find suitable inputs for some of the required elements for the uncovered operations. This was most likely caused by the inability to generate valid requests for some of the operations. Specifically, on Amazon there were two operations that had only invalid requests generated and on eBay there were 31 operations that had only invalid requests.

On eBay there were 22 operations that had no requests generated until after 788,000 other requests were generated. All of these operations had requests generated within 5,000 requests of each other, indicating that they were likely dependent on the same or related elements. This also led to significantly fewer requests being generated for each of these operations (approximately 2,000 requests each rather than the approximately 13,000 requests for the other operations).

As mentioned earlier, only 7% of the generated requests for Amazon were valid while 48% of the generated requests for eBay were valid. This difference occurred because Amazon had the majority of the elements defined as optional in the WSDL even though they were in fact required according to the HTML documentation. This issue is discussed further in Section 4.2.3.

It is possible for WebAppSleuth to generate duplicate requests. On Amazon, WebAppSleuth submitted 913,399 unique requests, with some of the duplicate requests being submitted as many as 3,300 times. The operations `CustomerContentSearch` and `Help` account for the majority of the duplicate requests submitted. This is because both of these operations had relatively simple input message structures with a small number of values for each of the elements.

On eBay, WebAppSleuth submitted 962,883 unique requests with some requests submitted as many as 40 times. Similar to Amazon, there were a small number of requests that represented most of the duplications. The primary reason why the requests were duplicated fewer times than on Amazon is that the number of operations was larger so fewer requests were sent for each operation.

4.2.2. *Running Time*

It took approximately two days and six hours to generate and submit one million requests to the Amazon Advertising web service, and approximately six days and 23 hours to generate and submit one million requests to the eBay Trading web service. After collecting the requests, the analysis to infer the properties took three minutes and 19 seconds for Amazon, and 16 minutes and eight seconds for eBay.

The primary bottleneck in both cases was the time it took the web service to respond to a request. Specifically, Amazon claims to throttle requests to one per second (although the data gathered for this study suggests that this is not actually the case, at least at the time requests were submitted). In the case of eBay, the slower performance is probably a result of the fact that development servers were used, which, presumably, have lower priority in terms of performance than production servers such as those used for Amazon. It should be noted that on earlier and smaller attempts at collecting data for these two services Amazon was slower than eBay.

4.2.3. *Required and Optional Elements*

Table IV presents a breakdown on how the variables were classified by WebAppSleuth, the WSDL file, and the documentation for the services.

For both of the applications, a substantial majority of the elements were defined as optional in both the WSDL and supporting documentation. On Amazon, WebAppSleuth classified nearly half of these elements as optional. WebAppSleuth classified three of the elements as required. Two of these elements were the simple elements, `About` and `HelpType`, for the `Help` operation. Manually submitting `Help` requests without one or both of these two elements generated error messages indicating that these two elements are required, demonstrating that both the documentation and the WSDL are incorrect. The third element is the `Items` complex element within the `CartCreateRequest` type use by the `CartCreate` operation. This element serves as a container for holding multiple `Item` elements, and the documentation indicates that a `CartCreate` operation must include at least one `Item` element, therefore the `Items` element should be marked as required.

(a) Amazon

WSDL	Documentation	Total	Optional	Required	Unknown
Optional	Optional	180	88	3	89
Optional	Required	29	1	23	5
Required	Optional	0	0	0	0
Required	Required	4	0	2	2

(b) eBay

WSDL	Documentation	Total	Optional	Required	Unknown
Optional	Optional	1198	381	9	808
Optional	Required	134	7	25	102
Required	Optional	12	5	2	5
Required	Required	20	2	1	17

Table IV. Required and optional elements, rows indicate specified optional/required status and columns indicate WebAppSleuth-identified optional/required status

For eBay, WebAppSleuth was able to verify the classification for 32% of the elements defined as optional according to both the WSDL minOccurs element and the documentation. WebAppSleuth classified nine elements that were specified as optional as being mandatory. Of those nine, five are defined as being conditionally required in the documentation. The incorrect classification of these five elements demonstrates a limitation of the implementation; the random approach to request generation has difficulty generating requests that satisfy complex constraints. The remaining four elements appear to be incorrectly identified as optional in the WSDL and should be changed.

Of these four, the `SKU` element is used to hold individual entries within the `SKUArrayType` type. An element of `SKUArrayType` type constructed without any nested `SKU` elements is essentially a zero-length array. In places where an operation does not actually need an array of SKUs, the element with `SKUArrayType` type is itself marked optional. This suggests that the desired behavior is to include only the array of SKUs when there is at least one element within it, suggesting that at least one `SKU` element should be required within an element of `SKUArrayType`.

Examination of the documentation for the element `EventType` within `Notification-EnableType` suggests that it should be required; `EventType` indicates which category of events are being enabled or disabled. Including an `NotificationEnableType` element without including the nested `EventType` element does not make sense for the application.

The final two of these four are `ShippingIncludedInTax` and `JurisdictionID` nested within the `TaxJurisdictionType`. According to the documentation the `TaxJurisdictionType` is used in requests to `SetTaxTable` and returned in responses to `GetTaxTable`. When `TaxJurisdictionType` is used in requests to `SetTaxTable`, these two elements are required. However, requests generated by `GetTaxTable` are allowed to not include these two elements. Therefore the elements have to be defined as optional. However, since WebAppSleuth uses the requests only to classify elements as optional or required, it correctly identified these elements as being required.

In this case, because of the difference in how the type is used in the two different operations, the separation of the `TaxJurisdictionType` into two separate types might be indicated.

> *Impact.* Of the seven elements incorrectly specified as optional in both the WSDL and documentation, the two for the `Help` operation on Amazon are unlikely to cause significant problems as they were not intended for use in user applications. However, the remaining five make it more difficult for developers to understand how to use the services and could lead to increased application failures if the developer fails to determine that these elements are required.

The second largest class of elements for both applications are elements defined as required in the documentation and optional in the WSDL. In the case of Amazon, these elements are defined as optional in order to support REST/Query and SOAP clients with the same WSDL[**] and it is possible that eBay specified these elements as optional in the WSDL for similar reasons. Due to this, it is expected that for these elements, the documentation should be considered more reliable. In spite of this expectation, WebAppSleuth classified one element on Amazon and seven on eBay as optional, indicating that the web services returned valid responses to requests missing these elements. WebAppSleuth agreed with the documentation-specified classification for 23 of 29 elements on Amazon and 25 of 134 elements on eBay.

> *Impact.* There are two possible errors for the eight elements that WebAppSleuth classified as optional; either the documentation is incorrect and the elements should not be required (possibly having some default value) or the web service implementations are missing input validation for these elements. If the documentation is incorrect, it is less likely to lead to errors as most applications using the service will include values for these elements and any default values that exist for these elements have probably been chosen by the service developers to be suitable for most requests. However, when problems do occur (e.g., in cases where the application mistakenly fails to provide a value for an element and the default value is inappropriate), it could be difficult to debug. Alternatively, if the problem is missing input validation, it is likely that the service developers have not specified the correct behavior for requests missing these elements, and therefore, the service could exhibit erratic and unexpected behaviors when processing these types of requests. These could lead to subtle errors that may not even be visible until later related service requests are processed, making it more difficult to debug applications that use the service.

The smallest class of elements for both applications are elements defined as required in the WSDL and optional in the documentation; for Amazon, there were no such elements and for eBay only 12 such elements. WebAppSleuth identified five of these elements as optional, indicating that for those five elements, the WSDL should be modified to make these elements optional.

WebAppSleuth identified two elements in this class as required. One of these elements is defined as conditionally required in the documentation, therefore it is likely that WebAppSleuth failed to construct

[**]Post on Amazon Web Services discussion forums: `http://developer.amazonwebservices.com/connect/message.jspa?messageID=101385`

a request that satisfied the conditions for leaving off this element. The other, `MaxDistance` for the `ProximitySearch` operation, was present in nearly 13,000 requests of which only 71 were successful, and absent in only 207 requests. This suggests that the other constraints for generating a valid `ProximitySearch` request are difficult to satisfy and that an insufficient number of requests were generated without a `ProximitySearch` element.

> *Impact.* The presence of elements marked as optional in the documentation and required in the WSDL on eBay was highly surprising. Unlike the opposite case, there is no valid reason for such a combination. Also, on eBay the documentation was present in machine-readable annotations embedded in the WSDL itself, so a simple consistency checker would have readily identified these errors. The elements that are actually optional are unlikely to cause problems for client applications as most systems that use the WSDL files to construct requests do not enforce the constraints on minimum number of occurrences. However, if using a system that enforces these constraints (or worse transitioning from one that does not to one that does), the inconsistent information could lead to developer confusion. If elements in this class that were actually required had been identified, they could lead to problems similar to those mentioned for elements that are marked as optional in both the WSDL and documentation.

For the final class of elements, defined as required in both the documentation and WSDL, WebAppSleuth classified two of four elements for Amazon and one of 20 elements for eBay as required. WebAppSleuth found two of these elements on eBay optional. Both of these cases are related to the `AmountType` type, a complex type that extends `double` and adds an attribute, `currencyID`. The current implementation does not correctly handle such types which led to these false reports.

4.2.4. Types

Tables V, VI and VII show the declared and WebAppSleuth inferred types for Amazon and eBay, respectively. In these tables, the columns correspond to the WSDL declared types for the elements, the rows correspond to the WebAppSleuth inferred types for the elements, and the numbers show the number of elements for each declared/inferred type combination. For example, the last column of the third from bottom row of Table V indicates that for Amazon there were two elements with the inferred type of `boolean` and the declared type of `string`.

This data shows several interesting trends in these web services. First, the kinds of types used by the services are drastically different. eBay tends to use implementation-oriented types, e.g.,`int` or `float`, rather than `integer` or `decimal`. Using implementation-oriented types potentially allows the clients and the server to optimize the choice of data types in their implementations, which can improve their performance. On the other hand, Amazon tends to use semantics-oriented types, using `float` for only one element and never using `double` or any of the implementation-oriented integer types. The use of these types require clients and the server to use arbitrary precision integer and decimal types to ensure that the full potential range of values can be represented, and potentially increasing the runtime overhead of both the client and service. However, by using types such as `positiveInteger` and `nonNegativeInteger`, Amazon provides some additional information about the expected range of values that is not included in eBay.

Another interesting distinction is eBay's use of the `token` type. The `token` type specifies that the only allowable white-space character is the space character, that leading and trailing white-space is disallowed, and that adjacent white-space characters are not allowed. In practice, XML parsers allow any arbitrary string to appear in elements of type `token`, and normalize the values of the elements according to the rules of `token` when parsing the file. This has the advantage of automatically removing spurious white-space from values, potentially making applications that use the `token` type (and related type `normalizedString`) more robust to minor variations in generated XML.

Overall, it appears that eBay does a slightly better job than Amazon of correctly classifying elements as a numeric or `boolean` type rather than `string` or `token` type: WebAppSleuth finds that only 20 out of 241 string-type elements could potentially be redefined to numeric or `boolean` types on eBay and that 22 out of 193 string-type elements could be redefined on Amazon. For eBay, the identified elements can be divided into two general classes. One of these classes, including 17 of the 20 elements, are identifiers. These elements could probably have their types changed to appropriate numeric types. The remaining three elements are postal codes. In the United States, postal codes are numeric but can include leading zeros, and in other countries postal codes can include non-numeric characters, therefore these elements should be defined as a string type.

On Amazon, two of the identified elements are identifiers that could probably be changed to an appropriate numeric type. 17 of the identified elements represent quantities or ranks, and should be numeric types. According to the HTML documentation, the two elements identified by WebAppSleuth as `boolean` allow only the values "true" and "false" and should, therefore, be redefined as `boolean` in the WSDL. Finally, for one of the elements, `MetalStamp`, it is unclear based on the documentation if it could safely be redefined as a numeric type.

Inferred Type / Declared Type	**Total**	integer	positiveInteger	nonNegativeInteger	float	decimal	boolean	string
(byte,unsignedByte,float,decimal)	1							1
(short,float,decimal)	1	1						
(positive,byte,unsignedByte,float,decimal)	26		11	9		1		5
(positive,short,unsignedShort,float,decimal)	5			5				
(positive,int,unsignedInt,float,decimal)	2			1				1
(positive,int,unsignedInt,double,decimal)	2							2
(positive,unsignedLong,decimal)	1		1					
(nonNegative,byte,unsignedByte,float,decimal)	7			4				3
(nonNegative,short,unsignedShort,float,decimal)	11	2		3				6
(nonNegative,int,unsignedInt,float,decimal)	5	1		3				1
(nonNegative,unsignedLong,decimal)	16			16				
all integer types	77	4	12	41		1		19
(positive,float,decimal)	2				1	1		
(positive,double,decimal)	1							1
(nonNegative,double,decimal)	2					2		
all float types	5				1	3		1
anyURI	76							76
boolean	5						3	2
string	95							95
Total	258	4	12	41	1	4	3	193

Table V. Types for Amazon, rows indicate WebAppSleuth-identified types and columns indicate WSDL-specified types

Inferred Type / Declared Type	**Total**	int	long	float	double	decimal	boolean	dateTime	duration	string	anyURI	token
(byte,float,decimal)	3	3										
(byte,unsignedByte,float,decimal)	43	26	10	3	1					3		
(short,float,decimal)	1	1										
(int,float,decimal)	6	5								1		
(int,double,decimal)	16	16										
(long,decimal)	4		4									
(positive,byte,unsignedByte, float,decimal)	22	19								3		
(positive,short,unsignedShort, float,decimal)	12	6								6		
(positive,unsignedShort, float,decimal)	1	1										
(positive,int,unsignedInt, float,decimal)	4									3		1
(positive,int,unsignedInt, double,decimal)	3	3										
(positive,long,unsignedLong, double,decimal)	1									1		
(nonNegative,byte,unsignedByte, float,decimal)	22	20	1							1		
(nonNegative,short,unsignedShort, float,decimal)	8	8										
(nonNegative,int,unsignedInt, float,decimal)	1	1										
(nonNegative,int,unsignedInt, double,decimal)	2	2										
(nonNegative,long,unsignedLong, double,decimal)	1									1		
(nonPositive,byte, float,decimal)	1	1										
all integer types	151	112	15	3	1					19		1

Table VI. Types for eBay, rows indicate WebAppSleuth-identified types and columns indicate WSDL-specified types

Inferred Type / Declared Type	**Total**	int	long	float	double	decimal	boolean	dateTime	duration	string	anyURI	token
double	1				1							
decimal	1					1						
(nonNegative,float,decimal)	3			3								
(nonNegative,double,decimal)	1				1							
all float types	6			3	2	1						
boolean	61						61					
dateTime	32							32				
duration	1								1			
string	42									42		
anyURI	187									38	8	141
Total	480	112	15	6	3	1	61	32	1	99	8	142

Table VII. Types for eBay (continued from table VI), rows indicate WebAppSleuth-identified types and columns indicate WSDL-specified types

> *Impact.* By specifying elements as `string` type, the service developers have provided no feedback to client developers about the kinds of values expected for these elements. By identifying more specific applicable types, WebAppSleuth is able to provide additional guidance to the client developers about the expected kinds of values for these types.

The use of the constructor for java.net.URI to identify elements that could be of type `anyURI` generated large numbers of false positives for both applications, identifying 76 elements on Amazon and 179 elements on eBay as being candidates to be changed to type `anyURI`. Examination of the WSDL file finds that most of these elements do not logically represent URIs, suggesting that improved criteria for identifying URIs are required.

Both services do a good job of identifying when a floating-point type is needed rather than an integer type, with WebAppSleuth identifying only one case for Amazon and four cases for eBay where the type could potentially be changed. In the case of eBay, the only value observed for the identified elements was 0, suggesting that these are likely false positives reported due to an insufficient number of samples. For Amazon, the identified element is a `Rating` element nested in a customer review data structure. Examination of Amazon's review system indicates that a reviewer can provide only integer ratings of products with their reviews, therefore an integer type may be more appropriate.

In general, it appears that identification of an element as `nonNegative` or `positive` is correct. On Amazon, WebAppSleuth identified eight elements that were not already defined as `positiveInteger` or `nonNegativeInteger` for which it observed only positive or non-negative values. For Amazon, all five of the elements found to be `nonNegative` should not logically contain negative values. Two of the elements identified as `positive` could potentially include the value 0, and therefore should be `nonNegative`. The third element identified as `positive` is a request processing time that logically should always have a positive value, and therefore could have an appropriate range restriction applied to it.

For eBay, 62 elements were identified by WebAppSleuth as `nonNegative`, `positive` or `nonPositive` and had numeric declared types. Of these elements, the 61 that were identified as `nonNegative` or `positive` appear to primarily represent values such as counts or identifiers that should always be greater than or equal to zero. The element identified as `nonPositive` can potentially have positive values on the production eBay web service, but only negative values and zero were observed on the development eBay service where data was collected.

> *Impact.* Selecting a specific integer or floating-point type for an element is a design decision that can be influenced by a number of factors, including the expected range of the values, the necessary precision required for values, the server platform and the expected client platforms. Often the developer of a web service will make an overly conservative choice, declaring a type that is more general than the actual expected range of values. By providing information about the range of values that have actually resulted in or been returned in valid responses, WebAppSleuth can provide additional information about the actual expected range of values for a particular element.

4.3. Discussion

This study shows that the WebAppSleuth methodology can be successfully applied to non-trivial production web services. For these services a number of anomalies were found, both in the specified minOccurs values and in the specified types for the elements. For the Amazon service a developer was contacted who verified that for at least one of the identified cases where an element was defined as `string`, it should have been defined as a `boolean` and who explained the rationale for the number of cases where the minOccurs was 0 on required elements.[††] (A similar attempt to contact the developers at eBay did not elicit a response).

The anomalies in the required versus optional state of elements primarily consisted of elements identified as optional in the documentation that were in fact required. Most identified cases of this problem occurred when an element was optional with respect to the entire request, but it was required if its enclosing element was present. This situation can be difficult to understand and to concisely express in documentation.

The most useful element type anomalies found were the cases where an element was defined as a `string` when it should have been a numeric or `boolean` type. Even in cases where it would not necessarily be a good idea to change the declared type (e.g., for most ids or for zip codes) the additional knowledge about the expected types of values for a string (e.g., that it should consist entirely of digits) can provide useful information to the client developer that may not have been explicitly stated in the documentation. Within the numeric types, the most useful information WebAppSleuth was able to identify was whether an element value could only be positive, non-negative, or unconstrained. While these types of constraints are often included in the documentation of the service, they could also be readily added to the WSDL description for the service.

This study also shows several limitations of the methodology, in particular in the request generation methodology. Random request generation had difficulty producing inputs to identify conditionally required elements and valid non-zero values for some numeric elements. A more systematic approach to request generation, such as that used in WS-TAXI [5], could be used instead of or in conjunction with the current random approach to increase the likelihood of covering these corner cases.

Additionally, the fact that even with a million requests, there were operations within these services for which no requests were generated indicates a weakness in relying on the previously submitted requests to get input values for these operations. This could be mitigated by prompting the user for input values for additional simple elements if, after submitting a number of requests, WebAppSleuth is still missing inputs for some simple elements.

There are also some threats to the validity of this evaluation. First, the choice of objects may not be characteristic of other web services. To mitigate this, two production web services with sizable user bases were chosen. However, both of the services are targeted toward use by applications intended for general consumers for shopping (and, in the case of eBay, selling) merchandise. Services intended for different types of applications (e.g., business to business applications) and services in other domains

[††]`http://developer.amazonwebservices.com/connect/message.jspa?messageID=101385`. The issues identified in the forum post were based on an earlier version of WebAppSleuth and the Amazon web service and were specifically chosen by the first author based on his understanding of the web service and WebAppSleuth, and his belief that they were likely to be actual problems.

(e.g., stock price tracking) may have different types or numbers of issues. Second, during the study there were several points where choices about some property of the methodology had to be made, most importantly in the choice of input values for simple elements. Individuals with more or less knowledge about the services being considered or with less knowledge about the operation of WebAppSleuth might have made different choices.

5. Related Work

The first step of the WebAppSleuth methodology is the static generation of the relationship between input and output values of various WSDL operations; this is similar to the Jungloids described by Mandelin *et al.* [22] or the relationships found as the first step of the WSDL-based test generation strategy proposed by Bai *et al.* [4].

The request generation strategy is similar to random or exhaustive test generation strategies proposed for object-oriented applications [3, 8, 24, 29, 31]. The primary differences between these methods and this work is the domain (object-oriented applications versus web services) and the intended use of the inputs (testing versus generating inferences). One methodology [21] 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 less rigorously defined, making the problem of relating the types of inputs and outputs more difficult.

There has also been some work on systematically generating inputs for applications, such as web services, that use XML inputs. XPT uses a form of bounded exhaustive generation to generate a wide variety of XML documents from an XML schema and then uses heuristics based on category-partition testing [23] to reduce the number of test cases [6]. WS-TAXI uses the XPT approach to generate test inputs for web services with WSDL descriptions [5]. It would be possible to adapt WS-TAXI for use in the WebAppSleuth methodology at the level of determining the structure of the XML requests, but this would need to be modified to account for changing available input values at each step.

There has been a great deal of work on techniques for dynamically inferring properties about programs [2, 9, 12, 16, 20, 21, 25, 26, 30, 32]. These techniques 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 $v > 0$), temporal relationships between events (e.g., an open event must be followed by a close event) or even a state transition diagram describing the changes to interior states in the component. These methodologies differ from the WebAppSleuth approach in two major aspects. First, most assume that an engineer can instrument the code they are interested in examining (one exception to this is Perracotta which has been applied to detecting temporal properties in the Windows Kernel APIs by instrumenting the application code that uses those APIs rather than the API code itself [32]). Since WebAppSleuth works with web services and applications that are accessed via a network, it is assumed that the system can be instrumented, instead only the inputs and outputs are available. Second, all of these methods (except Henkel *et al.* [21]) require that there be an existing set of executable inputs. The WebAppSleuth methodology generates executable inputs automatically from simple input values provided by the user or mined from the WSDL file.

Ernst *et al.* [13] present a dynamic approach for the detection of substitutable and composable web services. Their approach is similar to WebAppSleuth in that it submits requests to the targeted web services and analyzes the responses to the requests to perform its analysis. However, like most of the work in dynamically inferring properties about programs, it assumes a preexisting source of executable inputs. Additionally, the inferred properties (substitutability and composability) are different from the properties inferred by WebAppSleuth (anomalies in the WSDL files).

Halfond and Orso [17, 18, 19] have developed a static approach to automatically discover the interfaces for web applications Their approach statically analyzes the Java servlet code for a web application to discover properties such as required and optional parameters (similar to our number of occurrences property), related groups of parameters (similar to the implication properties defined in Fisher *et al.* [15]), and the types of parameters (similar to WebAppSleuth's type properties). Unlike WebAppSleuth, their approach is entirely static, and, therefore, requires the source code for the application. It has only been implemented for Java-based web applications.

Previous work on WebAppSleuth [15] focused on specialized search engines, a limited subset of web applications that allow searches over structured data using structured inputs (e.g., a flight search that takes departure and arrival dates and locations and returns matching flights). The prior work presented a variety of request generation techniques, including random, covering array based and feedback based techniques. It also presented various property inference algorithms, including required and optional variables, ranges of values for variables, and descriptions of the relationships between different variables and values for variables. WebAppSleuth was applied to six different live, production search applications, and anomalous behaviors were found in five of those applications. In this work, WebAppSleuth is extended to cover a wider variety of web-based applications, specifically arbitrary web services described by WSDL files. In this work, new request generation and inference algorithms are developed and other existing components are adapted to support web services.

6. Conclusion

This works presents WebAppSleuth, a methodology for automatically identifying inconsistencies in web services with WSDL interfaces. WebAppSleuth collects static information from the WSDL file to drive a request generation process, and automatically submits large numbers of the generated requests to the web service to collect and analyze information about its behavior. WebAppSleuth is applied to two commercial web services.

The evaluation of WebAppSleuth shows that it can successfully generate large numbers of requests for production web services. Additionally, it was able to identify several cases where the behavior of the Amazon and eBay web services differed from what might be expected based the WSDL description of the service. The types of inferences produced by WebAppSleuth could be used by client developers to help them identify potential issues within the service that could affect the development of their client. Additionally, the information provided by WebAppSleuth could also be reported back to the service developers who could use it to improve the quality of the web service or the WSDL description of the service (as was attempted with differing degrees of success with both Amazon and eBay).

Future work will proceed along three major branches of inquiry. First, there are several plans for improving the methodology itself, including the development of better input generation techniques and additional inferred improvements to the WSDL files or underlying web services. Second, additional studies of the methodology are needed, both to evaluate the effectiveness of the identified inconsistencies and to explore its generalizability. Third, WebAppSleuth is an instance of *probing analysis*, a new class of black box dynamic program analysis technique that probes a component by executing it using a variety of automatically generated inputs and analyzing the resulting outputs. Probing analysis requires only a partial description of the interface of the component being analyzed (used to generate inputs) and the ability to execute the component with those inputs. Significantly, probing analysis does not need access to the source or executable code being analyzed, something which is generally not available to users of web services. Future work will further explore and defined this new class of program analysis techniques.

Acknowledgments

The authors would like to thank Barbara G. Ryder who provided valuable editorial advice.

REFERENCES

1. Amazon.com, Inc. Why use Amazon Web Services? `http://www.amazon.com/Why-Use-AWS-home-page/b/?node=15763371`.
2. G. Ammons, R. Bodìk, and J. Larus. Mining specifications. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pages 4–16, 2002.
3. S. Artzi, M. Ernst, A. Kieżun, C. Pacheco, and J. Perkins. 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.
4. X. Bai, W. Dong, W.-T. Tsai, and Y. Chen. WSDL-based automatic test case generation for web services testing. In *Proceedings of the IEEE International Workshop on Service-Oriented System Engineering*, 2005.
5. C. Bartolini, A. Bertolino, E. Marchetti, and A. Polini. WS-TAXI: A WSDL-based testing tool for web services. In *Proceedings of the International Conference on Software Testing, Verification and Validation*, pages 326–335, April 2009.
6. A. Bertolino, J. Gao, E. Marchetti, and A. Polini. 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, pages 114–129, September 2006.
7. E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. Web services description language (WSDL) 1.1. `http://www.w3.org/TR/wsdl`, March 2001.
8. C. Csallner and Y. Smaragdakis. JCrasher: An automatic robustness tester for Java. *Software: Practice and Experience*, 34(11):1025–1050, September 2004.
9. V. Dallmeier, C. Lindig, A. Wasylkowski, and A. Zeller. Mining object behavior with ADABU. In *Proceedings of the International Workshop on Dynamic Analysis*, May 2006.
10. eBay Developers Program. About the eBay Developers Program. `http://developer.ebay.com/businessbenefits/aboutus/`.
11. S. Elbaum, K.-R. Chilakamarri, M. Fisher II, and G. Rothermel. Web application characterization through directed requests. In *Proceedings of the 4^{th} International Workshop on Dynamic Analysis*, pages 46–56, May 2006.
12. M. Ernst, J. Cockrell, W. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, 27(2):99–123, February 2001.
13. M. Ernst, R. Lencevicius, and J. Perkins. Detection of web service substitutability and composability. In *Proceedings of the International Workshop on Web Services – Modeling and Testing*, pages 123–135, June 2006.
14. D. Fallside and P. Walmsley. XML schema part 0: Primer second edition. `http://www.w3.org/TR/xmlschema-0`, October 2004.

15. M. Fisher II, S. Elbaum, and G. Rothermel. Dynamic characterization of web application interfaces. In *Fundamental Approaches to Software Engineering*, volume 4422/2007 of *Lecture Notes in Computer Science*, pages 260–275, March 2007.
16. P. Guo, J. Perkins, S. McCamant, and M. Ernst. Dynamic inference of abstract types. In *Proceedings of the ACM International Symposium on Software Testing and Analysis*, pages 255–265, July 2006.
17. W. G. Halfond, S. Anand, and A. Orso. Precise interface identification to improve testing and analysis of web applications. In *Proceedings of the* 18^{th} *ACM International Symposium on Software Testing and Analysis*, pages 285–296, New York, NY, USA, 2009. ACM.
18. W. G. J. Halfond and A. Orso. Improving test case generation for web applications using automated interface discovery. In *Proceedings of the Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, September 2007.
19. W. G. J. Halfond and A. Orso. Automated identification of parameter mismatches in web applications. In *Proceedings of the ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 181–191, November 2008.
20. S. Hangal and M. Lam. Tracking down software bugs using automatic anomaly detection. In *Proceedings of the International Conference on Software Engineering*, pages 291–301, May 2002.
21. J. Henkel, C. Reichenbach, and A. Diwan. Discovering documentation for java container classes. *IEEE Transactions on Software Engineering*, 33(8):526–543, August 2007.
22. D. Mandelin, L. Xu, R. Bodik, and D. Kimmelman. Jungloid mining: Helping to navigate the API jungle. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2005.
23. T. Ostrand and M. Balcer. The category-partition method for specifying and generating functional tests. *Communications of the ACM*, 31(6):676–686, 1988.
24. C. Pacheco and M. Ernst. Eclat: Automatic generation and classification of test inputs. In *Proceedings of the European Conference on Object-Oriented Programming*, pages 504–527, 2005.
25. J. Perkins and M. Ernst. Efficient incremental algorithms for dynamic detection of likely invariants. In *Proceedings of the ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 22–32, November 2004.
26. S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems*, 15(4):391–411, November 1997.
27. Sun Microsystems. JAX-WS reference implementation. `https://jax-ws.dev.java.net/`.
28. The Apache Software Foundation. Apache Axis2/Java. `http://ws.apache.org/axis2/`.
29. W. Visser, C. Pasareanu, and S. Khurshid. Test input generation with Java PathFinder. In *Proceedings of the ACM International Symposium on Software Testing and Analysis*, July 2004.
30. J. Whaley, M. Martin, and M. Lam. Automatic extraction of object-oriented component interfaces. In *Proceedings of the ACM International Symposium on Software Testing and Analysis*, pages 218–228, 2002.
31. T. Xie and D. Notkin. Tool-assisted unit-test generation and selection based on operational abstractions. *Automated Software Engineering*, 13(3):345–371, July 2006.
32. J. Yang, D. Evans, D. Bhardwaj, T. Bhat, and M. Das. Perracotta: Mining temporal API rules from imperfect traces. In *Proceedings of the International Conference on Software Engineering*, pages 282–291, May 2006.