Fault Injection-based Test Case Generation for SOA-oriented Software

Jia Zhang, Member, IEEE; Robin G. Qiu, Sr. Member, IEEE

Abstract— The concept of Service Oriented Architecture (SOA) implies a rapid construction of a software system with components as published Web services. How to effectively and efficiently test and assess available Web services with similar functionalities published by different service providers remains a challenge. In this paper, we present a step-by-step fault injection-based automatic test case generation approach. Preliminary test results are also reported.

Index Terms— Boundary value, fault injection, test case generation, Web services.

I. Introduction

The concept of Service Oriented Architecture (SOA) has been widely acknowledged to be the strategic model for the next generation of Internet computing [1, 2]. SOA enables rapid development of new business software by integrating components of published Web services, which are self-contained software components universally accessible through standard Internet protocols. The Web services technology enables cross-language and cross-platform interoperability for distributed computing and resource sharing. Thus, it facilitates Business-to-Business (B2B) e-Commerce within and across organizational boundaries, by means of business organizations enabling universal Internet access to their software services through standard programmatic interfaces [3].

The backbone of the Web services paradigm encompasses three fundamental techniques: communication protocols, service descriptions, and service registration and discovery [4, 5]. Each category is represented by an *ad hoc* industrial standard. The Simple Object Access Protocol (SOAP) [6] acts as a lightweight protocol for exchanging structured and typed information between Web services; the Web Service Description Language (WSDL) is an eXtensible Markup Language (XML)-based description language to describe the programmatic interfaces of Web services [7]; and the Universal Description, Discovery, and Integration (UDDI) standard [8] provides a mechanism to publish, register, and locate Web services.

This SOAP+WSDL+UDDI technology stack enables the publication, discovery, and transportation of a specific Web service. However, as the paradigm of SOA and Web services changes the face of the Internet from a repository of data into a repository of services [9], a UDDI query may return a long list of available Web services with similar functionalities declared. How to effectively and efficiently test, assess, and select a qualified Web service that matches some predefined requirements becomes critical [9]. Moreover, an SOA-oriented software system may require multiple Web services as components. How to select a Web service that can coexist with other components further complicates the challenge.

The last fifty years of software development has witnessed the establishment of a research branch *software testing,* which contains a wealth of theories, technologies, methodologies, and tools to guide the verification process of a software product against a list of attributes, such as reliability, scalability, efficiency, security, reusability, adaptability, interoperability, maintainability, availability, portability, etc. However, Web services pose new challenges to software testing due to their unique features. Web services are hosted by their corresponding service providers and can only be accessed through published Web interfaces described in WSDL documents. In addition, their distinctive features of *dynamic discovery and invocation* require efficient testing and assessment of Web services components at run time.

In this research, we aim to explore effective and efficient techniques of automatic Web services test case generation to verify Web services-oriented systems. By "Web services-oriented system," we mean a software system that consists of one or more components that will be fulfilled by Web services. As the first step, we focus on reliability testing, which verifies the "probability of failure-free operation of a computer program for a specified time in a specified environment" [10]. Our essential idea is to automatically elicit boundary value-based test cases from WSDL documents. The concept of fault injection is exploited to guide the test case generation.

The remainder of this paper is organized as follows. In Section 2, we discuss our boundary values-based Web services reliability testing approach. In Section 3, we discuss in detail how to generate test cases. In Section 4, we present

Jia Zhang is with Northern Illinois University, DeKalb, IL 60173 USA. She is also a Guest Scientist at National Institute of Standards and Technology (e-mail: jiazhang@cs.niu.edu).

Robin G. Qiu is with the Pennsylvania State University, Malvern, PA 19355. The work was partially supported by NSF grant (DMI-0620340).

our preliminary experiments. In Section 5, we compare our approach with related works. In Section 6, we make conclusions and discuss future work.

II. BOUNDARY VALUES-BASED WEB SERVICES TESTING

Our research applies boundary values and faulty data to test reliability of Web services candidates. To increase efficiency, our major strategy is to generate test cases to eliminate Web service candidates, instead of proving reliability of the candidates. Since it is obviously impractical to test every piece of datum in the possible input space outlined by corresponding operational profiles, the question can be broken down into the following two pieces: (1) How to decide possible test case space? and (2) How many test cases are sufficient and necessary to validate the full state of a remote Web service?

Our proposed solution is to utilize boundary values together with faulty data perturbed from boundary values to quickly verify the reliability of a Web service candidate. Each test case tests a Web service upon a function call whose signature contains several parameters, each requiring a specific data type with implicit boundary constraints. Our approach focuses on finding out the boundary values for each input parameter's data type. Let us examine a simple example: suppose that a Web service exposes a WSDL interface that includes a string-type parameter defined as follows:

```
<part name="firstName" type="xs:string"/>
```

For this parameter, we can test on boundary values such as: null, "" (empty string), short string (i.e., one character long), very long string (e.g., 100 characters long), string containing "new line" characters, non-string values (i.e., integer 5), etc.

For every WSDL interface exposed by a Web service, we list boundary values for each input parameter. Then we assemble all boundary values to obtain a list of test cases. For example, suppose a Web service interface contains five input parameters, each one being a string type without further constraints. As shown above, each parameter can have five boundary values. By assembling them together, we will get a list of twenty-five different test cases for the functional call.

These boundary values are definitely within the input domain. In order to test the fault tolerance of a Web service, we adopt the concept of fault injection. Injecting faulty data to verify fault tolerance is not new; traditional software testing establishes the fault injection technique [11, 12]. Here we first briefly review the concept of fault injection and then discuss the technical challenges we are facing in the domain of Web services. Derived from the technique used in traditional industry for a long time (e.g., automobile manufacture), fault injection is a set of techniques that provide worst-case predictions for how badly a system will behave in the future [11, 12]. More specifically, the Interface Propagation Analysis (IPA) technique proposed by Voas and colleagues is an advanced fault injection technique to test upon black-box-like software systems [13]. We believe that IPA is a right candidate concept to test the reliability of Web services due to the following reason: similar to normally called Commercial-off-the-shelf (COTS) components, users of Web services have no access to their internal source code. Users can only access Web services via SOAP request messages, and get results from Web services via SOAP response messages [14]. Therefore, Web services can be considered like black-box systems from users' perspectives.

The IPA technique suggests injecting corrupted data to the input of a black-box system [11], and monitoring the output of the system to obtain knowledge of its fault tolerance, as shown in Figure 1(a). IPA can help us test the vulnerability of a Web service serving as a component in a software system with respect to two levels: (1) the Web service in isolation, and (2) the Web service as a component interoperating with other parts of a system. As shown in Figure 1(b), the second level can be considered along with two scenarios: (a) when the Web service component returns corrupted information or no information at all, and (b) when the Web service fails to interoperate with other components of the system. In short, IPA can be applied to test the degree of how a system can tolerate a Web service as a component; or in other words, IPA can help test the interoperability of a Web service in a system.

However, although the basic concept of IPA seems appropriate to be applied to test both the fault tolerance and interoperability of Web services, how to apply the IPA technique in the domain of Web services remains a challenge. To our best knowledge, there still lacks a systematic approach to generate test cases to test fault tolerance of Web services. The core challenge of the IPA technique is how to create corrupted data for a testing component. Voas and colleagues propose to perturb the input domain to find corrupted data [11]. In traditional component-based testing, a

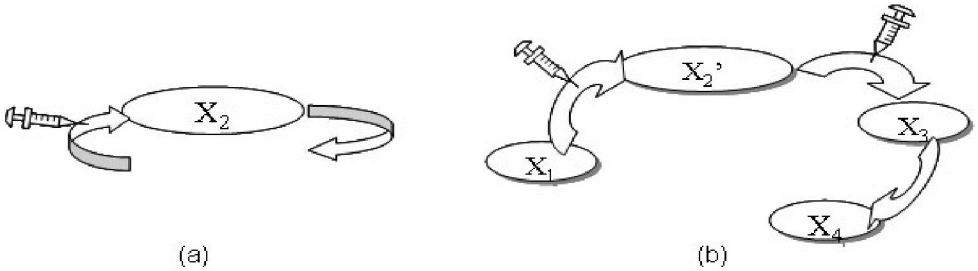

Figure 1. Injecting faulty data for interoperability testing

testing component is already deployed in its execution environment; thus, it is feasible to conduct an arbitrary amount of testing over the testing component. When we deal with Web services, on the other hand, we are facing remote Web components so that network traffic needs to be considered imperatively, let alone the fact that some Web services might have access charges associated. Furthermore, unlike traditional software components, Web services found from public registries oftentimes reveal limited information except for their access prototypes defined in WSDL.

Therefore, our strategy of designing faulty data to test the fault tolerance of a Web service focuses on perturbing the boundary values for each input parameter's data type. Let us examine a simple example: suppose that a Web service function requires a string-type input parameter with a length limitation of 8 to 16 characters. Eight and 16 character-long strings are both boundary values for the input parameter. Perturbing these two boundary values, we can obtain 7, 9, 15, and 17 character-long strings, which can be used as faulty data to test the fault tolerance of the Web service.

In summary, the faulty data should be divided into two sets with different purposes: (1) to test the Web service in isolation, as shown in Figure 1(a); and (2) to test the Web service as a component in the system environment, as shown in Figure 1(b). In order to test the vulnerability of a Web service in isolation, we will perturb each boundary value to generate faulty test cases, and monitor and analyze the post-condition of the Web service to decide whether the output events from the Web service is undesirable. It should be noted that a certain amount of testing should be performed to achieve a particular level of assurance. On the other hand, in order to test the interoperability of a Web service in its latter operating environment, specific operation scenarios and profiles need to be considered, in addition to our proposed boundary value perturbing approach. Exploring generating test cases based upon operation profiles is an area of future research.

III. DESIGN OF TEST CASES

In this section, we discuss detailed procedures of test case generation. To be specific, a test case of a Web service is a set of mappings between input variables and their values. Each test case can be used to generate a SOAP input message to test a corresponding Web service.

A. *Design of valid test cases for Web services*

As we discussed in the previous section, we generate valid test cases of a Web service by eliciting boundary values from the Web services interfaces written in WSDL. Here we first briefly examine the related WSDL specification on Web services interface definition.

WSDL is "an XML format for describing network services as a set of endpoints operating on messages containing either document-oriented or procedure-oriented information" [7]. As shown in Figure 2, using the WSDL, a Web service is defined as a set of ports, each publishing a collection of port types that bind to network addresses using common binding mechanisms. Every port type is a published operation that is accessible through messages. Messages are in turn categorized into input messages containing incoming data arguments and output messages containing results. Each message consists of data elements; and every data element must belong to a data type, either an XML Schema Definition (XSD) simple type or an XSD complex type. (Here we omit the fact that WSDL allows other data type system in addition to XSD, since XSD is its canonical type system.)

In summary, in order to design test cases for a Web service, our basis is its WSDL operations, input messages, and output messages. For simplicity, we do not consider designing test cases on the binding of the Web service.

Our basic strategy is to design test cases based upon boundary values of each formal argument of the published WSDL definition of the Web service. It should be noted that one major motivation is to enable automatic test case generation. Therefore, our challenge here turns into how to find efficient boundary values for each formal argument. Since each input parameter must be an XML-allowed data

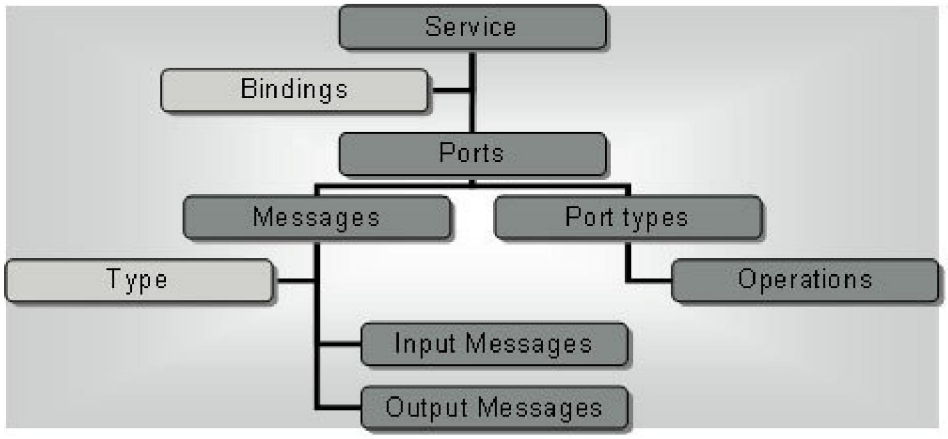

Figure 2. Web service interface definition using WSDL

type, it can be either XML built-in types or user-defined compound types, as shown in Figure 3. Let us discuss XML built-in type first.

B. *Boundary values for XML built-in primitive type*

As shown in Figure 3, XML built-in types include built-in simple types and built-in complex types. The former can be in turn divided into built-in primitive types and built-in derived types. A built-in complex type is defined in terms of built-in primitive types and built-in derived types by unioning their value spaces and lexical spaces. Built-in derived types actually depend on built-in simple types [15]. In other words, built-in primitive types are base types, and other types can be derived in terms of primitive types. Thus, we only need to investigate how to extract boundary values for built-in primitive types.

As shown in Table I, W3C Specification of the XML Schema language defines 19 built-in primitive types: string, decimal, boolean, duration, dataTime, time, date, gYearMonth, gYear, gMonthDay, gDay, gMonth, base64Binary, hexBinary, float, double, anyURI, QName, and NOTATION [15][16]. For each primitive type, W3C XML specification defines a set of constraining facets. Each constraining facet restricts an aspect of the value space of a built-in primitive type (e.g., minimum value, maximum value, etc). Taking *string* as an example, it has six constraining facets: length, minLength, maxLength, pattern, enumeration, and whiteSpace.

As summarized in Table I, there are altogether twelve kinds of constraining facets: (1) length: the number of units of length based upon data types, (2) minLength: the minimum number of units of length, (3) maxLength: the maximum number of units of length, (4) pattern: regular expression that restricts the lexical spaces to literals, (5) enumeration: a set of values, (6) whiteSpace: space, tab, line feed, and carriage return, (7) maxInclusive: inclusive upper bound, (8) minInclusive: inclusive lower bound, (9) maxExclusive: exclusive upper bound, (10) minExclusive: exclusive lower bound, (11) totalDigits: maximum number of digits, and (12) fractionDigits: maximum number of digits in the fractional part. Detailed information about each constraining facet can be found in W3C XML Schema [15].

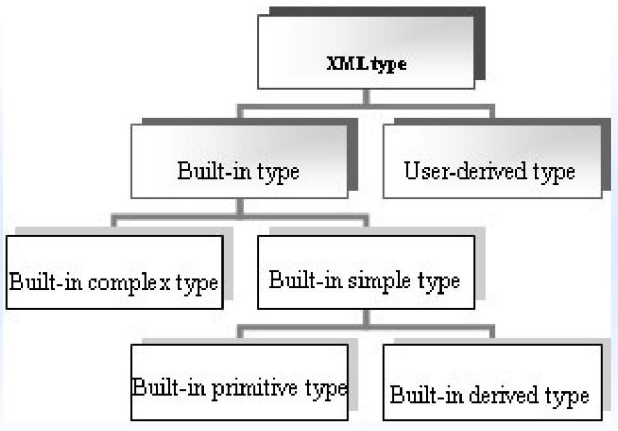

Figure 3. XML schema data types

We use these constraining facets as guidelines to identify boundary values. For example, consider a WSDL input argument that is an XML data type *string* with constraining facet of length: <length value = '6'>. We can identify a boundary value of a string with a 6-character long length. For each input parameter, we then search for its constraining facets. These constraining facets are part of the corresponding XML schema definition, which can be either included in the corresponding WSDL definitions, or referenced by separate XSD files. The keywords for the twelve constraining facets are utilized to search for the corresponding specifications. Using the example above, the keyword "length" can be used to search in the corresponding XSD specifications for the constraining facet *length* and its specified value of 6.

The twelve constraining facets can be divided into five categories based upon how they can be used to identify boundary value-based test cases. (1) Four constraining facets explicitly specify the boundary values to test: maxInclusive, minInclusive, maxExclusive, and minExclusive. (2) One constraining facet explicitly defines the set of values to test: enumeration. (3) Five constraining facets define the length of a test case: length, minLength, maxLength, totalDigits, and fractionDigits. (4) The *WhiteSpace* facet guides to generate test cases on spaces. (5) The *Pattern* facet guides to generate test cases based upon specified regular expressions. The first, second, and fourth categories explicitly define the boundary values that can be used. The third category specifies the length of test cases. The fifth category specifies the rules to validate test cases, which deserve further separate investigation and will not be discussed in this paper.

Therefore, for each input parameter defined in a WSDL document, we obtain a set of possible constraining facets from Table I based upon the XML data type of the parameter. The keywords of this set of possible constraining facets are used to search from the corresponding schema definitions for defined boundary values or rules. Note that the constraining facets summarized in Table I are possible facets for each data type. If, for a defined input parameter, there is no value defined for a possible constraining facet, an implicit constraint value should be used based upon the corresponding IEEE standards [15]. For example, consider an input parameter with type *float*. If there is no value defined for a possible constraining facet, say *maxInclusive*, an implicit value is actually defined. XML schema adopts for the type *float* the IEEE single-precision 32-bit floating point type. Thus, the basic value space of a *float* consists of the values $m \times 2^e$, where m is an integer whose absolute value is less than 2^24, and e is an integer between -149 and 104. Therefore, an implicit *maxInclusive* for a type *float* is 2^24 x 2^104 – 1 = 2^128 – 1. Similar rules are applied to other numeric XML data types, such as double. Similarly, ISO standards of Gregorian time values should be applied to time-related data types: duration, dateTime, time, date, gYearMonth, gYear, gMonthDay, gDay, and gMonth.

TABLE I CONSTRAINING FACETS OF XML BUILT-IN PRIMITIVE TYPES

	length	min-Length	max-Length	pattern	enume-ration	white Space	max-Inclusive	max-Exclusive	min-Exclusive	min-Inclusive	total Digits	fraction Digits
string	✓	✓	✓	✓	✓	✓						
boolean				✓	✓							
decimal				✓	✓	✓	✓	✓	✓	✓	✓	✓
float				✓	✓	✓	✓	✓	✓	✓		
double				✓	✓	✓	✓	✓	✓	✓		
duration				✓	✓	✓	✓	✓	✓	✓		
dateTime				✓	✓	✓	✓	✓	✓	✓		
time				✓	✓	✓	✓	✓	✓	✓		
date				✓	✓	✓	✓	✓	✓	✓		
gYearMonth				✓	✓	✓	✓	✓	✓	✓		
gYear				✓	✓	✓	✓	✓	✓	✓		
gMonthDay				✓	✓	✓	✓	✓	✓	✓		
gDay				✓	✓	✓	✓	✓	✓	✓		
gMonth				✓	✓	✓	✓	✓	✓	✓		
hexBinary	✓	✓	✓	✓	✓	✓						
base64Binary	✓	✓	✓	✓	✓	✓						
anyURI	✓	✓	✓	✓	✓	✓						
QName	✓	✓	✓	✓	✓	✓						
NOTATION	✓	✓	✓	✓	✓	✓						

For each XML primitive type, we extract its comprehensive boundary values from three dimensions: (1) XML constraining facets, (2) operational profiles, and (3) semantic meanings. XML constraining facets provide generic guidelines for us to find boundary values; and the operational profiles of a Web service will help us find more efficient boundary values. For example, let us consider a login id field with type *string*. From the XML constraining facets of *string*, we know that we need to test the length of the string. A specific operational profile can help us decide to test whether the string can accept more than 16 characters. In addition, operational profiles can help decide the boundary values for patterns testing, as defined by the corresponding XML constraining facets. Taking the login id field as an example again, the operational profiles may help to generate boundary values to test the string such as: whether the field accepts a string containing only digits, whether the field is case sensitive, whether the first character can be a digit, etc.

Finally, the semantic meanings of an argument can further facilitate boundary values elicitation. Taking an input field of credit card expiration year as an example, it is intuitive for us to test the following cases: whether the input year is a future year or a past year, whether the year is too far in the future, whether the combination of the year and the month represents a date in the future, whether the month is between 1 to 12, etc.

Although operational profiles and semantic meanings can facilitate more accurate and comprehensive boundary value elicitation, they mainly require manual involvement. On the other hand, constraining facets-based boundary value elicitation can mainly be performed through an automatic process, following the methods we discussed in this section. Regarding Web services testing, automatic test case generation is critical due to the unique time and dynamic feature requirements. In this research, we focus on test case generation based upon constraining facets-based approach. Automating test case generation based upon operational profiles and semantic meanings will be a future research topic.

C. *Boundary values for XML compound type*

As shown in Figure 3, based upon the 19 built-in primitive types, XML schema defines 25 built-in derived data types, such as normalizedString, token, language, etc [15]. In addition, complex data types can be composed of primitive types and derived types. Furthermore, users can define their own data types. In general, each user-derived data type must be defined in terms of another data type in one of three ways [15]: 1) by assigning constraining facets that restrict the value space of the user-derived data type to a subset of that of its base type; 2) by creating a list of data types whose value space consists of finite-length sequences of values of its item types; or 3) by creating a union data type whose value space consists of the union of the value space of its member types. In other words, each compound data type is associated with a hierarchical tree of how it is composed of simpler data types. Each leaf element of the tree is an XSD built-in primitive data type. Therefore, for a compound data type, we can navigate through its hierarchy tree and design test cases based upon each leaf element that is an XSD built-in primitive data type.

Figure 4 shows a simplified XSD compound data type StudentInfo. The personal information of a student contains four elements: her id as a double type, name as a string type, contact information as a complex type, and a list of addresses each as a complex type. Contact information is composed of

four elements: a *homePhone* as a string type, a *cellPhone* as a string type, a *fax* as a string type, and an *email* as a string type. Not including the simple data types expanded from the Address complex data type, there are six leaf elements in this StudentInfo data type: *id, name, homePhone,cellPhone, fax,* and *email.* Each element belongs to an XSD built-in primitive data type, either *double* or *string*. Then for each element, we can apply our method of designing boundary values for XSD built-in primitive data types, as we discussed in the previous section. Since we prefer to locate errors if there are any, each test case only focuses on testing one boundary value of one leaf element, without combining several boundary values of multiple elements. In other words, we have purposely limited the boundary values to a single parameter to avoid the number of combinatorial edge values that could be set at each SOAP input message from developing too fast. Accumulating all of these test cases together, we will obtain a set of test cases targeting testing the overall StudentInfo data type.

D. The design of test cases for fault tolerance of Web services

In this section we will discuss how to perturb boundary values to validate fault tolerance of an individual Web service. Our approach is again based upon XML schema constraining facets. In the last section we discussed how to extract boundary values from the WSDL definition of a Web service to efficiently test its correctness. These elicited boundary values can be perturbed to generate faulty data. Since our boundary values are generated from constraining facets, it is straightforward to generate faulty data in terms of constraining facets also. Table II summaries our methods of creating faulty data based on each constraining facet.

For *length*, since it defines the exact length of the characters/digits to be used in an argument, two test cases are generated, one with (length+1) and one with (length-1). For *minLength*, since it defines the minimum length of the digits to be used in an argument, one test case is generated with a smaller length of (minLength-1). For *maxLength*, since it defines the maximum length of the characters/digits to be used in an argument, one test case is generated with a larger length of (maxLength+1). For *enumeration*, since it defines explicitly the set of values to be used, we can generate one or more test cases with values out of the defined set. For *whiteSpace*, we can generate test cases with value null, one or multiple white spaces, or tabs. For *maxInclusive*, since it defines the largest value that can be used, one test case can be generated with a value of (maxInclusive+1). For *maxExclusive*, since it defines the largest value that cannot be used, one test case can be generated with a value of (maxInclusive). For *minExclusive*, since it defines the smallest value that cannot be used, one test case can be generated with a value of (minExclusive). For *minExclusive*, since it defines the smallest value that can be used, one test case can be generated with a value of (minInclusive-1). The approach to perturb regular expression patterns will not be discussed in this paper.

Table II summarizes our perturbation algorithm over each constraining facet. Recall that using the algorithm discussed in the previous section, a suite of test cases with boundary values will be generated. For each such test case, we iterate through each input argument, find out from which constraining facet it is generated, and perturb the data using the algorithm defined in Table II. Each perturbation creates a new test case. A suite of test cases can then be generated by combining all such test cases.

```
<element name="StudentInfo" type="tns:StudentInfoType"/>
<complexType name="StudentInfoType">
  <all>
    <element name="id" type="string/>
    <element name="name" type="string"/>
    <element name="contactInfo" type="ContactInfo"/>
    <element name="addresses">
      ...
    </element>
  </all>
</complexType>
<complexType name="ContactInfo">
  <all>
    <element name="homePhone" type="string"/>
    <element name="cellPhone" type="string"/>
    <element name="fax" type="string"/>
    <element name="email" type="string"/>
  </all>
</complexType>
```

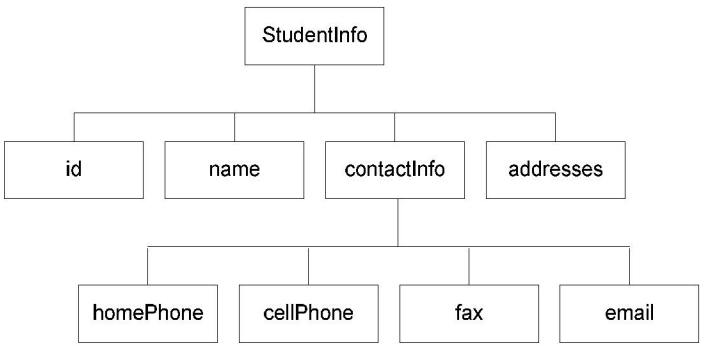

Figure 4. A simple example of XSD complex data type StudentInfo

TABLE II PERTURBATION STRATEGY TO GENERATE FAULTY DATA

Constraining facets	Perturbation strategy
length	+1/-1
minLength	-1
maxLength	+1
pattern	--
enumeration	Values outside of the set
whiteSpace	Null/tabs/space/multiple spaces
maxInclusive	+1
maxExclusive	The value
minExclusive	-1
minInclusive	Use the value
totalDigits	+1
fractionDigits	+1

It should be noted that the test cases with faulty data generated from our strategy obviously do not cover all faulty data domain. However, it is by no means our objective to test a Web service with all possible faulty data. Our goal is to find efficient faulty data to eliminate a Web service candidate. Our strategy covers faulty data violating constraining facets that definitely should be tested. In addition, as shown in Table II, our approach of generating faulty test cases can be easily automated, which meets the requirements of Web services testing. Faulty data can be further elicited from operational profiles and semantic meanings, which will be a topic of our future research.

IV. EXPERIMENTS

We carried out a series of experiments to verify the effectiveness and efficiency of our boundary value-based test case generation algorithm. We built a typical Web application, which is a student registration and records system where students can register for courses and retrieve course grades online.

As shown in Table III, we embedded six types of errors, (1) computational faults, such as changing a double-type value into a character-type value, (2) input SOAP processing faults, such as errors of parsing incoming SOAP messages, (3) output SOAP processing faults, such as errors of generating SOAP response messages, (4) data exception handling faults, such as improper handling over boundary values, (5) incorrect method calls, such as calling wrong methods, and (6) other errors, such as random errors. In order to facilitate the experiments, we carefully implant code to throw meaningful exceptions if an error is found. For each type, we embedded three different errors. The total number of seeded errors is 18.

We performed three categories of test case generation methods: (1) manually and randomly pick up test cases from the input space, (2) manually go through possible test cases from input data space, and (3) automatically generate test cases using our boundary value-based approach. Using our approach, the number of automatically generated test cases is 200. The results are shown in Figure 5.

We found that random test case selection is the least robust algorithm to find errors. As the number of test cases increased, the second exhaustive approach can find more and more errors. Meanwhile, it should be noted that both the first and the second algorithms have to go through a manual process of test case generation. As shown in Figure 5, we found that our boundary value-based test case generation approach is efficient in finding most errors. It found 16 out of 18 seeded errors (88.89%). When the number of test cases increased by randomly picking up more test cases in addition to automatically generated test cases, no more errors were found.

We also found that our algorithm is good at finding errors, such as SOAP processing faults, data exception handling errors, and incorrect method calls. Two computational errors were not found from our algorithm. The errors of class casting of grade from data type *double* to *string* were not found. In other words, it is difficult to test those errors

TABLE III DISTRIBUTION OF ERRORS SEEDED INTO THE SERVICES

Error type	Number of errors
Computational faults	3
Input SOAP processing faults	3
Output SOAP processing faults	3
Data exception handling faults	3
Incorrect method calls	3
Other errors	3

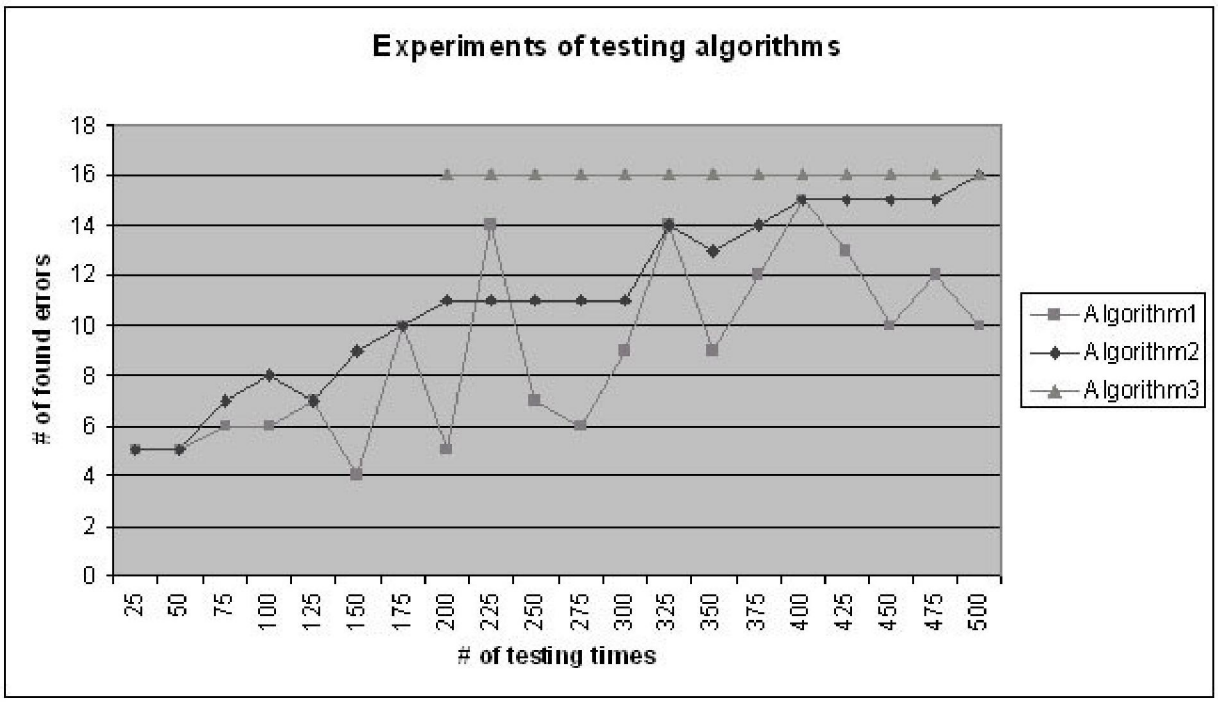

Figure 5. Comparison of different test case generation algorithms

depending on computational logic.

In order to further test the efficiency of the three algorithms, we chose to set up the reliability decision threshold to zero, which means that all test cases were conducted. The testing results are similar to that was described above.

Our preliminary experiments showed that our test case generation algorithm is effective and efficient.

V. COMPARISON WITH RELATED WORK

Casati et al. suggest that Web service providers define *service quality metrics*, which contain non-functional parameters specifying the cost, duration, and other characteristics of a service, to help service requestors make decisions over multiple candidates [17]. However, their work remains as a high-level abstraction without technical discussions such as how to construct *service quality metrics*.

Simulation has been utilized to validate and monitor Web services composition. Narayanan and McIlraith translate DAML-S service descriptions of composite services into a Petri nets formalism in order to provide decision procedures for Web services simulation, verification, and composition [18]. Cardoso and Sheth use simulation to validate Web services composition based upon a mathematical Quality of Service (QoS) model that emphasizes timeliness, cost of service, and reliability [19, 20]. Miller and colleagues focus on utilizing simulation analysis to monitor Web process composition [21]. Lerner uses parameterized state machine to verify process models [22]. Contrasted with their work, our research focuses on efficiently generating test cases to assess Web services and concentrating on reliability attribute only.

Offutt and Xu propose to adopt data perturbation technique to generate test cases of testing message communications between pairs of Web services. Data perturbation includes two approaches: data value perturbation modifies values according to the data types specified by Web services; and interaction perturbation tests RPC communication and data communication [23]. Their goal is to use mutation analysis to find faults from Web services. In contrast with their approach, our work aims to help service requestors automatically create test cases to select Web services found from public registries. From a service requestor's perspective, a Web service is a complete black box with its published WSDL definitions. Therefore, the basis of our test cases generation is the found WSDL definition files of the Web services. In addition, their research uses machine-related boundary values as data perturbation strategy (e.g., largest number for a double data type). Our research proposes a much finer-grain strategy to find boundary values based upon constraining facets and XML schema-referenced data type standards.

Bai and colleagues also explore how to generate test cases from WSDL documents [24]. Contrasted with their work focusing on generating functional test cases with valid data, our research focuses on efficiently generating reliability test cases with boundary values and faulty data.

Siblini and Mansour [25] define a set of mutation operators to guide test case generation for Web services testing using WSDL. Their major idea is to switch elements

of the same type in an input document to simulate faulty data. However, their work reports at a high level and only simulates incorrect element orders. Contrasted with their work, we analyze the boundaries and defined facets of WSDL documents to elicit faulty data input.

Our previous research [26] explored how to measure reliability of Web services using the techniques of mobile agents. This research explores how to generate test cases for Web services reliability assessment.

VI. CONCLUSIONS

In this paper we proposed a step-by-step boundary value-based approach to automatically generate valid and faulty data test cases for Web services reliability assessment. Our approach is appropriate for testing the reliability of Web services candidates with limited exposed interfaces. By perturbing the test data to imitate unusual events, our approach is capable of testing whether the hosts of Web services act maliciously or errantly at invocation times.

The generated faulty test cases can also be used to test other attributes (e.g., interoperability) of a Web service. As shown in Figure 1(b), in order to test the interoperability of a Web service (X_2), faulty data should be injected into X_2, and then we could monitor the output of X_2 and the output of its successor X_3, and so on.

Our future work will focus on exploring test case generation for testing other attributes of Web services such as availability, interoperability, and security.

REFERENCES

[1] P. Holland, "Building Web Services from Existing Application", *eAI Journal*, 2002: pp. 45-47.

[2] M. Stal, "Web Services: Beyond Component-based Computing", *Communications of the ACM*, 2002. 45(10): pp. 71-76.

[3] P. Fremantle, S. Weerawarana, and R. Khalaf, "Enterprise Services", *Communications of the ACM*, 2002. 45(10): pp. 77-82.

[4] J. Roy and A. Ramanujan, "Understanding Web Services", *IEEE IT Professional*, 2001: pp. 69-73.

[5] F. Curbera, R. Khalaf, N. Mukhi, S. Tai, and S. Weerawarana, "The Next Step in Web Services", *Communications of the ACM*, 2003. 46(10): pp. 29-34.

[6] SOAP, "Simple Object Access Protocol (SOAP) 1.1", World Wide Web Consortium (W3C), May 2000, http://www.w3.org/TR/SOAP.

[7] WSDL, "Web Services Description Language", 2004, http://www.w3.org/TR/wsdl.

[8] UDDI, "Universal Description, Discovery, and Integration, UDDI Specification Version 3", 2004, http://www.oasis-open.org/committees/uddi-spec/doc/tcspecs.htm#uddiv3.

[9] J. Zhang, "Trustworthy Web Services: Actions for Now", *IEEE IT Professional*, 2005: pp. 32-36.

[10] J.D. Musa, A. Iannino, and K. Okumoto, *Software Reliability Measurement Prediction Application*, 1987: McGraw-Hill.

[11] J. Voas and G. McGraw, *Software Fault Injection: Inoculating Programs Against Errors*, 1998: New York: John Wiley & Sons, ISBN 0-471-18381-4.

[12] J. Voas, "Certifying Off-The-Shelf Software Components", *IEEE Software*, 1998: pp. 53-57.

[13] J. Voas, F. Charron, and K. Miller, "Robust Software Interfaces: Can COTS-based Systems Be Trusted without Them?" *Proceedings of the 15th International Conference on Computer Safety, Reliability, and Security, Springer-Verlag*, Oct., 1996.

[14] SOAP, "Simple Object Access Protocol (SOAP) 1.2", World Wide Web Consortium (W3C), May, 2003, http://www.w3.org/TR/soap12-part1/.

[15] P.V. Biron and A. Malhotra, "W3C Recommendation "XML Schema Part 2: Datatypes"", 2001, http://www.w3.org/TR/2001/REC-xmlschema-2-20010502.

[16] W.C.D. Types, http://www.w3.org/TR/xmlschema-2/.

[17] F. Casati, M. Castellanos, U. Dayal, and M.-C. Shan, "Probabilistic, Context-sensitive, and Goal-oriented Service Selection", *Proceedings of the 2nd ACM International Conference on Service Oriented Computing*, New York, NY, USA, 2004, pp. 316-321.

[18] S. Narayanan and S.A. McIlraith, "Simulation, Verification and Automated Composition of Web Services", *Proceedings of the eleventh ACM International Conference on World Wide Web (WWW)*, Honolulu, Hawaii, USA, May 7-11, 2002, pp. 77-88.

[19] J. Cardoso and A. Sheth, "Semantic E-Workflow Composition", *Journal of Intelligent Information Systems*, 2003. 21(3): pp. 191-225.

[20] J. Cardoso, A.P. Sheth, J.A. Miller, J. Arnold, and K.J. Kochut, "Modeling Quality of Service for Workflows and Web Service Processes", *Web Semantics Journal: Science, Services and Agents on the World Wide Web Journal*, 2004. 1(3): pp. 281-308.

[21] J. Miller, J. Cardoso, and G. Silver, "Using Simulation to Facilitate Effective Workflow Adaptation", *Proceedings of 35th Annual Simulation Symposium*, San Diego, CA, USA, 2002, pp. 177-181.

[22] B.S. Lerner, "Verifying Process Models Built Using Parameterized State Machines", *Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis*, Boston, MA, USA, 2004, pp. 274-284.

[23] J. Offutt and W. Xu, "Generating Test Cases for Web Services using Data Perturbation", *Proceedings of Workshop on Testing, Analysis and Verification of Web Services (TAV-WEB)*, Jul., 2004, pp. 1-10.

[24] X. Bai, W. Dong, W.-T. Tsai, and Y. Chen, "WSDL-Based Automatic Test Case Generation for Web Services Testing", *Proceedings of IEEE International Workshop on Service-Oriented System Engineering (SOSE)*, Beijing, China, Oct. 20-21, 2005, pp. 215-220.

[25] R. Siblini and N. Mansour, "Testing Web Services", *Proceedings of The 3rd ACS/IEEE International Conference on Computer Systems and Applications*, Cairo, Egypt, Jan. 3-6, 2005, pp. 135-142.

[26] J. Zhang, L.-J. Zhang, and J.-Y. Chung, "An Approach to Help Select Trustworthy Web Services", *Proceedings of IEEE International Conference on E-Commerce Technology for Dynamic E-Business (CEC 2004 East)*, Beijing, China, Sep. 13-15, 2004, pp. 84-91.