Generating Test Cases for Web Services Using Data Perturbation*

Jeff Offutt & Wuzhi Xu
Department of Information and Software Engineering
George Mason University
Fairfax, VA 22030-4444 USA
ofut@ise.gmu.edu, wxu2@gmu.edu

ABSTRACT

Web services have the potential to dramatically reduce the complexities and costs of software integration projects. The most obvious and perhaps most significant difference between Web services and traditional applications is that Web services use a common communication infrastructure, XML and SOAP, to communicate through the Internet. The method of communication introduces complexities to the problems of verifying and validating Web services that do not exist in traditional software. This paper presents a new approach to testing Web services based on **data perturbation**. *Existing XML messages are modified based on rules defined on the message grammars, and then used as tests. Data perturbation uses two methods to test Web services: data value perturbation and interaction perturbation. Data value perturbation modifies values according to the data type. Interaction perturbation classifies the communication messages into two categories: RPC communication and data communication. At present, this method is restricted to peer-to-peer interactions. The paper presents preliminary empirical evidence of its usefulness.*

keywords: Web Services, XML, SOAP, Data perturbation

1. INTRODUCTION

Web applications run large-scale software applications for e-commerce, entertainment, and numerous other activities [13]. They run on distributed hardware platforms and heterogeneous computer systems. The addition of Web services provide a common communications infrastructure that is supported by many Web application vendors. They are based on a distributed environment that does not require a specific underlying technology platform for development or deployment.

Testing Web services is difficult because they are distributed applications with numerous runtime behaviors that are different from traditional applications. A service subscriber has to use black-box testing because the design and implementation details of Web services are not available. A single business process often involves multiple Web services. Moreover, these multiple Web services might be located on different servers, and belong to different companies. Web services interact by passing messages and data in structured ways. Although technologies exist to verify syntactic aspects of the interactions, the problem of whether the two Web services behave correctly with all possible messages is more difficult. This paper presents a novel technique for testing the interaction between pairs of Web services. Our approach is called data perturbation, and is based on modifying values in request messages, then analyzing the values in response messages.

The rest of this paper presents details about the data perturbation method. Some general background on Web services and testing is given, with specific definitions for the terms used in this paper. Then a model is constructed for Web service interactions, which is used to develop tests. The paper also presents initial results from an empirical investigation of data perturbation.

2. BACKGROUND

This section provides an overview of Web services, XML, and XML schemas. Then some of the early attempts to test Web services are presented.

2.1 Web Services and XML Schemas

One difficulty with scientific research in this area is the many definitions for Web services [10], many of which are contradictory and imprecise. This is often acceptable for practitioners who need to express concepts in a local context, but generally accepted definitions are required for scientists to develop ideas and results in a global context. The term "Web service" is also used in different ways by writers who have only passing familiarity with the technology. As a working definition for our research, we define a *Web service* to be an Internet-based, modular application that uses the Simple Object Access Protocol (SOAP) for communication and transfers data in XML through the Internet.

*This work was supported in part by the U.S. National Air and Space Administration, GSFC, under contract grant 220873 to Indus Corporation, subcontracted to George Mason University.

The goal of this communication is to allow Web services to be described, advertised, discovered and invoked through the Internet. This infrastructure uses several technologies to let Web services function together. The *Extensible Markup Language (XML)* [18] is used to transmit messages and data. The *Universal Description, Discovery and Integration (UDDI)* specification is used to maintain directories of information about Web services. These directories record information about Web services, including location and requirements, in a format that other Web services and applications can read. The *Web Services Description Language (WSDL)* is used to describe how to access Web services and what operations they can perform. The *Simple Object Access Protocol (SOAP)* helps software transmit and receive XML messages over the Internet.

Figure 1 illustrates the roles of these technologies for Web services. The components and legacy system are "wrapped" into Web services by using SOAP and XML wrappers. The components are put into a Web service server container and compiled by this server container. Then the server container generates a WSDL file for this Web service and publishes this WSDL file onto the Internet. The process of putting a component into a Web service server container then publishing the WSDL file onto the Internet is called "wrapping a component using a SOAP interface," because other applications can now use SOAP to call this Web service. The UDDI registry holds the specification of services and the URL that points to the WSDL specification of services. The applications can look up a Web service in the UDDI registry. After applications have the URL and specification of a Web service, they can use XML and SOAP to communicate with that service.

Like HTML, XML uses tags, which are textual descriptions of data enclosed in angle brackets ('<' and '>'). However, unlike HTML, XML follows strict SGML syntax. XML documents must be *well-formed*, that is, have a single document element with other elements properly nested under it, and every tag must have a corresponding closing tag. A simple example XML document for books is shown in Example 1. This example is used throughout the paper to illustrate the model and test strategy.

EXAMPLE 1. *:* **Simple XML for Books**

```
<?xml version="1.0" encoding="UTF-8"?>
  <!--Sample XML file for books-->
<books xmlns:xsi=
            "http://www.w3.org/2001/XMLSchema-instance"
   xsi:noNamespaceSchemaLocation="C:\Books\books.xsd">
 <book>
    <ISBN>0-672-32374-5</ISBN>
    <price>59.99</price>
    <year>2002</year>
 </book>
 <book>
    <ISBN>0-781-44371-2</ISBN>
    <price>69.99</price>
    <year>2003</year>
 </book>
</books>
```

XML documents can be constrained by grammar definitions. The grammars can be defined in either Document Type Definitions (DTD) or the more recent XML schema [21]. XML schemas provide more facilities for things like data types than DTDs do. Programmers can validate XML documents against either XML DTDs or schemas. Example 2 shows a schema for books.

The World Wide Web Consortium (W3C) has released several different recommendations concerning XML, including the XML recommendation [18], the Namespace recommendation [17], the Infoset recommendation [20], the XPath recommendation [22], and the Schema recommendation [21].

EXAMPLE 2. *:* **XML Schema for Books**

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
    elementFormDefault="qualified"
    attributeFormDefault="unqualified">
 <xs:element name="books">
  <xs:annotation>
   <xs:documentation>XML Schema for Books
       </xs:documentation>
  </xs:annotation>
  <xs:complexType>
   <xs:sequence>
    <xs:element name="book" maxOccurs="unbounded">
     <xs:complexType>
      <xs:sequence>
       <xs:element name="ISBN" type="xs:string"/>
       <xs:element name="price" type="xs:double"/>
       <xs:element name="year" type="yearType"/>
      </xs:sequence>
     </xs:complexType>
    </xs:element>
   </xs:sequence>
  </xs:complexType>
 </xs:element>

 <xs:simpleType name="yearType">
  <xs:restriction base="xs:int">
   <xs:totalDigits value="4"/>
  </xs:restriction>
 </xs:simpleType>
</xs:schema>
```

2.2 Testing Web Services

One way to describe Web services is that the components are wrapped with SOAP interfaces so they can exchange XML-based messages. This is simple and reasonably accurate, but it masks some of the complexities. To consider their complexities, we need to consider how traditional software evolves to Web services. Aoyama describes three evolutionary ways [1]: (1) from components to Web services, (2) from application service provider to Web server, and (3) from stand-alone application to applications that are integrated across organizations. In each of these, Web services are used to publish traditional software on the Internet or to integrate subsystems within an organization. One ramification is that Web services are more widely distributed than traditional software. The fundamental objective of using Web services today is the same as that of distributed computing technologies 20 years ago: to allow applications to work cooperatively with other applications over a common network [7]. However, these three methods of software evolution highlights some differences between Web services and traditional software.

A major difference that affects testing is that companies use Web services to reduce the cost of integrating heteroge-

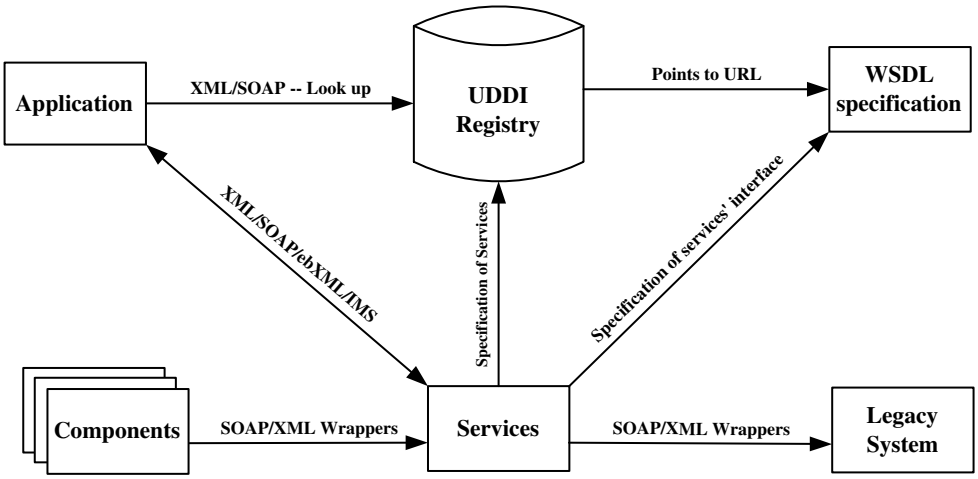

Figure 1: The Roles of Technologies in Web Services

neous subsystems across the organizations [1, 5, 23]. Web services can be considered to be **more** heterogeneous than Web applications. Web applications use multiple programming languages, but Web **services** also use different operating systems, different server containers [23].

Another difference is based on the fact that Web services do not have user interfaces [8]. This difference has more of an impact on a practical level than a theoretical level by affecting **how** testing is applied. Traditional applications and Web applications always have UIs for testers to input values. However, Web services use XML and SOAP to communicate directly from software component to software component. To be able to find, request, and receive "services" automatically and dynamically, they use a dynamic and loosely coupled Service-Oriented Architecture (SOA) [4, 7]. That is, Web services features **dynamic integration**. This is different from earlier distributed computing architectures that link applications together with static connections.

Current IT projects often treat Web services as being made up of special kinds of software components that "expose" their functionality via SOAP interfaces. Some testing techniques that are used to test software components are being extended to Web services. Some testing techniques for distributed applications are also used to test Web services. A few papers [5, 8, 4] have presented testing techniques for Web services, but the dynamic discovery and invocation capabilities of Web services bring up many testing issues. Three main aspects of Web services must be tested: (1) the discovery of Web services, (2) the data format exchanged, and (3) the request/response mechanisms. Bloomberg discusses several issues involving testing Web services [4, 5]:

- *Testing SOAP messages.* This activity targets the request/response mechanism aspect of Web services. Since Web services do not have UIs, testers must use SOAP to supply test cases. When testing SOAP messages, the tester should also test the format of the messages. Lee & Offutt presented an initial test approach for testing software that communicate with XML messages [11] (although not relying on SOAP).
- *Testing WSDL files and using WSDL files for test plan generation.* Businesses use WSDL to expose specific application programming interfaces as services available on the Internet. As a side benefit, testers can use the information in WSDL files to generate black box test plans [16].
- *Testing the publish, find, and bind capabilities of an SOA.* A fundamental characteristic of SOAs is the "publish, find, and bind" triangle. A Web **service** provider publishes WSDL files of Web services in a UDDI registry, where Web services **consumers** can find it. The consumers then *bind* to the services based on the WSDL file in the UDDI registry. From a testing perspective, this is a new way to *integrate* software that requires new testing techniques.
- *Web services consumer and producer emulation.* Test tools can emulate the consumer of a Web service by sending test messages to another Web service. In turn, test tools can emulate the producer of the Web service by returning a response message to the other Web service after the consumer sends a request message.

Bloomberg [4, 5] also presents other testing capabilities, such as testing synchronous and asynchronous capabilities, testing SOAP intermediaries, testing service-level agreement (SLA) and quality of services (QoS).

Most of these current testing tools currently focus on testing SOAP messages, testing WSDL files, and emulating consumer and producer emulation. Most Web services testing tools that test SOAP messages focus on testing RPC communications, and do not include general XML data communications. The research presented in this paper uses data perturbation to test the integration of software components that rely on **both** kinds of communications.

3. XML MESSAGE MODEL

Web services interact by passing messages that exchange data and activity state information. Figure 2 illustrates an

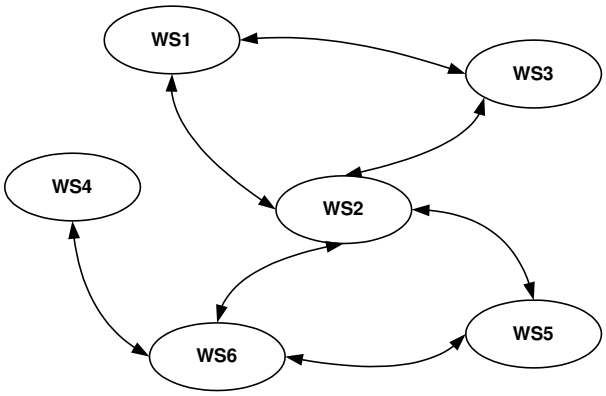

Figure 2: Multilateral Web Services Interaction

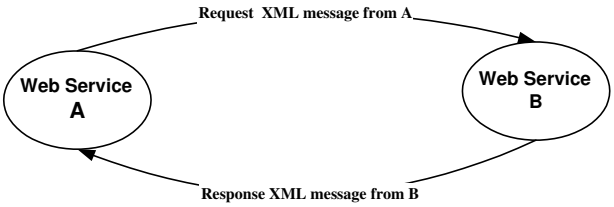

Figure 3: Peer-to-Peer Web Service Interaction Model

application that has six Web services. The arrows indicate possible message flows among the services. A *peer-to-peer interaction* is a message between two Web services. A *multilateral interaction* is a sequence of messages among multiple Web services, and are formed by aggregating multiple peer-to-peer interactions. As a first step, our research focuses on testing peer-to-peer interactions; multilateral interactions will be addressed in future work.

Figure 3 illustrates a simple peer-to-peer interaction between two Web services. Web service A sends a request to Web service B under HTTP over the Internet. This request might be a message for a remote procedure call or an XML data document. Web service B sends a response message back to A. This response might be a computation result from a remote procedure call or an XML document. Web services organize both kinds of messages in SOAP.

Peer-to-peer tests are based on XML messages. To construct them, we need a formal model for XML documents. Several formal models have been created [2, 6, 12, 14], each to satisfy a different purpose. So far these models are primarily syntactic in nature and capture information about structure and types. Syntactic models are appropriate for XML as it is seldom used to describe behavior. Makoto [12] and Chidlovskii [6] give two similar models, which we extend for testing. Our formal model is a regular tree grammar (RTG). We use RTG as a formal representation of an XML schema, and derive an XML document tree based on the RTG. A regular tree grammar (RTG) is a 6-tuple $< E, D, N, A, P, n_s >$, where:

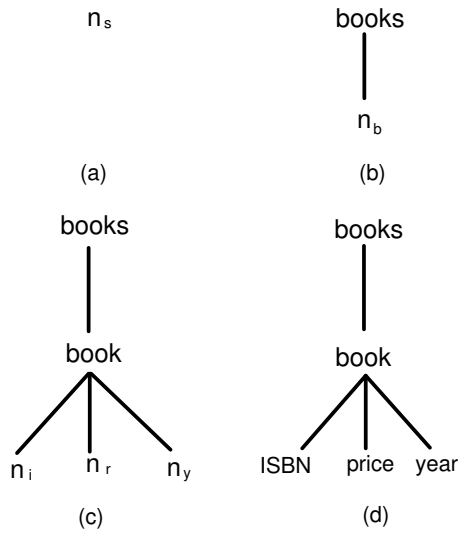

Figure 4: XML Document Tree Derivation

1. E is a finite set of element types
2. D is a finite set of data types
3. N is a finite set of non-terminals
4. A is a finite set of attribute types
5. P is a finite set of production rules of two forms:
   - $n \rightarrow a < d >$, where n is non-terminal in N, a is either an attribute type in A or an element type in E, and d is a data type in D
   - $n \rightarrow e < r >$, where n is non-terminal in N, e is an element in E, and r is a regular expression comprised of non-terminals
6. n_s is the starting non-terminal, $n_s \in N$

An RTG model can be created automatically from an XML schema or DTD. Given a sequence of non-terminals, we can repeatedly replace non-terminals with the right-hand side of the corresponding production rules. Consider the Books schema from Example 2 in Section 2.1. It is captured by an RTG $G =< E, D, N, A, P, n_s >$ where:

- $E = \{books,\ book,\ ISBN,\ price,\ year\}$
- $D = \{string,\ double,\ yearType\}$
- $N = \{n_s,\ n_b,\ n_i,\ n_r,\ n_y\}$
- $A = \{string,\ double\ yearType\}$
- $P = \{n_s \rightarrow books < (n_b)^* >$
  $n_b \rightarrow book < n_i\ n_r\ n_y >$
  $n_i \rightarrow ISBN < string >$
  $n_r \rightarrow price < double >$
  $n_y \rightarrow year < yearType >\}$

Figure 4 shows the derivation steps for the book XML document tree. The derivation starts with the starting non-terminal n_s in Figure 4.a. The productions for n_s and n_b are shown in Figures 4.b and 4.c. The productions for n_i, n_r, and n_y are in Figure 4.d.

RTG has two kinds of production rules: production to a non-leaf and production to a leaf. The non-leaf productions

reflect what are known as *entity relationships* (ER) in the database literature. However, XML schemas use a slight change to the usual ER semantics [2]. Database papers usually express the relationships between two entities with a foreign and primary key, XML schemas do not.

4. DATA PERTURBATION (DP)

Data Perturbation (DP) is our primary method for testing Web service components. The process works by modifying request messages, resending the modified request messages, and analyzing the response messages for correct behavior. Messages that supply traditional RPC calls using SOAP are said to be *RPC communications*. Web services are sometimes programs that transfer data from place to place, for example, Java servlets. Messages whose purpose is to transfer data are called *data communications*. In data communications, WSDL files are not enough for testing because WSDL files do not have information about the message format. Most Web services testing tools that test SOAP messages focus on testing RPC communications, and do not include data communication. This research uses data perturbation to test **both** kinds of communications.

To do so, data perturbation includes both data value perturbation and interaction perturbation. *Data value perturbation* modifies values in SOAP messages in terms of the types of the data. *Interaction perturbation* modifies messages in RPC and data communications.

There are some differences between RPC communications and data communications that affect the practical aspects of testing. Primarily, the method needed to extract data from a message is different. Values in RPC communications are directly extracted by the remote procedure functions. Programmers do not need to parse the messages. Values in data communications must be parsed by the program.

Another difference is in the relationships between message elements. RPC messages have a flat format. In terms of the tree structure, RPC data elements are all siblings, whereas data communication often use XML, which allows a more complicated hierarchical structure among the elements. Data communication messages can include information about data entity relationships and constraints. Data types for data communications messages are specified in XML schemas that two Web services agree to follow. When two Web services use data communications, they must have an agreement on the message format, which is usually encoded in an XML schema or DTD. RPC communications have no agreement on their message format.

The next three subsections describe variants of data perturbation. Data value perturbation modifies values, RPC communication perturbation tests data uses, and data communication perturbation tests the relationships and constraints on the data.

4.1 Data Value Perturbation (DVP)

Data value perturbation (DVP) modifies the values in SOAP messages according to rules defined on the types of the values. Data value perturbation relies on ideas from boundary value testing [3]. W3C's XML datatype recommendation [19] lists 19 primitive data types. Our research considers all 19 types, but due to space limitations we only list rules for the five types that correspond to primitive types in most programming languages. Test cases are derived from default boundary values of XML schemas. The boundary values for numeric (decimal, float and double), string and boolean types are listed in Table 1. Tests are created by replacing each value with each boundary value, in turn, for the appropriate type.

Data Type	Boundary Values
String	Maximum length, minimum length, upper case, lower case
Numeric	Maximum value, minimum value, zero
Boolean	true, false

Table 1: Boundary Values for Data Value Perturbation

In Table 1, the *maximum* and *minimum length* for Strings are the allowable maximum and minimum length of strings in XML schemas. The *upper* and *lower case* values are interpreted as operators that set the case for every letter in the string. The *maximum* and *minimum value* for numeric values are the allowable maximum and minimum values in XML schemas. The *zero* operators changes numeric values to zero. *Boolean* type has two values: true and false. The XML document for a book from Examples 1 and 2 uses three data types: string, int and double. Some of the DVP test cases that would be created are shown in Table 2 (due to space restrictions, only representative values are shown, not all). The string "—" represents a null value.

4.2 RPC Communication Perturbation (RCP)

Web services are often used to simplify remote procedure calls (RPCs) by using SOAP to format and transmit the calls. The RPC messages include values for the arguments of remote procedure functions. As previously said, the elements in RPC messages are siblings, and values for these elements are directly extracted by the functions. The response messages are results that are computed by the functions. RPC communication perturbation focuses on testing the data uses, as broken into two types: normal data uses and SQL uses. Normal data uses are when values are used within programs, and *SQL uses* occur when strings are used as inputs to a database.

Mutation analysis [9] is a software testing technique that creates modified versions of programs, called mutants, and then asks the tester to find test inputs to cause the mutant programs to fail. In the past, mutation has only been used on program text. This research uses the same concept, making small modifications to syntactic objects, but applies it to modify **values** instead of programs. This constitutes a major advance and divergence in the application of mutation analysis.

Data perturbation is used to generate values that are related to normal data uses, and SQL injection [15] is extended to test SQL uses. Traditional mutation operators modify source code, not data, so different types of operators have been defined.

Original Value	Perturbed Values	Test Case
<ISBN>0-672-32374-5</ISBN>	00000000000000000000000000	maximum length
	—	minimum length
<price>59.99</price>	$2^{63}-1$	maximum value
	-2^{63}	minimum value
	0	zero
<year>2002</year>	$2^{32}-1$	maximum value
	-2^{32}	minimum value
	0	zero

Table 2: Book Document Tests for Data Value Perturbation

First, we present an abstract definition of a mutation operator that can apply to both program text mutation and data perturbation. Formally, an abstract mutation operator is defined as:

DEFINITION 1. *Given a set of all element instances N, a mutation operator is $r = f(n_1, \ldots, n_i)$, where f is a function, $i \geq 1$, each $n_1, \ldots, n_i \in N$ and has the same data type, and r has the same data type as $n_1, \ldots, n_i$.*

The following numeric data type operators have been defined:

- *Divide(n)*: Change value n to $1 \div n$, where n is double data type
- *Multiply (n)*: Change value n to $n \times n$
- *Negative (n)*: Change value n to $-n$
- *Absolute (n)*: Change value n to $\mid n \mid$

When a Web service has two arguments of the same type, then can be exchanged. An *SQL injection* occurs when an SQL statement is included in a string value that is submitted to a Web application or service. The expectation is that the string will be stored in a database, and the foreign SQL statement will be executed. This can allow unauthorized access to databases on the server, with significant potential consequences to the server and database. Spett described several common SQL injection techniques [15]. We extend *authorization bypass* to SQL uses and define it with the operator *Unauthorized()*. The definitions of *Exchange()* and *Unauthorized()* are:

- *Exchange (n_1, n_2)*: Replace value n_1 with n_2 and vice versa, where both n_1 and n_2 have the same type
- *Unauthorized (str)*: Change string value *str* to $str' \ OR \ '1' = ' \ 1$

Example 3 contains a SOAP message that requests a login. The values in the message are used in an SQL query to verify whether the username and password are valid.

EXAMPLE 3. *:* **A SOAP Message for Login Request**

```
<?xml version="1.0" encoding="UTF-8"?>
<soapenv:Envelope
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
 <soapenv:Body>
   <adminLogin soapenv:encodingStyle=
              "http://schemas.xmlsoap.org/soap/encoding/">
     <arg0 xsi:type="xsd:string">turing</arg0>
     <arg1 xsi:type="xsd:string">enigma</arg1>
   </adminLogin>
 </soapenv:Body>
</soapenv:Envelope>
```

The *Unauthorized()* operator modifies the example in the following way:

EXAMPLE 4. *:* **The Modified SOAP Body of Login Message**

```
.............
 <soapenv:Body>
   <adminLogin soapenv:encodingStyle=
           "http://schemas.xmlsoap.org/soap/encoding/">
      <arg0 xsi:type="xsd:string">turing' OR '1'='1</arg0>
      <arg1 xsi:type="xsd:string">enigma' OR '1'='1</arg1>
   </adminLogin>
 </soapenv:Body>
.............
```

The original SQL query looks like:

```
SELECT username FROM adminuser WHERE username='turing' AND
password ='enigma'
```

After the *Unauthorized()* operator is used, the SQL query becomes:

```
SELECT username FROM adminuser WHERE username='turing' OR
'1'='1' AND password ='enigma' OR '1'='1'
```

The first SQL query selects just the record where username = "turing" and password = "enigma", and returns a null table if that combination is not in the database. The second SQL query selects **all** usernames from the table adminuser.

4.3 Data Communication Perturbation (DCP)

Most test tools for SOAP messages focus on testing RPC communications and do not include data communication. However, data communication is an important part of Web service interaction and must be tested. As an example, consider an online book store. The Web server for the book store collects information for a customer's order in XML format, organizes the XML document in SOAP format, and sends the SOAP message to the book publishers' servers. The XML document includes information about the customer and the order. The purpose of data communication is to transfer data, so it usually includes more data than RPC communication messages do. For example, it is common to include database relationships and constraints. Therefore, data communication perturbation focuses on testing these relationship and constraints. However, the semantics of these relationship and constraints are slightly different from those in the database literature [2].

To handle this difference, we use the RTG formal model to define the messages. The relationships and constraints are included in the finite set of production rules p in the RTG. The production rule $n \rightarrow a\langle d \rangle$ is a production rule from a non-terminal element to a terminal element or an attribute. $n \rightarrow e\langle r \rangle$ is a production rule from a non-terminal element to another non-terminal element. Formally, the relationship and constraints can be defined with RTG as follows:

DEFINITION 2. *Given an XML schema $G =< E, D, N, A, P, n_s >$, a relationship is a production rule in P: $n \rightarrow e < r >$, where n is a non-terminal in N, e is an element in E, and r is a regular expression made up of non-terminals.*

DEFINITION 3. *Given an XML schema $G =< E, D, N, A, P, n_s >$, a constraint is a production rule in P: $n \rightarrow \beta < \tau >$, where n is a non-terminal in N, β is an element in E or an attribute in A, and τ is a data type in D.*

For relationships, we focus on testing the referential integrity, which is the set of rules that govern the relationships between primary keys and foreign keys of tables within a relational database. They are used to determine data consistency. XML schemas define relationships as parent-child associations between two non-terminal elements. An XML schema uses *maxOccurs* to specify referential relationships between parent elements and child elements. In a regular expression of a relationship, the '?', '+', and '*' operators denote zero-or-one, at least one, and any number of element occurrences. These operators reflect cardinality constraints in an XML schema. In term of these operators, three testing strategies are defined as follows:

- Given a relationship $n \rightarrow e\langle r \rangle$, if there is an expression $\alpha?$ in r, there will be two test cases. One contains one α instance and the other contains an empty instance.
- Given a relationship $n \rightarrow e\langle r \rangle$, if there is an expression $\alpha+$ in r, there will be two test cases. One contains one α instance and the other one contains an allowable number of α instances.
- Given a relationship $n \rightarrow e\langle r \rangle$, if there is a expression α^* in r, there will be two test cases. One contains $\alpha^*\alpha$ and the other contains α^{*-1}, where $\alpha^*\alpha$ duplicates one element instance and α^{*-1} deletes one element instance.

The RTG for the book XML schema contains two relationships, $n_s \rightarrow books\langle(n_b)^*\rangle$ and $n_b \rightarrow book\langle n_i\ n_r\ n_y\rangle$. Example 1 for the book XML schema contains two instances for the element *book*. Test cases will duplicate one element instance and delete one element instance. Two test case examples are shown below in Examples 5 and 6.

EXAMPLE 5. *:* **Test Case for Duplicating Book Instance**

```
<?xml version="1.0" encoding="UTF-8"?>
<!--Sample XML file for books-->
<books xmlns:xsi=
            "http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="C:\Books\books.xsd">
  <book>
    <ISBN>0-672-32374-5</ISBN>
    <price>59.99</price>
    <year>2002</year>
  </book>
  <book>
    <ISBN>0-781-44371-2</ISBN>
    <price>69.99</price>
    <year>2003</year>
  </book>
  <book>
    <ISBN>0-781-44371-2</ISBN>
    <price>69.99</price>
    <year>2003</year>
  </book>
</books>
```

EXAMPLE 6. *:* **Test Case for Deleting Book Instance**

```
<?xml version="1.0" encoding="UTF-8"?>
<!--Sample XML file for books-->
<books xmlns:xsi=
            "http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="C:\Books\books.xsd">
  <book>
    <ISBN>0-672-32374-5</ISBN>
    <price>59.99</price>
    <year>2002</year>
  </book>
</books>
```

Database constraints are rules that govern the expressions of attributes in tables within a relational database. For example, *null not allowed* and *non-null uniqueness* are often used in a database. In XML schemas, the definition of constraints is slightly different. A constraint in an XML schema is a parent-child association between a non-terminal element and a terminal element. All component constraints defined in W3C's XML datatype recommendation [19] are considered.

Consider the XML schema for books. The production set P contains three constraints: (1) $n_i \rightarrow ISBN\langle string\rangle$, (2) $n_r \rightarrow price\langle double\rangle$, and (3) $n_y \rightarrow year\langle yearType\rangle$. The first two production rules have no component constraints. The component constraint for the third production rule is $\langle xs : totalDigitsvalue = "4"\rangle$. The test case created for this component constraint is 9999.

5. EMPIRICAL RESULTS

An initial proof-of-concept empirical study has been carried out. We have built a small collection of Web services, the Mars Robot Communication System (MRCS), and applied data perturbation to create tests. MRCS is an experimental system that simulates support for a scientific mission to Mars by carrying geological and meteorological data from Mars to a space station in orbit around Earth, then to a ground control system that stores the data in a database and provides access to scientists. MRCS has three separate components: A robot on Mars, the space station, and ground control in Houston. The ground control system is a 3-tier Web application that includes the Web server, an application server, and a database server. All communications between the three components and among the servers in the ground control system are in XML under SOAP. The interaction diagram for MRCS is shown in Figure 5.

MRCS contains five Web services that use RPC communication, six servlets that use data communication, and two Java beans. All servlets output XML messages and five of them also produce an HTML Web page. The MRCS has six primary functions:

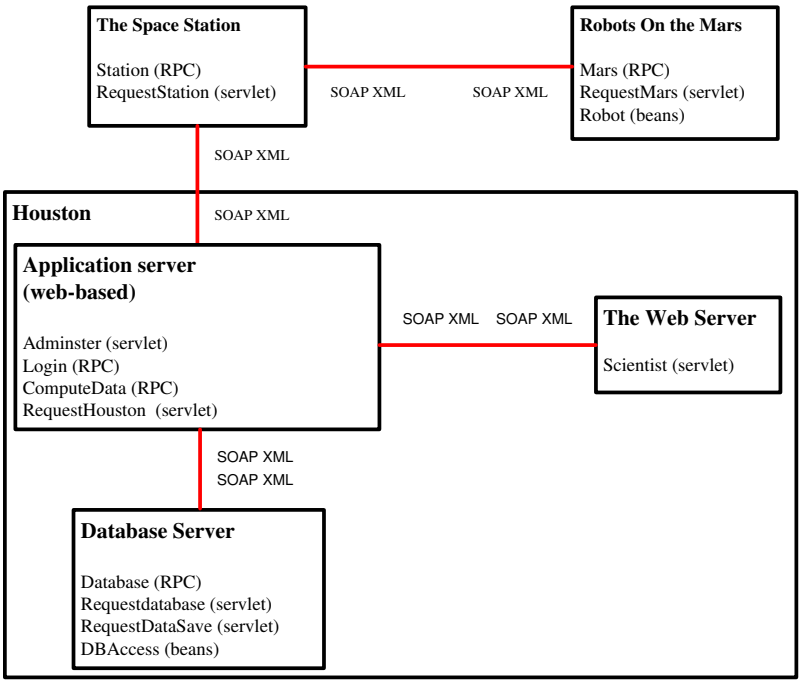

Figure 5: The Interaction Diagram of MRCS

1. Administrator Login: Administrators need to log into the application server on the ground before commanding the robot on Mars or accessing the database
2. Command the Robot: Administrators operate the robot on Mars from the application server in the ground control system.
3. Data request from the ground: Administrators request data that the Robot collected about its own status or Mars.
4. Scientist Login: Scientists must login from the Web server in the ground control system before they can access the database or carry out computations.
5. Data request from scientists: Scientists can request data from the database.
6. Computing data request from scientists: Scientists can carry out computations on the data in the database.

Functions 1, 2, 4, and 6 use RPC communication. These RPC communications have ten different SOAP message contents. Functions 3 and 5 use data communications. The SOAP messages for these communications are based on four XML schemas: `rock.xsd`, `weather.xsd`, `status.xsd`, and `requestType.xsd`.

We inserted 18 faults into MRCS, as summarized in Table 3. Five were faults in predicates, three were faults in computation statements, three were SQL mistakes, three called an incorrect method, two were mistakes related to the use of XML and SOAP, one was an incorrect initial value for a variable and the final fault was incorrect HTML output.

We generated 100 DVP tests, 15 RCP tests, and 27 DCP tests, for a total of 142. These tests found 14 of the 18 faults (78%), as summarized in Table 4. Three computation faults were not found because they were in methods that do not have arguments, but compute their results based on data taken from the database. For example, one fault modified the following statement by replacing the multiplication operator with the plus operator:

```
distance += Math.sqrt (dx[0] * dx[0]+dy[0]*dy[0]);
distance += Math.sqrt (dx[0] + dx[0]+dy[0]*dy[0]);
```

The array `dx[]` was read from a database on the server, and our testing has no way to modify those values so could not detect that fault. Thus our tests do not have any way to "stress" the incorrect computations. This brings up a *controllability* issue: it is difficult to develop tests that depend on values in the database.

The fourth fault was related to saving the changes in SOAP messages before the messages were sent out. The tests also found two naturally occurring faults. One is related to a security problem found by the RCP test cases, and the other was no XML document validation against the XML schema, found by test cases in DVP.

	Number of faults
Predicate faults	5
Computation faults	3
SQL faults	3
Wrong method call	3
XML and SOAP faults	2
Other faults	2

Table 3: The Seeded Faults Distribution

	DVP	RCP	DCP	Total
Number of test cases	100	15	27	142
Faults found (seeded)	14	7	4	14
Faults found (natural)	1	1	0	2

Table 4: Test Cases and Results

6. CONCLUSION AND FUTURE WORK

This paper has presented a new approach for the problem of testing Web services. *Data perturbation* works by modifying values in request messages and analyzing the response messages. Data perturbation has two components: data value perturbation and interaction perturbation. Data value perturbation generates test cases according to the types as specified in the XML schema or DTD. Interaction perturbation classified the communication into two categories, RPC communication and data communication. Separate test strategies are designed for each.

In our preliminary study, the DVP tests found many more faults than the RCP and DCP tests. In fact, DVP found **all** of the faults the others did, plus more. This data is too preliminary to throw out RCP and DCP, but their relative usefulness will definitely be a topic of future experiments as this research project continues.

One part of testing Web services that we have not addressed is related to *controllability*. Specifically, when values come from a database or other data store, it is difficult for the tester to control those values. This issue is common with Web applications as well as other software that relies on disk storage.

The testing so far relies on peer-to-peer interactions (between two components at a time). Multilateral interactions allow scenarios such as a message from Web service component A flowing to component B, which causes another message to flow to C, and the resulting message flows back to B, and then in turn to A. In addition to the messages, this requires testing of aspects of the states of the Web services. As this ongoing research project progresses, we expect to address issues that arise from multilateral interactions.

Another limitation is that we presently have no theory to decide whether the data perturbation operators are complete. More experience with specific types of faults that occur in Web services will help us address this problem.

The approach in this paper also relies strictly on syntactic information about the XML messages. We are currently attempting to bring in behavioral information by considering the XML schema to define communication protocols, and then apply methods from specification-based testing such as model checking. This should allow for more detailed kinds of analysis.

7. REFERENCES

[1] Mikio Aoyama, Sanjiva Weerawarana, Hiroshi Maruyama, Clemens Szyperski, Kevin Sullivan, and Doug Lea. Web services engineering: Promises and challenges. In *Proceedings of the 24th International Conference on Software Engineering*, Orlando, Florida, May 2002.

[2] Marcelo Arenas, Wenfei Fan, and Leonid Libkin. What's hard about XML Schema constraints? In *The International Conference on Database and Expert Systems Applications (DEXA)*, Springer, Heidelberg, 2002.

[3] B. Beizer. *Software Testing Techniques*. Van Nostrand Reinhold, Inc, New York NY, 2nd edition, 1990. ISBN 0-442-20672-0.

[4] Jason Bloomberg. Report: Testing Web services. White paper of report for Parasoft SOAPTest, 2002. http://www.parasoft.com/jsp/templates/misc/soap/web_services_excerpt.pdf (accessed November 2003).

[5] Jason Bloomberg. Testing Web services today and tomorrow. The Rational Edge E-zine for the Rational Community, 2002. http://www.therationaledge.com/content/oct_02/PDF/WebTesting_TheRationalEdge_Oct02.pdf (accessed November 2003).

[6] Boris Chidlovskii. Using regular tree automata as XML Schemas. In *IEEE Advances in Digital Libraries 2000 (ADL 2000)*, Washington, D.C., May 2000.

[7] Joe Clabby. *Web services explained: Solutions and applications for the real world*. Pearson Education Inc., 2003.

[8] Neil Davidson. Web services testing. The Red-gate software technical papers, 2002. http://www.red-gate.com/dotnet/more/web_services_testing.htm (accessed November 2003).

[9] R. A. DeMillo and A. J. Offutt. Constraint-based automatic test data generation. *IEEE Transactions on Software Engineering*, 17(9):900–910, September 1991.

[10] IBM. Web services: Taking e-business to the next level. IBM white paper, 2000. http://www-900.ibm.com/developerWorks/cn/wsdd/download/pdf/e-businessj.pdf (accessed November 2003).

[11] Suet Chun Lee and Jeff Offutt. Generating test cases for XML-based Web component interactions using mutation analysis. In *Proceedings of the 12th International Symposium on Software Reliability Engineering*, pages 200–209, Hong Kong China, November 2001. IEEE Computer Society Press.

[12] Makoto Murata. Hedge automata: A formal model for XML schema. Web page, 2000. http://citeseer.nj.nec.com/article/murata99hedge.html (accessed November 2003).

[13] Jeff Offutt. Quality attributes of Web software applications. *IEEE Software: Special Issue on Software Engineering of Internet Software*, 19(2):25–32, March/April 2002.

[14] Florian Reuter and Norbert Luttenberger. Cardinality constraint automata: A core technology for efficient XML schema-aware parsers. In *The Twelfth International World Wide Web Conference (WWW2003)*, Budapest, Hungary, May 20-24 2003.

[15] Kevin Spett. SQL injection: Are your Web applications vulnerable? White paper in SPI Dynamics, Inc., 2002. http://www.spidynamics.com/whitepapers/WhitepaperSQLInjection.pdf (accessed November 2003).

[16] W. T. Tsai, Ray Paul, Weiwei Song, and Zhibin Cao. Coyote: An XML-based framework for Web services testing. In *7th IEEE International Symposium on High Assurance Systems Engineering (HASE'02)*, Tokyo, Japan, October 2002.

[17] W3C. Namespaces in XML, 1999. http://www.w3.org/TR/REC-xml-names/ (accessed November 2003).

[18] W3C. Extensible markup language (XML) 1.0 (second edition) – W3C recommendation, October 2000. http://www.w3.org/XML/.

[19] W3C. Datatypes in XML. W3C Recommendation, 2001. http://www.w3.org/TR/xmlschema-2/ (accessed April 2004).

[20] W3C. XML information set, 2001. http://www.w3.org/TR/xml-infoset/ (accessed November 2003).

[21] W3C. XML schema – recommendation, May 2001. http://www.w3c.org/tr/.

[22] W3C. XML path language (XPath) 2.0, 2003. http://www.w3.org/TR/xpath20/ (accessed November 2003).

[23] Joseph Williams. The Web services debate: J2EE vs. .NET. *E-services: A cornucopia of digital offerings ushers in the next Net-based evolution*, 46(6):58–63, June 2003.