

Rt  
437  
c.1

## MUTATION TESTING OF WEB SERVICES

By

**Reda M. Siblini**

A thesis submitted in partial fulfillment of the  
requirements for the degree of Master of Science  
in Computer Science

Thesis Advisor: Dr. Nashat Mansour


Department of Computer Science  
LEBANESE AMERICAN UNIVERSITY  
2004

# MUTATION TESTING OF WEB SERVICES

Reda M. Siblini


## THESIS

Submitted in partial fulfillment of the requirement of the degree of Master of  
Science in Computer Science at the Lebanese American University  
Beirut, Lebanon  
Feb. 2004




---

**Dr. Nash'at Mansour (Advisor)**  
Associate Professor of Computer Science  
Lebanese American University



---

**Dr. Ramzi Haraty**  
Associate Professor of Computer Science  
Lebanese American University



---

**Dr. Faisal Abu Khzam**  
Assistant Professor of Computer Science  
Lebanese American University

I grant to the LEBANESE AMERICAN UNIVERSITY the right to use this work, irrespective of any copyright, for the University's own purpose without cost to the University or to its students, agents and employees. I further agree that the University may reproduce and provide single copies of the work, in any format other than in or from microforms, to the public for the cost of reproduction.

---

# MUTATION TESTING OF WEBSERVICES

## ABSTRACT

By

Reda M. Siblini

*Web Service is a new promising software development technology. It provides application-to-application interaction. Built on top of existing Web protocols and based on open XML standards, it is divided into communication protocols, service descriptions, and service discovery. Communication among Web Services is done using Simple Object Access Protocol (SOAP), Web Services are described using Web Services Description Language (WSDL), and the Universal Description, Discovery, and Integration directory (UDDI) provide a registry of Web Services descriptions. Testing Web Services is essential for both the Web Service provider and the Web Service user. This thesis proposes an approach for testing Web Services using mutation analysis. The approach consists of applying mutation operators to the WSDL document to generate mutated Web Service interfaces that will be used to test the Web Service. The proposed method defines mutant operators specific to the WSDL language. The experimental results show the usefulness of the method by applying it to Web Services written in Microsoft VB.NET.*

### **Acknowledgments**

I would first like to express my grateful acknowledgement to my advisor Dr. Nashat Mansour for his patience, support, and friendship.

Dr. Mansour first indicated the use of Mutation testing to test web applications. He also motivated me to study software testing, and provided many helpful ideas in this research as well as in the area of software engineering. I would like also like to show gratitude to Dr. Ramzi Haraty and Dr. Faisal AbuKhzam for their help and assistance.

## TABLE OF CONTENTS

<i>List of Figures</i>	<i>vii</i>
<i>List of Tables</i>	<i>viii</i>
<i>Introduction</i>	<i>3</i>
<i>Background</i>	<i>5</i>
2.1    Web Services:	5
2.2    Web Services Description Language:	8
2.3    Mutation Analysis	12
<i>Proposed Approach</i>	<i>15</i>
3.1    Applying Interface Mutation to Web Services	15
3.2    WSDL special language features	17
3.2.1    Types	18
3.2.2    Messages	23
3.2.3    Port Types	24
3.2.4    Bindings	27
3.2.5    Ports	27
3.2.6    Services	28
<i>Experimental Results</i>	<i>31</i>
4.1    Detailed Example	31
4.2    More experimental results	43
<i>Conclusion</i>	<i>63</i>
<i>References</i>	<i>65</i>
<i>Glossary</i>	<i>66</i>

## List of Figures

<i>Figure</i>	<i>Title</i>	<i>Page</i>
Figure1:	Web Service.....	5
Figure2:	Mutation analysis.....	14
Figure3:	Web Services testing.....	30
Figure4:	Web Service client interface .....	35
Figure5:	Web Service client interface2 .....	36
Figure6:	Web Service client interface3 .....	36
Figure8:	ASP.NET mutated client code.....	40
Figure9:	Web Service client interface4 .....	41
Figure10:	Web Service client interface5 .....	41
Figure11:	VB.NET Web Service code.....	42
Figure12:	ASP.NET Web Service testing application.....	44
Figure13:	Tester Web Service.....	64

## List of Tables

<i>Table</i>	<i>Title</i>	<i>Page</i>
<i>Table 1.</i>	<i>WSDL mutation operators.</i>	29



## *Chapter 1*

### **Introduction**

Web Services (W3C, 2003) are emerging to provide a framework for application-to-application interaction, built on top of existing Web protocols and based on XML standards. Nowadays, Web applications are integrating Web Services from a variety of resources. One of the major benefits is Web Services' ease of integration. You will easily integrate your software with other pieces of software. You can run Web Services on all kinds of machines, from the desktop to the mainframe, either within your enterprise or at external sites. This ease of integration will enable tighter business relationships and more efficient business processes.

There has been a lot of research dedicated to testing software components; however, there has not been much research on testing Web Services on web platforms. To assure the quality of web applications, testing techniques are needed to evaluate the integration of Web Services.

Several Web Services are written in different programming languages, usually distributed over a network or on the Internet that may not have the same runtime environment. Since the client communicates with the Web Service through an interface presented by the Web Service, there may be errors in the way the Web Service is used. Interface errors could occur at the level that Web Services have been defined.

Since Web Service source code is usually not available, we will deal with them as black box software, contrary to the client application code that is going to use the Web Service and whose source code is available, it will be dealt with as white box software. Interface errors will result from interactions between black box and white box software,

considering the high cost of testing; adding more tests to the client application will be inefficient.

So, a desirable and effective testing technique will be needed to test just the interface of the Web Service and its interaction with the web application. The technique proposed in this thesis is based on mutation analysis (Offutt and Untch, 2000). While mutation analysis makes mutants by injecting faults to every statement of a program, the proposed technique will mutate just the interface of the Web Service, which means just the parameters of calling Web Services methods, and the returned variable. The proposed method will mutate the Web Service Description Language document (WSDL) (W3C, 2001) by applying WSDL specific mutant operators.

The desired objective is to minimize the test case domain by applying mutation at the same time revealing Web Service errors. To the best of our knowledge the only research available on Web Service WSDL testing is concerning WSDL document validation.

A background of Web Service, WSDL, and mutation test is discussed in Chapter 2. Chapter 3 provides an accurate description and definition of the mutation techniques alongside definition of WSDL specific mutant operators. An analysis of the proposed method is given in Chapter 4 while using a Microsoft ASP.NET web application as a client interface to a Microsoft VB.NET Web Service. Conclusions and future work are featured chapter 5.

## Chapter 2

### Background

This chapter presents an overview of the concepts and technologies used throughout the thesis. Section 2.1 gives an overview of Web Services, section 2.2 describes WSDL, and section 2.3 presents mutation analysis.

#### 2.1 Web Services:

A Web Service is a software system identified by a URI, whose public interfaces and bindings are defined and described using XML (W3C, 1998). Its definition can be discovered by other software systems. These systems may then interact with the Web Service in a manner prescribed by its definition, using XML based messages conveyed by Internet protocols as indicated in Figure1 (W3C, 2003).

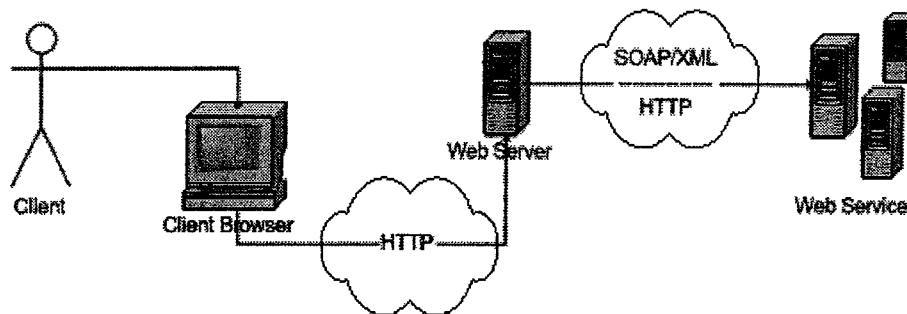


FIGURE1: WEB SERVICE

Web Services are enterprise applications that exchange data, share tasks, and automate processes over the Internet. As a new class of Internet-native applications, Web Service promises to increase interoperability, and lower the costs of software integration and data sharing with partners. As they are based on simple and non-proprietary

standards, Web Services are designed to make it possible for computer programs to communicate directly with one another and exchange data regardless of location, operating systems, or languages. Web Services allow pieces of software written in different languages, or running on different operating systems, running in different parts of an organization, or in different organizations, to talk to one another cheaply and easily using universal and non-proprietary data standards so that integration between new pieces of software and legacy systems will be simple.

The Web Service framework is divided into three areas: exchanging messages, service descriptions, and service discovery.

XML Web Services expose useful functionality to Web users through a standard Web protocol. In most cases, the protocol used is SOAP.

XML Web Services provide a way to describe their interfaces in enough detail to allow a user to build a client application to talk to them. This description is usually provided in an XML document called a WSDL document.

XML Web Services are registered so that potential users can find them easily. This is done with UDDI.

We have defined an XML Web Service as a software service exposed on the Web through SOAP, described with a WSDL file and registered in UDDI. Exposing existing applications, as XML Web Services, will allow users to build new, more powerful applications that use XML Web Services as building blocks. SOAP is the communications protocol for XML Web Services. SOAP is a specification that defines the XML format for messages. There are other parts of the SOAP specification that describe how to represent program data as XML and how to use SOAP to do Remote Procedure Calls. These optional parts of the specification are used to implement RPC-style applications where a SOAP message containing a callable function, and the

parameters to pass to the function, is sent from the client, and the server returns a message with the results of the executed function. Most current implementations of SOAP support RPC applications because programmers who are used to do COM or CORBA applications understand the RPC style. SOAP also supports document style applications where the SOAP message is just a wrapper around an XML document. Document-style SOAP applications are very flexible and many new XML Web Services take advantage of this flexibility to build services that would be difficult to implement using RPC.

There is a common misconception that SOAP requires HTTP. Some implementations support MSMQ, MQ Series, SMTP, or TCP/IP transports, but almost all current XML Web Services use HTTP because it is all-pervading. Since HTTP is a core protocol of the Web, most organizations have a network infrastructure that supports HTTP and people who understand how to manage it already. The security, monitoring, and load-balancing infrastructure for HTTP are readily available today.

WSDL stands for Web Services Description Language. For our purposes, we can say that a WSDL file is an XML document that describes a set of SOAP messages and how the messages are exchanged. In other words, WSDL is to SOAP what IDL is to CORBA or COM. Since WSDL is XML, it is readable and editable but in most cases, it is generated and consumed by software. WSDL will be described further in the next section.

Universal Discovery Description and Integration is the yellow pages of Web Services. As with traditional yellow pages, you can search for a company that offers the services you need, read about the service offered and contact someone for more information. You can, of course, offer a Web Service without registering it in UDDI, just as you can open a business in your basement and rely on word-of-mouth advertising but if you want to reach a significant market, you need UDDI so your customers can find you.

A UDDI directory entry is an XML file that describes a business and the services it offers.

## 2.2 Web Services Description Language:

WSDL is an XML-based language to define Web Services and how to access them. WSDL stands for Web Services Description Language.

WSDL is a document written in XML. The document describes a Web Service. It specifies the location of the service and the operations, or methods, the service exposes.

### The WSDL Document Structure

A WSDL document defines a Web Service using four major elements:

#### Element Defines

<portType> the operations performed by the Web Service

<message> the messages used by the Web Service

<types> the data types used by the Web Service

<binding> the communication protocols used by the Web Service

The main structure of a WSDL document looks like this:

```
<definitions>
  <types>
    definition of types.....
  </types>
  <message>
    definition of a message....
  </message>
```

```
<portType>  
    definition of a port.....  
</portType>  
<binding>  
    definition of a binding...  
</binding>  
</definitions>
```

A WSDL document can also contain other elements, like extension elements and a service element that makes it possible to group together the definitions of several Web Services in one single WSDL document.

### WSDL Ports

The **<portType>** element is the most important WSDL element. It defines a Web Service, the operations that can be performed, and the messages that are involved. The **<portType>** element can be compared to a function library (or a module, or a class) in a traditional programming language.

### WSDL Messages

The **<message>** element defines the data elements of an operation. Each message can consist of one or more parts. The parts can be compared to the parameters of a function call in a traditional programming language.

### WSDL Types

The **<types>** element defines the data type that is used by the Web Service. For maximum platform neutrality, WSDL uses XML Schema syntax to define data types.

## WSDL Bindings

The **<binding>** element defines the message format and protocol details for each port.

## WSDL Example

This is a simplified fraction of a WSDL document:

```
<message name="CreditCardRequest">
  <part name="term" type="xs:string"/>
</message>
<message name="CreditCardResponse">
  <part name="value" type="xs:string"/>
</message>
<portType name="CreditCardLibrary">
  <operation name="CheckCreditCard">
    <input message="CreditCardRequest"/>
    <output message="CreditCardResponse"/>
  </operation>
</portType>
```

In this example the **portType** element defines "CreditCardLibrary" as the name of a **port**, and "CheckCreditCard" as the name of an **operation**.

The "CheckCreditCard" operation has an **input message** called "CreditCardRequest" and an **output message** called "CreditCardResponse".

The **message** element defines the **parts** of each message and the associated data types.

Compared to traditional programming, CreditCardLibrary is a function library; "CheckCreditCard" is a function with "CreditCardRequest" as the input parameter and "CreditCardResponse" as the return parameter.



## WSDL Bindings

WSDL bindings define the message format and protocol details for a Web Service.

### Binding to SOAP

A request-response operation example:

```
<message name="CreditCardRequest">
  <part name="term" type="xs:string"/>
</message>
<message name="CreditCardResponse">
  <part name="value" type="xs:string"/>
</message>
<portType name="CreditCardLibrary">
  <operation name="CheckCreditCard">
    <input message="CreditCardRequest"/>
    <output message="CreditCardResponse"/>
  </operation>
</portType>
<binding type="CreditCardLibrary" name="b1">
  <soap:binding style="document"
transport="http://schemas.xmlsoap.org/soap/http" />
    <operation>
      <soap:operation soapAction="http://abc.com/CheckCreditCard"/>
      <input>
        <soap:body use="literal"/>
      </input>
      <output>
        <soap:body use="literal"/>
      </output>
    </operation>
  </binding>
```

The **binding** element has two attributes - the name attribute and the type attribute. The name attribute defines the name of the binding, and the type attribute points to the port for the binding, in this case the "CreditCardLibrary" port. The **soap:binding** element has two attributes - the style attribute and the transport attribute.

The style attribute can be "rpc" or "document". In this case we use document. The transport attribute defines the SOAP protocol to use. In this case we use HTTP.

The **operation** element defines each operation that the port exposes. For each operation the corresponding SOAP action has to be defined. We must also specify how the input and output are encoded. In this case we use "literal".

### 2.3 Mutation Analysis

Mutation analysis is a fault-based testing method that measures the adequacy of a set of externally created test cases (DeMillo and Ofutt, 1991). Mutation analysis induces faults into software by creating many versions of the software, each containing one fault. Test cases are used to execute these faulty programs from the original program. Hence, faulty programs are mutants of the original, and a mutant is killed by distinguishing the output of the mutant from that of the original program.

Mutants either represent likely faults, a mistake the program could have made, or they explicitly require a typical testing heuristic to be satisfied, such as execute every branch or cause all expressions to become zero. Mutants are limited to simple changes on the basis of the coupling effect, which says that complex faults are coupled to simple faults in such a way that a test data set that detects all simple faults in a program will detect most complex faults. The coupling effect has been demonstrated theoretically in 1995 (Offutt and Untch, 2000).

Mutation analysis induces faults into software by producing various versions of the software with one fault each. Mutation operators define these faults and each change or mutation created by a mutation operator is encoded in a mutant program. A typical mutant operator, for example would, replace each operand by every other syntactically legal operand, or modify expressions by replacing operators and inserting new operators, or delete entire statement. Test cases are used to execute these faulty programs with the goal of distinguishing the faulty program from the original one. The entire process is shown in Figure 2.

Upon adding a new test case to the mutation system, the test case is first executed against the original version of the test program to generate the expected output for that use case. The tester has to examine the output; if the output is incorrect, the program should be corrected and the mutation process should be restarted; otherwise, each live mutant is executed against the new test case. The output of the mutant is compared to the expected output. Mutants are killed if their output does not match that of the original, and remain alive if they do.

After all mutants have been executed, the tester is left with two kind of information. The portion of the mutants that die indicates how well the program has been tested. The live mutants that could not be distinguished by test cases from the original program are called equivalent mutants. To assess the adequacy of a test set, the mutation score is computed as follows:

Mutation score = Number of dead mutants / (Number of total mutants – Number of equivalent mutants).

The tester's main goal is to improve the mutation score to 1.00, indicating all mutants have been detected. A test set that kills all mutants is said to be adequate relative to the mutants.

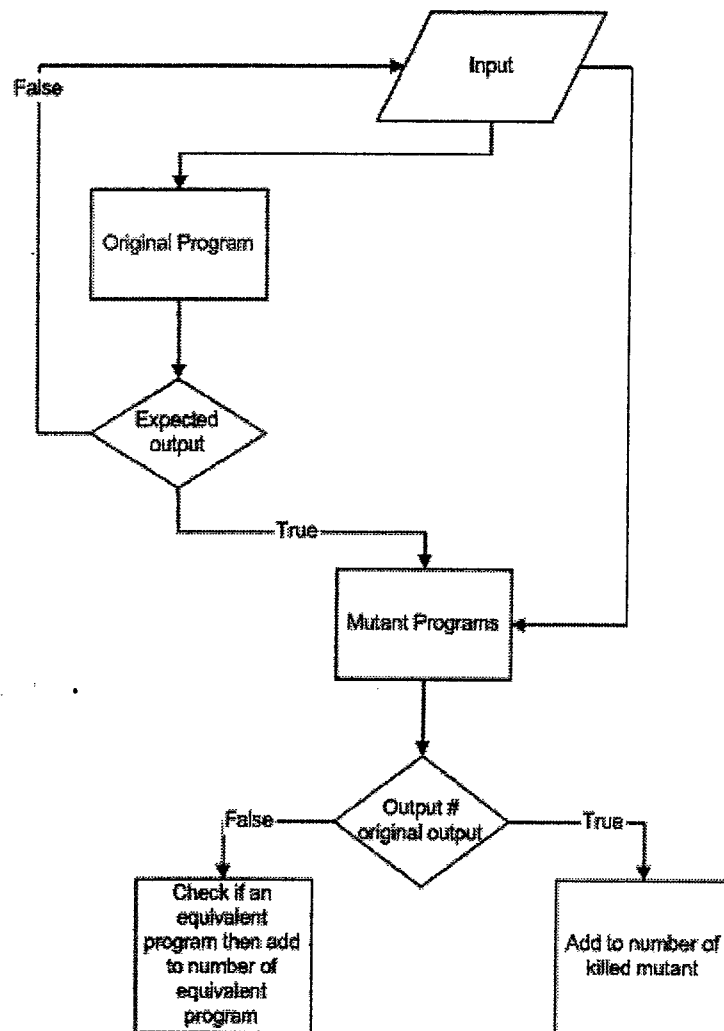


FIGURE2: MUTATION ANALYSIS

## *Chapter 3*

### **Proposed Approach**

The proposed approach for testing Web Services makes use of information available from the description of Web Services' interface. The description provided by the Web Service WSDL document is used to create coverage domains. The testing of Web Services is carried out through the WSDL document. This approach does not rely on the availability of the Web Service 's source code.

#### **3.1 Applying Interface Mutation to Web Services**

In traditional mutation, faults model simple errors that programmers may make in practice. In Web Service interface mutation, faults model the errors that programmers may make while defining, implementing and using interfaces. The method will also reveal Web Services errors, as you will see in the experimental results section. The set of errors that can be applied to the WSDL document constitute the mutation operators.

Mutation operators apply to operation calls defined in the WSDL document. The possible entities to which operators could apply are the operation input messages, operation output messages, and their data types. There could be an infinite number of mutation operators. One could change the value of a parameter in a number of ways (Ghosh and Marthur, 2001). Selection of a set of mutation operators is an important task in performing interface mutation.

Since interface description in WSDL does not contain any code the descriptions themselves have not executed, the mutation operators would have to apply implementations of the interface description. Sets of mutation operator that can be applied to WSDL have been identified.

This proposed method differs from interface mutation in the definition of operators that apply to the WSDL document. While defining the operators, the main purpose was to be able to find errors related not just to the interface, but also to the logic of Web Service programming.

Defining a set of operators is probably the most relevant point for making the use of mutation testing feasible and effective. The set of operators presented here is based on a method of testing and experimenting. It enables the construction of a test set that exercises the possible ways in which functions can interact. The effectiveness and utility of these operators can only be completely assessed by further exploring its characteristics in other studies.

Operators group will be defined next to ease the interpretation of each created mutant operator, and allow the categorizing of mutant operator into well-defined groups.

The main mutation operator group defined is the *Switch* group. The operators defined in this group replace the sequence of an element. Applying this operator to WSDL document will create many mutants WSDL documents. Each mutant WSDL will contain one variation of one replacement of one element with another one in the operation definition. For instance if the operation, has 2 input messages I1, I2 and one output message O. One mutant WSDL will be to replace the sequence of I1, by the sequence of I2. Another mutant WSDL will be replacing I1 by O.

The second mutation operator group defined for WSDL is the *Special* group. The operators defined in this group will modify the value of the element; the value would belong to the same type of the element. The Input submitted to the Web Service will be changed. The change will usually set the parameters to boundary values, to null values (if applicable), or to the next value in the same domain of the input. Each change may reveal a different type of error.

The third operator group is the *Occurrence*. The operators defined in this group will just delete or add an occurrence of an element. The Web Service operation will end up having one of its elements deleted or added.

After defining the three main categories of mutant operators, in the next section we will provide more details on mutant operators which will be generated from the above category and that will be specific to the WSDL language features.

### 3.2 WSDL special language features

Defining the Mutant operator will need the study of the WSDL language feature, and to specifically point to where and how these operators will be applied. This section will define the application of Mutant operator to the WSDL document.

We will explain in details all the sections of a services definition that are related to our need and provided in a WSDL document, and in the process describe the related WSDL specific mutation operators.

WSDL define an XML grammar for describing network services as collections of communication endpoints capable of exchanging messages. The WSDL document uses the following elements in the definition of network services:

**Types**— a container for data type definitions using some type system.

**Message**— an abstract, typed definition of the data being communicated.

**Operation**— an abstract description of an action supported by the service.

**Port Type**—an abstract set of operations supported by one or more endpoints.

**Binding**— a concrete protocol and data format specification for a particular port type.

**Port**— a single endpoint defined as a combination of a binding and a network address.

**Service**— a collection of related endpoints.

It is important to observe that WSDL does not introduce a new type definition language. WSDL recognizes the need for rich type systems for describing message formats, and supports the XML Schemas specification (XSD) as its canonical type system. However, since it is unreasonable to expect a single type system grammar to be used to describe all message formats present and future, WSDL allows using other type definition languages via extensibility. For simplicity extensibility will not be discussed in this paper.

The use of the **import** element allows the separation of the different elements of a service definition into independent documents, which can then be imported as needed.

### *3.2.1 Types*

The **types** element encloses data type definitions that are relevant for the exchanged messages. For maximum interoperability and platform neutrality, WSDL prefers the use of XSD as the canonical type system, and treats it as the intrinsic type system (W3C, 2001). We will cover in this section the XSD type system in order to define specific mutant operators that will be used in the WSDL mutation method.

The mutant operator name will follow the following convention:

Operator group name + WSDL Element Name + ElementType



For example, in the case of string type, the operator name will be SwitchTypesString that define the mutant operator that will switch 2 or more elements of type string in the Types element part of a WSDL document.

As the Type element will follow the XML schema element, we will introduce in this section the different types, element, attributes, and data type that are defined in the XML schema.

### **Complex type:**

In an XML Schema, there is a basic difference between complex types that allow elements in their content and may carry attributes, and simple types that cannot have element content and cannot carry attributes. There is also a major distinction between definitions, which create new types (both simple and complex), and declarations that enable elements and attributes with specific names and types (both simple and complex) to appear in document instances.

The following is a simplified description of a complex type:

```
<xsd:complexType name="" >
  <xsd:sequence>
    <xsd:element name="" type="" />
  </xsd:sequence>
  <xsd:attribute name="" type="" />
</xsd:complexType>
```

New complex types are defined using the complexType element and such definitions typically contain a set of element declarations, element references, and attribute declarations. The declarations are not themselves types, but rather an association between a name and the constraints that govern the appearance of that name in documents governed by the associated schema. Elements are declared using the element element, and attributes are declared using the attribute element.

The **SwitchTypesComplexTypeElement (STCE)** operator is defined as the operator that will switch elements of the same type in the complexType element that could be defined in the Types element of a WSDL document.

The **SwitchTypesComplexTypeAttribute (STCA)** operator is defined as the operator that will switch attributes of the same type in the complexType element that could be defined in the Types element of a WSDL document.

A sample example will be illustrated next to provide a complete understanding of the described operators:

```
<xsd:complexType name="USAddress" >
  <xsd:sequence>
    <xsd:element name="name" type="xsd:string"/>
    <xsd:element name="street" type="xsd:string"/>
    <xsd:element name="city" type="xsd:string"/>
    <xsd:element name="state" type="xsd:string"/>
    <xsd:element name="zip" type="xsd:decimal"/>
  </xsd:sequence>
  <xsd:attribute name="country" type="xsd:NMTOKEN" fixed="US"/>
</xsd:complexType>
```

This ComplexType definition shows the USAddress consisting of 5 elements and one attribute. Applying the SwitchTypesComplexTypeElement once to this schema will produce the following mutated definition:

```
<xsd:complexType name="USAddress" >
  <xsd:sequence>
    <xsd:element name="street" type="xsd:string"/>
    <xsd:element name="name" type="xsd:string"/>
    <xsd:element name="city" type="xsd:string"/>
    <xsd:element name="state" type="xsd:string"/>
    <xsd:element name="zip" type="xsd:decimal"/>
  </xsd:sequence>
  <xsd:attribute name="country" type="xsd:NMTOKEN" fixed="US"/>
</xsd:complexType>
```

</xsd:complexType>

The two elements name, and street are being switched since they follow the SwitchTypesComplexTypeElement operator rule. The complexType USAddress street and name elements are both of the same type, xsd:string, and could be switched.

The application of the SwitchTypesComplexTypeElement to the same definition will produce many other mutant definitions.

The minOccurs and maxOccurs attribute in the definition of an element define the minimum and maximum number of appearances of such an element in an XML document. Attributes may appear once or not at all, but no other number of times, and so the syntax for specifying occurrences of attributes is different than the syntax for elements. In particular, attributes can be declared with a use attribute to indicate whether the attribute is required, optional, or even prohibited.

The **OccurrenceTypesComplexTypeElement (OTCE)** operator is defined as the operator that will add or delete the occurrence of an element in the complexType element that could be defined in the Types element of a WSDL document.

The **OccurrenceTypesComplexTypeAttribute (OTCA)** operator is defined as the operator that will add or delete an optional attribute in the complexType element that could be defined in the Types element of a WSDL document.

The nil attribute is defined as part of the XML Schema namespace for instances. Note that the nil mechanism applies only to element values, and not to attribute values. An element with xsi:nil="true" may not have any element content but it may still carry attributes.

The **SpecialTypesElementNil (STEN)** operator is defined as the operator that will set the "nil" attribute to true in the complexType element that could be defined in the Types element of a WSDL document.

### **Simple Types:**

Element could be of type simple that belongs to the Simple Type built in XML schema. The following is a list of built in data types: string, normalizedString, token, byte, unsignedByte, base64Binary, hexBinary, integer, positiveInteger, negativeInteger, nonNegativeInteger, nonPositiveInteger, int, unsignedInt, long, unsignedLong, short, unsignedShort, decimal, float, double, Boolean, time, dateTime, duration, date, gMonth, gYear, gYearMonth, gDay, gMonthDay, Name, QName, NCName, anyURI, language, ID, IDREF, IDREFS, ENTITY, ENTITIES, NOTATION, NMTOKEN, NMTOKENS

New simple types are defined by deriving them from existing simple types (built-in's and derived). In particular, we can derive a new simple type by restricting an existing simple type. In other words, the legal range of values for the new type is a subset of the existing type's range of values. We use the simpleType element to define and name the new simple type. We use the restriction element to indicate the existing (base) type, and to identify the "facets" that constrain the range of values

The **SwitchTypesSimpleTypeElement (STSE)** operator is defined as the operator that will switch elements of the same data type (built in or derived) in the SimpleType element that could be defined in the Types element of a WSDL document.

The **SwitchTypesSimpleTypeAttribute (STSA)** operator is defined as the operator that will switch elements of the same data type (built in or derived) in the SimpleType element that could be defined in the Types element of a WSDL document.

### 3.2.2 Messages

Messages consist of one or more logical **parts**. Each part is associated with a type from some type system using a message-typing attribute. WSDL defines an element, which refers to an XSD element, and type, which refers to XSD simple or complex type, as message-typing attributes for use with XSD. In addition to the message **name** attribute that provides a unique name among all messages defined within the enclosing WSDL document, and the part **name** attribute that should provides a unique name among all the parts of the enclosing message.

Example message part in a WSDL document:

```
<message name=" ">
  <part name=" " element=" " type=" " />
</message>
```

Multiple part elements are used if the message has multiple logical units.

```
<message name=" ">
  <part name=" " element=" " />
  <part name=" " element=" " />
</message>
```

If the message contents are sufficiently complex, then an alternative syntax may be used to specify the composite structure of the message using the type system directly.

```
<types>
  <schema >
    <complexType name="Composite">
      <choice>
        <element name=" " type=" " />
        <element name=" " type=" " />
      </choice>
    </complexType>
  </schema>
```

```

</types>
<message name=" ">
  <part name=" " type="Composite"/>
</message>

```

The **SwitchMessagesPart(SMP)** operator is defined as the operator that will switch parts of the same element type in the Message element that is defined in a WSDL document.

### 3.2.3 Port Types

A port type is a named set of abstract operations and the abstract messages involved.

```

<wsdl:portType name=" ">
  <wsdl:operation name=" " />
</wsdl:portType>

```

The port type **name** attribute provides a unique name among all port types defined within the enclosing WSDL document, and an operation is named via the **name** attribute.

WSDL has four transmission primitives, referred to as **operations** that an endpoint can support:

**One-way.** The endpoint receives a message.

```

<wsdl:portType name=" ">
  <wsdl:operation name=" " >
    <wsdl:input name=" " message=" "/>
  </wsdl:operation>
</wsdl:portType>

```

The **input** element specifies the abstract message format for the one-way operation.

**Request-response.** The endpoint sends a message, and receives a correlated message.

```
<wsdl:portType name=" ">
  <wsdl:operation name=" " parameterOrder=" " >
    <wsdl:input name=" " message=" "/>
    <wsdl:output name=" " message=" "/>
    <wsdl:fault name=" " message=" "/>
  </wsdl:operation>
</wsdl:portType >
```

**Solicit-response.** The endpoint receives a message, and sends a correlated message.

```
<wsdl:portType name=" ">
  <wsdl:operation name=" " parameterOrder=" " >
    <wsdl:output name=" " message=" "/>
    <wsdl:input name=" " message=" "/>
    <wsdl:fault name=" " message=" "/>
  </wsdl:operation>
</wsdl:portType >
```

**Notification.** The endpoint sends a message.

```
<wsdl:portType name=" ">
  <wsdl:operation name=" ">
    <wsdl:output name=" " message=" "/>
  </wsdl:operation>
</wsdl:portType >
```

The **output** element specifies the abstract message format for the notification operation.

The **name** attribute of the input and output elements provides a unique name among all input and output elements within the enclosing port type. If the name attribute

is not specified, it is defaulted to a unique name, which is usually the name of the operation appended to the primitive transmission type of the message.

The **SwitchPortTypeMessage(SPM)** operator is defined as the operator that will switch messages of the same type in the operation element that is defined in a PortType element, which has as operation of type request-response or solicit-response, of a WSDL document.

Operations do not specify whether they are to be used with RPC-like bindings or not. However, when using an operation with an RPC-binding, it is useful to be able to capture the original RPC function signature. For this reason, a request-response or solicit-response operation MAY specify a list of parameter names via the **parameterOrder** attribute (of type nmtokens). The value of the attribute is a list of message part names separated by a single space. The value of the parameterOrder attribute MUST follow the following rules:

The part name order reflects the order of the parameters in the RPC signature. The **return** value part is not present in the list. If a part name appears in both the input and output message, it is an **in/out** parameter. If a part name appears in only the input message, it is an **in** parameter. If a part name appears in only the output message, it is an **out** parameter. ParameterOrder is not required to be present, even if the operation is to be used with an RPC-like binding.

Other mutant operators in this section could be defined to handle for example the parameterOrder attribute this operator may be defined in subsequent papers.



### 3.2.4 Bindings

A binding defines message format and protocol details for operations and messages defined by a particular portType. There may be any number of bindings for a given portType. The grammar for a binding is as follows:

```
<wsdl:definitions .... >
  <wsdl:binding name="nmtoken" type="qname">
    <wsdl:operation name="nmtoken">
      <wsdl:input name="nmtoken" >
      </wsdl:input>
      <wsdl:output name="nmtoken" >
      </wsdl:output>
      <wsdl:fault name="nmtoken">
      </wsdl:fault>
    </wsdl:operation>
  </wsdl:binding>
</wsdl:definitions>
```

Binding extensibility elements are used to specify the concrete grammar. A binding must specify exactly one protocol, and must not specify address information.

### 3.2.5 Ports

A port defines an individual endpoint by specifying a single address for a binding.

```
<wsdl:definitions .... >
  <wsdl:service .... > *
    <wsdl:port name="nmtoken" binding="qname">
    </wsdl:port>
  </wsdl:service>
</wsdl:definitions>
```

Binding extensibility elements are used to specify the address information for the port. A port must not specify more than one address, and must not specify any binding information other than address information.

### **3.2.6 Services**

A service groups a set of related ports together:

```
<wsdl:definitions ....>
  <wsdl:service name="nmtoken"> *
    <wsdl:port .... />
  </wsdl:service>
</wsdl:definitions>
```

We covered all the elements in a WSDL document. Binding types; however, will not be addressed in this paper, as our main concern is to apply mutant operators on the WSDL document elements independently on the binding type. Table 1 summarizes the proposed mutant operators.

**TABLE 1. WSDL MUTATION OPERATORS.**

<i>Mutation operator</i>	<b>Group</b>	<b>WSDL element</b>	<b>Description</b>
<b>STCE (SwitchTypesComplexTypeElement)</b>	Switch	Types	Switch elements of the same type in the complexType element
<b>STCA (SwitchTypesComplexTypeAttribute)</b>	Switch	Types	Switch attributes of the same type in the complexType element
<b>OTCE (OccurrenceTypesComplexTypeElement)</b>	Occurrence	Types	Add or delete occurrence of an element in the complexType
<b>OTCA (OccurrenceTypesComplexTypeAttribute)</b>	Occurrence	Types	Add or delete an optional attribute in the complexType element
<b>STEN (SpecialTypesElementNil)</b>	Special	Types	Set the nil attribute to true in the complexType element
<b>STSE (SwitchTypesSimpleTypeElement)</b>	Switch	Types	Switch elements of the same data type (built in or derived) in the SimpleType element
<b>STSA (SwitchTypesSimpleTypeAttribute)</b>	Switch	Types	Switch elements of the same data type (built in or derived) in the SimpleType element
<b>SMP (SwitchMessagesPart)</b>	Switch	Messages	Switch parts of the same element type in the Message element
<b>SPM (SwitchPortTypeMessage)</b>	Switch	PortType	Switch messages of same type in the operation element that is defined in a PortType element, which has as operation of type request-response or solicit-response

Once the mutants are generated, the next steps are the same as in traditional mutation testing: to execute the mutants, to evaluate test set adequacy, and to decide on mutant equivalence.

Now after the mutants WSDL are created, the tester, usually called oracle, will provide input test cases to the client. These test cases will be selected putting in mind that their main purpose is to kill the mutation. This tester will verify that the output from the original code is the expected one. The expected output would be the output that the Web Service promised to provide when giving the inserted input. If the output is not the expected one then the Web Service contains a bug. Otherwise, the input is then provided to the one of the mutant. If the mutant output is different than the original expected output, then the mutant is killed; we will not use it back in the testing method.

If a fault exists in the program, it is likely that there exists a mutant that can only be killed by a test that also finds the fault. Figure 3 describes the entire Web Service testing process.

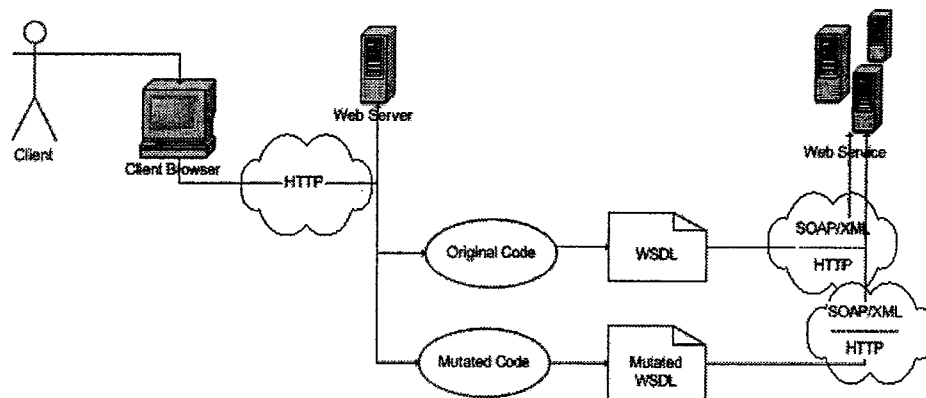


FIGURE3: WEB SERVICES TESTING

## *Chapter 4*

### **Experimental Results**

In this section we test the method described in the previous chapter. We will first define a detailed example of executing this method on a VB.NET (Microsoft, 2003) Web Service. Then we will continue with more experimental results on applying the method on online services.

#### **4.1 Detailed Example**

The experiment is performed on Microsoft .Net platform for the creation of the client and the Web Service. The Web Service used is a sample Web Service used as a credit card checking service. The service is built in VB.NET and will take as input a credit card number and credit card expiry date, and will return a message describing if the information is valid or not. The Client is a web application written in ASP.NET (Microsoft, 2002) that will call the Web Service credit card check function and display the relevant response.

We will start by providing the WSDL document of the Web Service alongside a brief explanation, and then we will provide sample input while checking for the expected output. The Web Service code and the client code, which are both written VB.NET, are also supplied. We will then apply one mutant operator to the WSDL document, create a new client that will fulfill the mutated WSDL document new requirement, and check if the mutant client request output is the same as the original client request output. In the Web Service that we use, both mutant and original client request output is the same,

which reveals a logical error in the Web Service, and verifies correctness of the WSDL mutation method defined.

The following is the WSDL document of the Web Service:

```
<?xml version="1.0" encoding="utf-8"?>
<definitions xmlns:http="http://schemas.xmlsoap.org/wsdl/http/"
xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
xmlns:s="http://www.w3.org/2001/XMLSchema" xmlns:s0="http://tempuri.org/"
xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:tm="http://microsoft.com/wsdl/mime/textMatching/"
xmlns:mime="http://schemas.xmlsoap.org/wsdl/mime/" targetNamespace="http://tempuri.org/"
xmlns="http://schemas.xmlsoap.org/wsdl/">
  <types>
    <s:schema elementFormDefault="qualified" targetNamespace="http://tempuri.org/">
      <s:element name="CheckCreditCard">
        <s:complexType>
          <s:sequence>
            <s:element minOccurs="0" maxOccurs="1" name="Creditcard" type="s:string" />
            <s:element minOccurs="0" maxOccurs="1" name="ExpiryDate" type="s:string" />
          </s:sequence>
        </s:complexType>
      </s:element>
      <s:element name="CheckCreditCardResponse">
        <s:complexType>
          <s:sequence>
            <s:element minOccurs="0" maxOccurs="1" name="CheckCreditCardResult"
type="s:string" />
          </s:sequence>
        </s:complexType>
      </s:element>
      <s:element name="string" nillable="true" type="s:string" />
    </s:schema>
  </types>
  <message name="CheckCreditCardSoapIn">
    <part name="parameters" element="s0:CheckCreditCard" />
  </message>
  <message name="CheckCreditCardSoapOut">
    <part name="parameters" element="s0:CheckCreditCardResponse" />
  </message>
  <message name="CheckCreditCardHttpGetIn">
    <part name="Creditcard" type="s:string" />
    <part name="ExpiryDate" type="s:string" />
  </message>
  <message name="CheckCreditCardHttpGetOut">
    <part name="Body" element="s0:string" />
  </message>
```

```

<message name="CheckCreditCardHttpPostIn">
  <part name="Creditcard" type="s:string" />
  <part name="ExpiryDate" type="s:string" />
</message>
<message name="CheckCreditCardHttpPostOut">
  <part name="Body" element="s0:string" />
</message>
<portType name="Service1Soap">
  <operation name="CheckCreditCard">
    <input message="s0:CheckCreditCardSoapIn" />
    <output message="s0:CheckCreditCardSoapOut" />
  </operation>
</portType>
<portType name="Service1HttpGet">
  <operation name="CheckCreditCard">
    <input message="s0:CheckCreditCardHttpGetIn" />
    <output message="s0:CheckCreditCardHttpGetOut" />
  </operation>
</portType>
<portType name="Service1HttpPost">
  <operation name="CheckCreditCard">
    <input message="s0:CheckCreditCardHttpPostIn" />
    <output message="s0:CheckCreditCardHttpPostOut" />
  </operation>
</portType>
<binding name="Service1Soap" type="s0:Service1Soap">
  <soap:binding transport="http://schemas.xmlsoap.org/soap/http" style="document" />
  <operation name="CheckCreditCard">
    <soap:operation soapAction="http://tempuri.org/CheckCreditCard" style="document" />
    <input>
      <soap:body use="literal" />
    </input>
    <output>
      <soap:body use="literal" />
    </output>
  </operation>
</binding>
<binding name="Service1HttpGet" type="s0:Service1HttpGet">
  <http:binding verb="GET" />
  <operation name="CheckCreditCard">
    <http:operation location="/CheckCreditCard" />
    <input>
      <http:urlEncoded />
    </input>
    <output>
      <mime:mimeXml part="Body" />
    </output>
  </operation>
</binding>
<binding name="Service1HttpPost" type="s0:Service1HttpPost">

```

```

<http:binding verb="POST" />
<operation name="CheckCreditCard">
  <http:operation location="/CheckCreditCard" />
  <input>
    <mime:content type="application/x-www-form-urlencoded" />
  </input>
  <output>
    <mime:mimeXml part="Body" />
  </output>
</operation>
</binding>
<service name="Service1">
  <port name="Service1Soap" binding="s0:Service1Soap">
    <soap:address location="http://localhost/WebService1/Service1.asmx" />
  </port>
  <port name="Service1HttpGet" binding="s0:Service1HttpGet">
    <http:address location="http://localhost/WebService1/Service1.asmx" />
  </port>
  <port name="Service1HttpPost" binding="s0:Service1HttpPost">
    <http:address location="http://localhost/WebService1/Service1.asmx" />
  </port>
</service>
</definitions>

```

This WSDL file shows that the input elements are: credit card of type string, expiry date as type string:

```

<s:element minOccurs="0" maxOccurs="1" name="Creditcard" type="s:string" />
<s:element minOccurs="0" maxOccurs="1" name="ExpiryDate" type="s:string" />

```

The sequence of the input is credit card first, and expiry date second. The output element is checkcreditcardresult of type string.

```

<s:element minOccurs="0" maxOccurs="1" name="CheckCreditCardResult" type="s:string" />

```

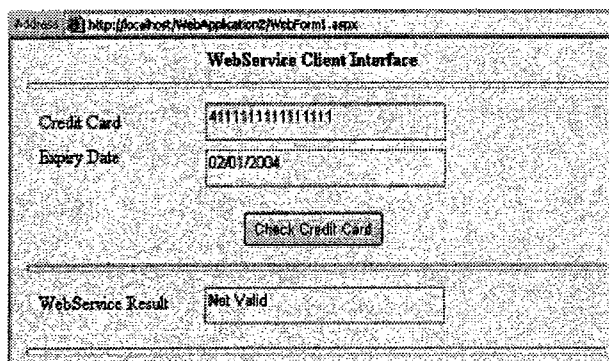
Using a Microsoft .Net tool called WSDL.exe, we create a proxy class for this Web Service. This tool takes the WSDL file as input and creates a proxy class as output. The client will be able to treat the Web Service call like a call to a local class. The client instantiates an instance of the proxy class and calls the checkcreditcard method. The proxy class marshals the parameter list and makes an HTTP request to the Web Service. The Web Service unmarshals the incoming parameters, run the method, and marshals the



output parameter. These are all sent back as a HTTP response to the proxy. The proxy unmarshals the returned parameter and passes back the result to the client. The process is completely transparent to the developer.

We will start to test the Web Service, taking into consideration two valid values. The first is the credit card number 4111111111111111, expiry date 01/01/2004 and the second being card number 4111111111111117, expiry date 01/01/2005.

Sending the input to the Web Service will give us the expected output as promised.



The screenshot shows a web browser window with the address bar displaying 'http://localhost/WebApplication2/WebForm1.aspx'. The page title is 'WebService Client Interface'. The form contains the following elements:

Credit Card	4111111111111111
Expiry Date	02/01/2004
<input type="button" value="Check Credit Card"/>	
WebService Result	Not Valid

FIGURE4: WEB SERVICE CLIENT INTERFACE

Invalid credit card number will result in a "Not Valid" response as promised.

A screenshot of a web browser displaying the 'WebService Client Interface'. The browser's address bar shows 'http://localhost/WebApplication2/WebForm1.aspx'. The interface has a title bar 'WebService Client Interface'. Below the title bar, there are two input fields: 'Credit Card' with the value '4111111111111112' and 'Expiry Date' with the value '01/01/2004'. Below these fields is a button labeled 'Check Credit Card'. At the bottom of the interface, there is a 'WebService Result' field containing the text 'Not Valid'.

FIGURE5: WEB SERVICE CLIENT INTERFACE2

The 2<sup>nd</sup> valid credit card number/ expiry date will give the expected result.

A screenshot of a web browser displaying the 'WebService Client Interface'. The browser's address bar shows 'http://localhost/WebApplication2/WebForm1.aspx'. The interface has a title bar 'WebService Client Interface'. Below the title bar, there are two input fields: 'Credit Card' with the value '4111111111111117' and 'Expiry Date' with the value '01/01/2005'. Below these fields is a button labeled 'Check Credit Card'. At the bottom of the interface, there is a 'WebService Result' field containing the text 'Valid'.

FIGURE6: WEB SERVICE CLIENT INTERFACE3

The original client code, found in Figure 7, shows the calling of the checkcreditcard operation taking as input the creditcardinfo and expirydate operation.

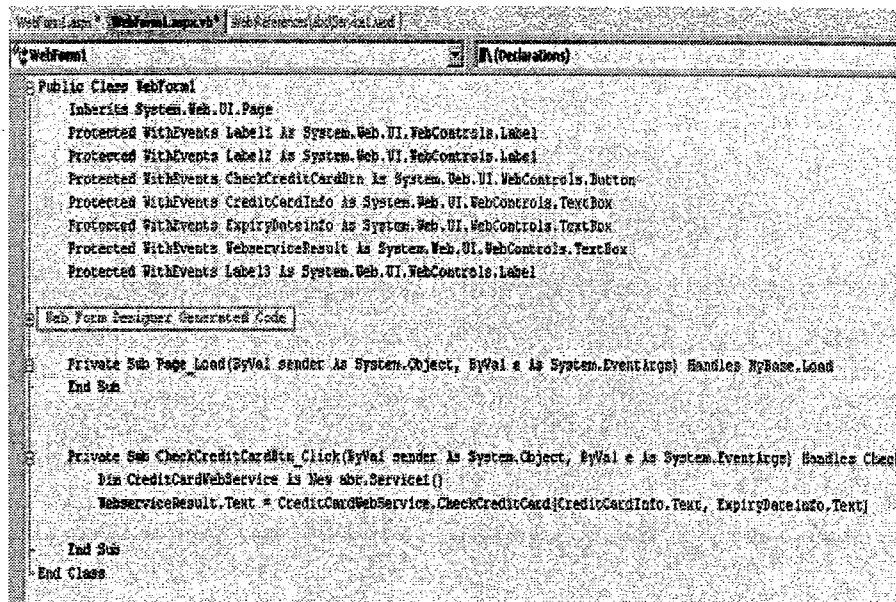


Figure7: ASP.NET original client code

Now we mutate the WSDL file as follows:

```

<?xml version="1.0" encoding="utf-8"?>
<definitions xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:s="http://www.w3.org/2001/XMLSchema" xmlns:s0="http://tempuri.org/"
    xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
    xmlns:tm="http://microsoft.com/wsdl/mime/textMatching/"
    xmlns:mime="http://schemas.xmlsoap.org/wsdl/mime/" targetNamespace="http://tempuri.org/"
    xmlns="http://schemas.xmlsoap.org/wsdl/">
    <types>
        <s:schema elementFormDefault="qualified" targetNamespace="http://tempuri.org/">
            <s:element name="CheckCreditCard">
                <s:complexType>
                    <s:sequence>
                        <s:element minOccurs="0" maxOccurs="1" name="ExpiryDate" type="s:string" />
                        <s:element minOccurs="0" maxOccurs="1" name="Creditcard" type="s:string" />
                    </s:sequence>
                </s:complexType>
            </s:element>
        </s:schema>
    </types>

```

```

        </s:sequence>
      </s:complexType>
    </s:element>
    <s:element name="CheckCreditCardResponse">
      <s:complexType>
        <s:sequence>
          <s:element minOccurs="0" maxOccurs="1" name="CheckCreditCardResult"
type="s:string" />
        </s:sequence>
      </s:complexType>
    </s:element>
    <s:element name="string" nillable="true" type="s:string" />
  </s:schema>
</types>
<message name="CheckCreditCardSoapIn">
  <part name="parameters" element="s0:CheckCreditCard" />
</message>
<message name="CheckCreditCardSoapOut">
  <part name="parameters" element="s0:CheckCreditCardResponse" />
</message>
<message name="CheckCreditCardHttpGetIn">
  <part name="ExpiryDate" type="s:string" />
  <part name="Creditcard" type="s:string" />
</message>
<message name="CheckCreditCardHttpGetOut">
  <part name="Body" element="s0:string" />
</message>
<message name="CheckCreditCardHttpPostIn">
  <part name="ExpiryDate" type="s:string" />
  <part name="Creditcard" type="s:string" />
</message>
<message name="CheckCreditCardHttpPostOut">
  <part name="Body" element="s0:string" />
</message>
<portType name="Service1Soap">
  <operation name="CheckCreditCard">
    <input message="s0:CheckCreditCardSoapIn" />
    <output message="s0:CheckCreditCardSoapOut" />
  </operation>
</portType>
<portType name="Service1HttpGet">
  <operation name="CheckCreditCard">
    <input message="s0:CheckCreditCardHttpGetIn" />
    <output message="s0:CheckCreditCardHttpGetOut" />
  </operation>
</portType>
<portType name="Service1HttpPost">
  <operation name="CheckCreditCard">
    <input message="s0:CheckCreditCardHttpPostIn" />
    <output message="s0:CheckCreditCardHttpPostOut" />
  </operation>
</portType>

```

```

</operation>
</portType>
<binding name="Service1Soap" type="s0:Service1Soap">
  <soap:binding transport="http://schemas.xmlsoap.org/soap/http" style="document" />
  <operation name="CheckCreditCard">
    <soap:operation soapAction="http://tempuri.org/CheckCreditCard" style="document" />
    <input>
      <soap:body use="literal" />
    </input>
    <output>
      <soap:body use="literal" />
    </output>
  </operation>
</binding>
<binding name="Service1HttpGet" type="s0:Service1HttpGet">
  <http:binding verb="GET" />
  <operation name="CheckCreditCard">
    <http:operation location="/CheckCreditCard" />
    <input>
      <http:urlEncoded />
    </input>
    <output>
      <mime:mimeType part="Body" />
    </output>
  </operation>
</binding>
<binding name="Service1HttpPost" type="s0:Service1HttpPost">
  <http:binding verb="POST" />
  <operation name="CheckCreditCard">
    <http:operation location="/CheckCreditCard" />
    <input>
      <mime:contentType="application/x-www-form-urlencoded" />
    </input>
    <output>
      <mime:mimeType part="Body" />
    </output>
  </operation>
</binding>
<service name="Service1">
  <port name="Service1Soap" binding="s0:Service1Soap">
    <soap:address location="http://localhost/WebService1/Service1.asmx" />
  </port>
  <port name="Service1HttpGet" binding="s0:Service1HttpGet">
    <http:address location="http://localhost/WebService1/Service1.asmx" />
  </port>
  <port name="Service1HttpPost" binding="s0:Service1HttpPost">
    <http:address location="http://localhost/WebService1/Service1.asmx" />
  </port>
</service>
</definitions>

```

By applying the **STCE** mutant operator, we switch elements of the same type in the complexType element. The ExpiryDate and CreditCard element in the WSDL document are defined of the built in type string. These two elements will be switched and we will have:

```
<s:element minOccurs="0" maxOccurs="1" name="ExpiryDate" type="s:string" />
<s:element minOccurs="0" maxOccurs="1" name="Creditcard" type="s:string" />
```

Instead of:

```
<s:element minOccurs="0" maxOccurs="1" name="Creditcard" type="s:string" />
<s:element minOccurs="0" maxOccurs="1" name="ExpiryDate" type="s:string" />
```

The proxy class generated by the WSDL.EXE tool contains the checkcreditcard operation that will only accept input parameter in this sequence: expirydate, creditcard.

The client code is then altered to reflect the proxy class changes.

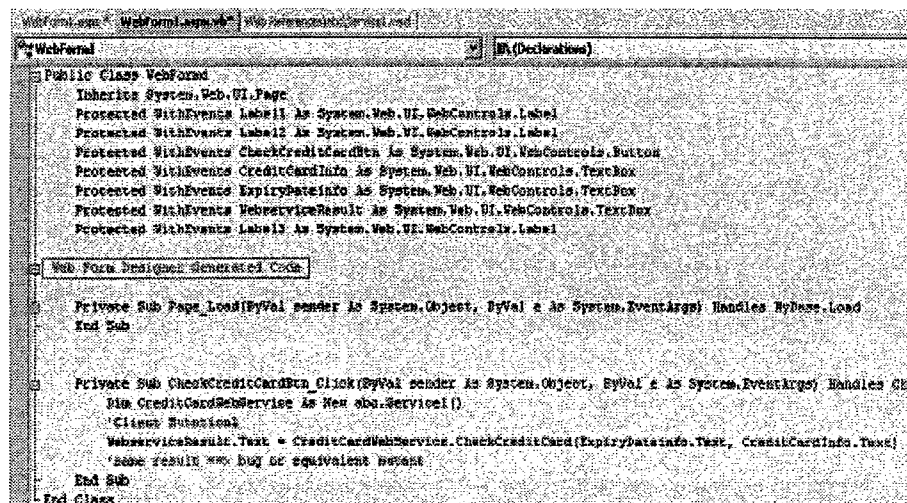


FIGURE8: ASP.NET MUTATED CLIENT CODE

The following input is then supplied to the mutated client code: credit card number 4111111111111111, and expiry date 01/01/2005. The expected output on the original program should not be valid because the expiry date does not match. However, the mutant client output is "Valid". This output is incorrect. In other words, running the same input to the original client surprisingly gave us the same output, that the card is valid. Have uncovered our first bug

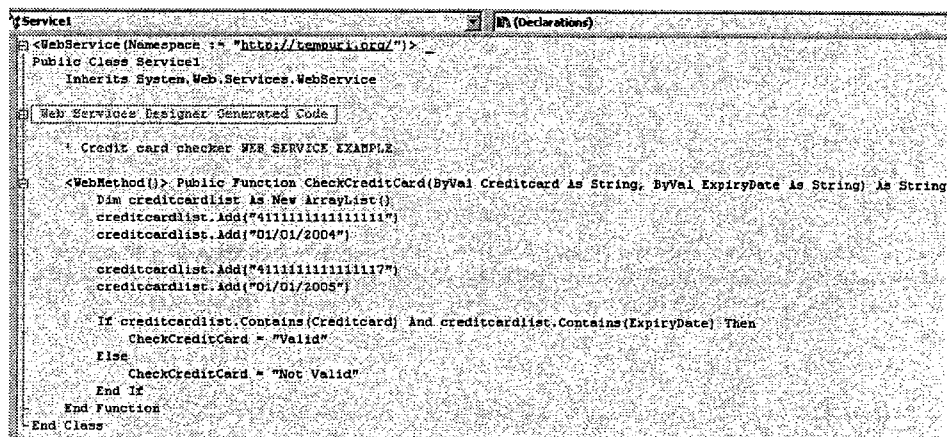
A screenshot of a web browser displaying a web application titled "WebService Client Interface". The browser's address bar shows "http://localhost/WebApplication2/WebForm1.aspx". The interface contains two input fields: "Credit Card" with the value "4111111111111111" and "Expiry Date" with the value "01/01/2005". Below these fields is a button labeled "Check Credit Card". At the bottom, a label "WebService Result" is followed by a text box containing the word "Valid".

FIGURE9: WEB SERVICE CLIENT INTERFACE4

A screenshot of a web browser displaying a web application titled "WebService Client Interface". The browser's address bar shows "http://localhost/WebApplication2/WebForm1.aspx". The interface contains two input fields: "Credit Card" with the value "01/01/2004" and "Expiry Date" with the value "4111111111111111". Below these fields is a button labeled "Check Credit Card". At the bottom, a label "WebService Result" is followed by a text box containing the word "Valid".

FIGURE10: WEB SERVICE CLIENT INTERFACE5

In order to be reliable, Web Services should be extensively validated. This Web Service contains a logical error; the validation of the credit card does not match its expiry date. Figure 11 shows the code of the Web Service. Using the proposed method to test the Web Service, we were able to find errors not only in the interface of the Web Service but even in the Web Service logic itself.



```

' Service1
' (Declarations)

<WebService(Namespace := "http://tempuri.org/")>
Public Class Service1
    Inherits System.Web.Services.WebService

    Web Services Designer Generated Code

    ' Credit card checker WEB SERVICE EXAMPLE 2

    <WebMethod()> Public Function CheckCreditCard(ByVal Creditcard As String, ByVal ExpiryDate As String) As String
        Dim creditcardlist As New ArrayList()
        creditcardlist.Add("4111111111111111")
        creditcardlist.Add("01/01/2004")

        creditcardlist.Add("4111111111111117")
        creditcardlist.Add("01/01/2005")

        If creditcardlist.Contains(Creditcard) And creditcardlist.Contains(ExpiryDate) Then
            CheckCreditCard = "Valid"
        Else
            CheckCreditCard = "Not Valid"
        End If
    End Function
End Class

```

FIGURE11: VB.NET WEB SERVICE CODE



## 4.2 More experimental results

We report further experimental results to investigate the use of proposed method for online Web Services. We selected an online Web Service by querying the UDDI business registry to find companies and production Web Services.

The analysis provided is concerned with the cost of applying interface mutation, which was minimal, and also the effectiveness of revealing errors. It is also important to stress the limited scope of this analysis that has used test Web Services and a small set of test cases. Despite its limited scope, this experiment serves as the starting point to evaluate the new method proposed in this thesis for mutation testing of Web Services.

In order to partially automate the testing process, a web application has been implemented. Figure 12 shows the interface of the web application. The application could be found at: <http://RedaSiblini.europe.webhostingmatrix.net/webtesting>. The application is developed in ASP.NET, and it tries to automate the Web Service testing process using mutation of WSDL. This application does not test all kind of Web Services; its main focus is to test Web Services that accept primitive types as input. In addition, test case requests to the Web Service are accomplished using Http. At this stage, requests made to the Web Service do not support SOAP.

The initial step in the testing procedure is to enter the address of the Web Service WSDL document. When this is performed, the user should click on the Get Methods button in order to retrieve a list of all the methods available in the Web Service. The user is now able to select from the list a method, which he/she wishes to test. Upon doing so, the parameters required for the method will be listed accordingly. The parameter's name and data type will be explicitly shown. Moreover, the user will be able to enter the test value for each parameter in the space provided. This method of adding input can be seen as a disadvantage, for the type of input, which can be inserted, is restricted to primitive types; therefore, objects cannot be inserted. The number of test cases allowed is not limited. In other words, the user is allowed to add as many test cases as desired simply by providing the test case and clicking on the Add Test Case button. In order to place the testing procedure in action, the user has to click on the Start Testing button provided. From this point forward, the process can be considered to be fully automated, that is the user is no longer required to provide any input or perform any action. At this stage, the mutation operators will be invoked upon the WSDL generating in return several mutated versions of the WSDL document. Each test case provided will be used to send an http web request to call the method selected by the user in accordance with the original WSDL document. The retrieved results will be recorded. Next, the same procedure will be repeated for the generated mutated versions of the WSDL. The results recorded encompass the following:

- ? Number of test cases: the total number of test cases which the user has provided
- ? Number of created mutant: the number of the generated mutant versions of the WSDL document
- ? Number of equivalent mutant: the number of mutants, which generate the same results as the original WSDL document.
- ? Number of Killed mutant: the number of mutants, which generate different results from the original WSDL document.
- ? Number of Sent Requests: Total number of HTTP web request sent to the Web Service.
- ? Number of Received Errors: In case the Web Service generated an error, this counter will count the total number of errors received from the Web Service.

In addition, the detailed result of the entire web requests will be generated. This result will include each parameter value, the type of WSDL that is T for Test case and M for Mutated, and the resulted output.

**1. Enter web service WSDL address:**

**WSDL:**

**2. Click Get Methods to get all web service methods**

**Method Name:**

**3. Select the method that you want to test**

**Test Case:**

(a) :

Int64

(b) :

Int64

**5. Click Add Test Case to add your input to your test cases**

**Result:**

**Number of Test Cases:** 1  
**Number of Created Mutant:** 5  
**Number of Equivalent Mutant:** 1  
**Number of Killed Mutant:** 4

**Number of Sent Requests:** 6  
**Number of Received Errors:** 4

**Result:** 80% Adequate

**6. Click Start Testing to start the test**

**Test Case Results for Method:**

**7. Check the test Results**

id	a	b	T/M	result
0	1	2	T	3
1	2	1	M	3
2	1	1	M	The remote server returned an error: (500) Internal Server Error.
3	1	1	M	The remote server returned an error: (500) Internal Server Error.
4	2	2	M	The remote server returned an error: (500) Internal Server Error.
5	2	2	M	The remote server returned an error: (500) Internal Server Error.

FIGURE12: ASP.NET WEB SERVICE TESTING APPLICATION

44

Figure12 presents an example of the complete procedure. The WSDL document describes a simple calculator Web Service. The selected method is the Add method, and this method takes in two parameters, a and b. The two parameters are of the same type, integer. The user in this sample has entered one test case, a=1 and b=2. The test case result shows that 5 mutant has been created. In addition, the result shows that we have 1 equivalent mutant and that's when the element of the method has been switched and we have the same output;  $1+2 = 2+1 = 3$ . The number of killed mutant is 4, the total number of sent requests is 6, and the number of received errors is 4. The test case Results grid shows us that the Web Service in test has generated an "Internal Server Error" when one of the parameters is null. At the end the result shows that the mutation score or result is 80%, which means that the test case is adequate for testing this Web Services. The result grid gives the opportunity for the tester to examine all the test cases and all the results. In case of errors received, the user will be able to know what kinds of inputs could generate errors.

Next we will go over set of Web Services that has been tested using this web application, and we will show the final result in addition to the generated result grid.

First we will continue with the simple calculator Web Service. The Web Service WSDL address is <http://www.xmlwebservices.cc/ws/v1/calc/SimpleCalc.asmx?WSDL>. This Web Service mimics a very simple calculator, with the following methods: add, subtract, multiply, divide.

Function name:

? Add

Parameter in:

? A as integer

? B as integer

Parameter out

? Integer

Number of Test Cases	1
Number of Created Mutant	5
Number of Equivalent mutant	1
Number of Killed mutant	4
Number of Sent Request	6
Number of Received Errors	4
Mutation Score	80%

id	a	b	T/M	result
0	5	6	T	11
1	6	5	M	11
2		5	M	The remote server returned an error: (500) Internal Server Error.
3	5		M	The remote server returned an error: (500) Internal Server Error.
4	6		M	The remote server returned an error: (500) Internal Server Error.
5		6	M	The remote server returned an error: (500) Internal Server Error.

Function name:

? Subtract

Parameter in:

? A as integer

? B as integer

Parameter out

? Integer

Number of Test Cases	1
Number of Created Mutant	5
Number of Equivalent mutant	0
Number of Killed mutant	5
Number of Sent Request	6
Number of Received Errors	4
Mutation Score	100%

id	a	b	T/M	result
0	5	6	T	-1
1	6	5	M	1
2		5	M	The remote server returned an error: (500) Internal Server Error.

id	a	b	T/M	result
3	5		M	The remote server returned an error: (500) Internal Server Error.
4	6		M	The remote server returned an error: (500) Internal Server Error.
5		6	M	The remote server returned an error: (500) Internal Server Error.

Function name:

? Multiply

Parameter in:

? A as integer

? B as integer

Parameter out

? Integer

Number of Test Cases	1
Number of Created Mutant	5
Number of Equivalent mutant	1
Number of Killed mutant	4
Number of Sent Request	6
Number of Received Errors	4
Mutation Score	80%

id	a	b	T/M	result
0	5	6	T	30
1	6	5	M	30
2		5	M	The remote server returned an error: (500) Internal Server Error.
3	5		M	The remote server returned an error: (500) Internal Server Error.
4	6		M	The remote server returned an error: (500) Internal Server Error.
5		6	M	The remote server returned an error: (500) Internal Server Error.

Function name:

? Divide

Parameter in:

? A as integer

? B as integer

Parameter out

? Integer

Number of Test Cases	2
Number of Created Mutant	10
Number of Equivalent mutant	0
Number of Killed mutant	10
Number of Sent Request	12
Number of Received Errors	9
Mutation Score	100%

id	a	b	T/M	result
0	10	5	T	2
1	5	10	M	0
2		10	M	The remote server returned an error: (500) Internal Server Error.
3	10		M	The remote server returned an error: (500) Internal Server Error.
4	5		M	The remote server returned an error: (500) Internal Server Error.
5		5	M	The remote server returned an error: (500) Internal Server Error.
6	10	0	T	The remote server returned an error: (500) Internal Server Error.
7	0	10	M	0
8		10	M	The remote server returned an error: (500) Internal Server Error.
9	10		M	The remote server returned an error: (500) Internal Server Error.

id	a	b	T/M	result
10	0		M	The remote server returned an error: (500) Internal Server Error.
11		0	M	The remote server returned an error: (500) Internal Server Error.

Next we will test a sample credit card checker Web Service. The Web Service WSDL address is <http://cmws.europe.webmatrixhosting.net/CheckCreditCard.asmx?wsdl>. The method checks a credit card, by giving the credit card type, and card number. It returns if the credit card is valid or not.

Function name:

? CheckCardNumber

Parameter in:

? CardType as String

? CardNumber as String

Parameter out

? String

Number of Test Cases	1
Number of Created Mutant	5
Number of Equivalent mutant	1
Number of Killed mutant	4
Number of Sent Request	6
Number of Received Errors	4
Mutation Score	80%

id	CardType	cardNumber	T/M	Result
0	VISA	4111111111111111	T	This is no valid Credit Card.
1	4111111111111111	VISA	M	The remote server returned an error: (500) Internal

id	CardType	cardNumber	T/M	Result
				Server Error.
2	VISA	VISA	M	The remote server returned an error: (500) Internal Server Error.
3	VISA		M	The remote server returned an error: (500) Internal Server Error.
4	4111111111111111		M	The remote server returned an error: (500) Internal Server Error.
5		4111111111111111	M	This is no valid Credit Card.

Next we will test a currency converter Web Service. The Web Service WSDL address is <http://www.websvcx.net/CurrencyConverter.asmx?wsdl>. The method returns conversion rate from one currency to another currency.

Function name:

? ConversionRate

Parameter in:

? FromCurrency as String

? ToCurrency as String

Parameter out

? Decimal



Number of Test Cases	1
Number of Created Mutant	5
Number of Equivalent mutant	0
Number of Killed mutant	5
Number of Sent Request	6
Number of Received Errors	4
Mutation Score	100%

id	FromCurrency	ToCurrency	T/M	result
0	USD	LBP	T	1514
1	LBP	USD	M	0.0007
2		USD	M	The remote server returned an error: (500) Internal Server Error.
3	USD		M	The remote server returned an error: (500) Internal Server Error.
4	LBP		M	The remote server returned an error: (500) Internal Server Error.
5		LBP	M	The remote server returned an error: (500) Internal Server Error.

Next we will test a Web Service that gives back location information. The Web Service WSDL address is <http://teachatechie.com/GJTTWebServices/ZipCode.asmx?wsdl>.

Function name:

? GetNearbyZipCodes: Returns a DataSet containing maximum of 250 zip codes and radius milage within a given radius of a zip code

Parameter in:

? ZipCode as String

? RadiusMiles as Integer

Parameter out

? String

Number of Test Cases	1
----------------------	---

Number of Created Mutant	5
Number of Equivalent mutant	0
Number of Killed mutant	5
Number of Sent Request	6
Number of Received Errors	2
Mutation Score	100%

id	ZipCode	RadiusMiles	T/M	result
0	22041	10	T	20001WASHINGTONDISTRICT OF....
1	10	22041	M	
2		22041	M	
3	22041		M	The remote server returned an error: (500) Internal Server Error.
4	10		M	The remote server returned an error: (500) Internal Server Error.
5		10	M	

Function name:

?      GetLocation: Returns a DataSet with all locations that have a given zip code

Parameter in:

?      ZipCode as String

Parameter out

?      String

Number of Test Cases	1
Number of Created Mutant	1
Number of Equivalent mutant	0
Number of Killed mutant	1
Number of Sent Request	2
Number of Received Errors	1
Mutation Score	100%

id	ZipCode	T/M	result
0	22041	T	22041BAILEYS CROSSROADSFAIRFAXVA22041FALLS CHURCHFAIRFAXVA
1		M	Non-negative number required. Parameter name: byteCount

Function name:

? GetNearbyZipCodesWhereClause: Returns a string Where clause  
containing maximum of 250 zip codes within a given radius of a zip code

Parameter in:

? ZipCode as String  
? RadiusMiles as Integer  
? WhereFieldName as String

Parameter out

? String

Number of Test Cases	1
Number of Created Mutant	23
Number of Equivalent mutant	0
Number of Killed mutant	23
Number of Sent Request	24
Number of Received Errors	12
Mutation Score	100%

id	ZipCode	RadiusMiles	WhereFieldName	T/M	result
0	22041	10	c	T	c='22041' or c='22040' or...
1	22041	c	10	M	The remote server returned an error: (500) Internal Server Error.
2	10	22041	c	M	
3	10	c	22041	M	The remote server returned an error: (500) Internal

id	ZipCode	RadiusMiles	WhereFieldName	T/M	result
					Server Error.
4	c	22041	10	M	
5	c	10	22041	M	
6		10	22041	M	
7		22041	10	M	
8	10		22041	M	The remote server returned an error: (500) Internal Server Error.
9	10	22041		M	
10	22041		10	M	The remote server returned an error: (500) Internal Server Error.
11	22041	10		M	
12	c		22041	M	The remote server returned an error: (500) Internal Server Error.
13	c	22041		M	
14		c	22041	M	The remote server returned an error: (500) Internal Server Error.
15		22041	c	M	
16	22041	c		M	The remote server returned an error: (500) Internal Server Error.
17	22041		c	M	The remote server returned an error: (500) Internal Server Error.

id	ZipCode	RadiusMiles	WhereFieldName	T/M	result
18	c	10		M	
19	c		10	M	The remote server returned an error: (500) Internal Server Error.
20	10	c		M	The remote server returned an error: (500) Internal Server Error.
21	10		c	M	The remote server returned an error: (500) Internal Server Error.
22		c	10	M	The remote server returned an error: (500) Internal Server Error.
23		10	c	M	

Function name:

? GetNearbyLocations: Returns a DataSet containing all locations within a given radius of a zip code

Parameter in:

? ZipCode as String

? RadiusMiles as integer

Parameter out

? String

Number of Test Cases	1
Number of Created Mutant	5
Number of Equivalent mutant	0
Number of Killed mutant	5
Number of Sent Request	6
Number of Received Errors	2
Mutation Score	100%

id	ZipCode	RadiusMiles	T/M	result
0	22041	10	T	20001WASHINGTONDISTRICT...
1	10	22041	M	
2		22041	M	
3	22041		M	The remote server returned an error: (500) Internal Server Error.
4	10		M	The remote server returned an error: (500) Internal Server Error.
5		10	M	

Function name:

? GetDistance: returns the decimal distance between two zip codes

Parameter in:

? ZipCode as String

? ZipCode as String

Parameter out

? String

Number of Test Cases	1
Number of Created Mutant	5
Number of Equivalent mutant	1
Number of Killed mutant	4
Number of Sent Request	6
Number of Received Errors	4
Mutation Score	80%

id	ZipCode1	ZipCode2	T/M	result
0	22041	22042	T	2.733824
1	22042	22041	M	2.733824
2		22041	M	The remote server returned an error: (500) Internal Server Error.

id	ZipCode1	ZipCode2	T/M	result
3	22041		M	The remote server returned an error: (500) Internal Server Error.
4	22042		M	The remote server returned an error: (500) Internal Server Error.
5		22042	M	The remote server returned an error: (500) Internal Server Error.

Next we will test The Html2Xml Web Service that is created by Reflection IT. The Web Service WSDL address is <http://www.html2xml.nl/Services/html2xml/version1/Html2Xml.aspx?wsdl>. The Html2Xml webservice takes a Url of a page and converts it into a well-formed Xml.

Function name:

? URL2XMLNode

Parameter in:

? urlAddress as String

Parameter out

? String

Number of Test Cases	1
Number of Created Mutant	1
Number of Equivalent mutant	0
Number of Killed mutant	1
Number of Sent Request	2
Number of Received Errors	1
Mutation Score	100%

id	urlAddress	T/M	result
0	http://www.google.com	T	Google
1		M	The remote server returned an error: (500) Internal

id	urlAddress	T/M	result
			Server Error.

Last Web Service to test is the Official Exchange Rates of the Litas against Foreign Currencies. This Web Service provides official (established by Bank of Lithuania) exchange rates of the Litas against Foreign Currencies. The Web Service WSDL address is <http://webservices.lb.lt/ExchangeRates/ExchangeRates.asmx?wsdl>.

Function name:

?        `getCurrentExchangeRate`

Parameter in:

?        Currency as string - Currency code, example USD

Parameter out

?        Decimal number - Exchange Rate (expressed in Litas per 1 currency unit).

?        If given parameter is invalid or error appear than negative number is returned.

Number of Test Cases	1
Number of Created Mutant	1
Number of Equivalent mutant	0
Number of Killed mutant	1
Number of Sent Request	2
Number of Received Errors	0
Mutation Score	100%

id	Currency	T/M	result
0	USD	T	2.9231
1		M	-1

Function name:

?        `getExchangeRate`

Parameter in:

?        Currency as string- Currency code, example JPY

?        Date as string - Exchange Rate date, example 2002-09-01



Parameter out

- ?        Decimal number - Exchange Rate (expressed in Litas per 1 currency unit).
- ?        If given parameter is invalid or error appear than negative number is returned.

Number of Test Cases	1
Number of Created Mutant	5
Number of Equivalent mutant	0
Number of Killed mutant	5
Number of Sent Request	6
Number of Received Errors	0
Mutation Score	100%

id	Currency	Date	T/M	result
0	USD	2004-03-03	T	2.7853
1	2004-03-03	USD	M	-1
2	USD		M	-1
3	USD		M	2.9231
4	2004-03-03		M	-1
5		2004-03-03	M	-1

Function name:

- ?        getExchangeRatesByDate

Parameter in:

- ?        Date - Exchange Rate's date, example 2002-10-05

Parameter out:

- ?        Exchange Rates as string

Number of Test Cases	1
Number of Created Mutant	1
Number of Equivalent mutant	0
Number of Killed mutant	1
Number of Sent Request	2

Number of Received Errors	0
Mutation Score	100%

id	Date	T/M	result
0	2004-03-03	T	2004.03.03AED107.5472LTL per 10 currency units 2004.03.03ALL1002.6239LTL per 100 currency units 2004.0
1		M	2004.04.23AED107.8346LTL per 10 currency units 2004.04.23ALL1002.6995LTL per 100 currency units 2004.0

Function name:

?       getExchangeRatesByCurrency

Parameter in:

?       Currency as string- Currency code, example SEK

?       DateLow as string - Interval low date, example 2002-10-04

?       DateHigh as string - Interval high date, example 2002-10-24

Parameter out:

?       Exchange Rates as string

Number of Test Cases	1
Number of Created Mutant	23
Number of Equivalent mutant	1
Number of Killed mutant	22
Number of Sent Request	24
Number of Received Errors	0
Mutation Score	100%

id	Currency	DateLow	DateHigh	T/M	result
0	USD	2004-03-03	2004-04-04	T	2004.03.03USD12.7853LTL per 1 currency unit 2004.03.04USD12.8311LTL per 1 currency unit 2004.03.05USD1
1	USD	2004-04-04	2004-03-03	M	Parameter value is invalid!
2	2004-03-	USD	2004-04-	M	Parameter value is invalid!

id	Currency	DateLow	DateHigh	T/M	result
	03		04		
3	2004-03-03	2004-04-04	USD	M	Parameter value is invalid!
4	2004-04-04	USD	2004-03-03	M	Parameter value is invalid!
5	2004-04-04	2004-03-03	USD	M	Parameter value is invalid!
6		2004-03-03	USD	M	Parameter value is invalid!
7		USD	2004-03-03	M	Parameter value is invalid!
8	2004-03-03		USD	M	Parameter value is invalid!
9	2004-03-03	USD		M	Parameter value is invalid!
10	USD		2004-03-03	M	Parameter value is invalid!
11	USD	2004-03-03		M	2004.03.03USD12.7853LTL per 1 currency unit2004.03.04USD12.8311LTL per 1 currency unit2004.03.05USD1
12	2004-04-04		USD	M	Parameter value is invalid!
13	2004-04-04	USD		M	Parameter value is invalid!
14		2004-04-04	USD	M	Parameter value is invalid!
15		USD	2004-04-04	M	Parameter value is invalid!
16	USD	2004-04-04		M	2004.04.04USD12.8141LTL per 1 currency unit2004.04.05USD12.8019LTL

id	Currency	DateLow	DateHigh	T/M	result
					per 1 currency unit2004.04.06USD1
17	USD		2004-04-04	M	Parameter value is invalid!
18	2004-04-04	2004-03-03		M	Parameter value is invalid!
19	2004-04-04		2004-03-03	M	Parameter value is invalid!
20	2004-03-03	2004-04-04		M	Parameter value is invalid!
21	2004-03-03		2004-04-04	M	Parameter value is invalid!
22		2004-04-04	2004-03-03	M	Parameter value is invalid!
23		2004-03-03	2004-04-04	M	Parameter value is invalid!

## *Chapter 5*

### **Conclusion**

We have introduced a new method for testing Web Services involving the application of mutation analysis to the WSDL. The procedure depends on the usage of WSDL specific mutant operators to mutate the WSDL document of a Web Service. We have identified mutation operators and experimented with their functionalities. The mutant operators list described in this paper is a basic list that is not exhaustive and will be further elaborated in subsequent papers. The application of mutation operators to the WSDL documents reveals interface errors, as well as logical ones. The derived results present the success and adequacy of our method.

In future work, we envision a Tester Web Service that provides almost complete automated testing of other Web Services. Figure 13 presents the tester Web Services flow. This Web Service would take as input a WSDL path of another Web Service, and the desired mutant score value. The Tester Web Service will generate test cases and execute them on the Web Service that is being tested using the original interface description, and then using mutants. If test case doesn't kill any mutant, it will be removed from the list, otherwise it will be recorded in a database. This method is repeated, each time generating test cases to target live mutants, until the desired mutant score value is reached or the tester stops the process. At the end the Web Service will output an XML document containing effective test cases and the generated output that the tester needs manually to examine in order to discover if it is the expected output or a fault in the service. This Web Service could be an extension of the provided web application; the main difference is that it would be a completely automated Web Service.

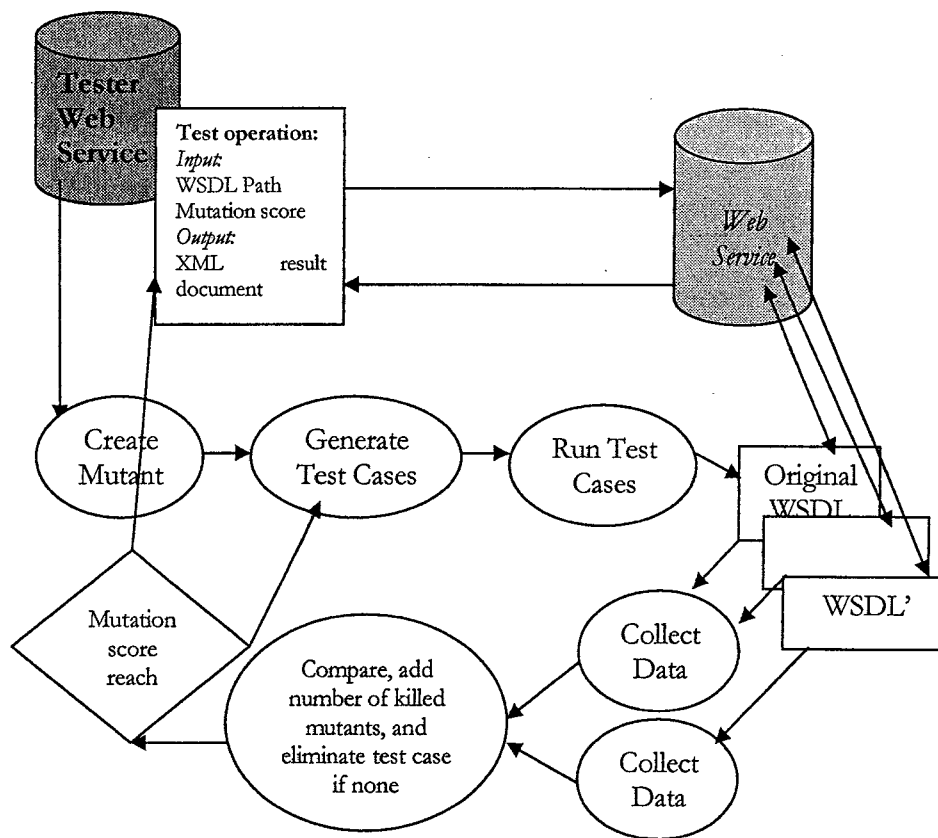


FIGURE13: TESTER WEB SERVICE

## References

- W3C (2003). Web Service architecture, *W3C working Draft*, [www.w3.org/tr/ws-arch/](http://www.w3.org/tr/ws-arch/).
- W3C (2001). Web Services description language (WSDL) 1.1. *W3C Note 15*, [www.w3.org/tr/wsdl](http://www.w3.org/tr/wsdl).
- DeMillo, R. A. and Offutt, A. J. (1991) Constraint-based automatic test data generation. *IEEE transaction on software engineering*, 17(9), 900-910.
- Offutt, A. J. and Untch, R. H (2000) Mutation 2000: Uniting the orthogonal. *Mutation testing in the twentieth and twenty-first centuries*, 45-55.
- Ghosh, S. and Mathur, A. P. (2001) Interface Mutation. *Software testing verification and reliability*, 11, 227-247.
- Yoon, H. and Choi B. (2001) An effective testing technique for component composition in EJBs. *Proceedings of eight Asia-pacific software engineering conference*.
- Martins, E. and Toyota, C. (2001), Constructing self-testable software components. *Proceedings of the international conference on dependable systems and networks*.
- Delamaro M., Maldonado, J., and Mathur, A. (2001) Interface mutation: an approach for integration testing. *IEEE transactions on software engineering*, 27(3), 228-247.
- Chan, W., Chen, T., and TSE T. (2002) An overview of Integration testing techniques for object oriented programs. *proceedings of the 2<sup>nd</sup> ACIS*.
- Vincenzi, A., Maldonado, J., Barbosa, E., and Delamaro M. (2001) Unit and integration testing strategies for C programs using mutation. *software testing, verification and reliability*; 11, 249-268.
- W3C (1998). Extensible Markup Language (XML) 1.0, *W3C recommendation*, <http://www.w3.org/TR/1998/REC-xml-19980210>
- Microsoft (2002). ASP.NET, <http://msdn.microsoft.com/asp.net/>
- Microsoft (2003). Visual Basic .Net, <http://msdn.microsoft.com/vbasic/>

## Glossary

### ***ASP.NET***

ASP.NET (originally called ASP+) is the next generation of Microsoft's Active Server Page (ASP). It is a component of the Microsoft .NET Framework for building, deploying, and running Web applications and distributed applications.

### ***CORBA***

Common Object Request Broker Architecture (CORBA) is an architecture and specification for creating, distributing, and managing distributed program objects in a network. It allows programs at different locations and developed by different vendors to communicate in a network through an "interface broker."

### ***COM***

Component Object Model (COM) is Microsoft's framework for developing and supporting program component objects. COM provides the underlying services of interface negotiation, life cycle management licensing, and event services.

### ***HTTP***

Hypertext Transfer Protocol (HTTP) is the set of rules for transferring files (text, graphic images, sound, video, and other multimedia files) on the World Wide Web. HTTP is an application protocol that runs on top of the TCP/IP suite of protocols.

### ***IDL***

Interface definition language (IDL) is a generic term for a language that lets a program or object written in one language communicates with another program written in an unknown language. An interface definition language works by requiring that a program's interfaces be described in a stub or slight extension of the program that is compiled into it.

### ***RPC***

Remote Procedure Call (RPC) is a protocol that one program can use to request a service from a program located in another computer in a network without having to understand network details.

### ***SOAP***

Simple Object Access Protocol (SOAP) is a simple, XML-based protocol for exchanging structured data and type information on the World Wide Web.

### ***TCP/IP***



Transmission Control Protocol/Internet Protocol (TCP/IP) is the basic communication language or protocol of the Internet. It can also be used as a communications protocol in a private network (either an intranet or an extranet).

### ***UDDI***

Universal Description, Discovery, and Integration (UDDI) is a specification for publishing and locating information about Web Services. It defines a standards-based way to store and retrieve information about services, service providers, binding information, and technical interface definitions, all classified using a set of standard or custom classification schemes.

### ***URI***

Uniform Resource Identifier (URI) is the way you identify any points of content on the internet, whether it be a page of text, a video or sound clip, a still or animated image, or a program. A URI typically describes the mechanism used to access the resource, the specific computer that the resource is housed in, and the specific name of the resource on the computer.

### ***VB.NET***

Visual Basic (VB) is a programming environment from Microsoft in which a programmer uses a graphical user interface to choose and modify selected sections of code written in the BASIC programming language.

VB.NET is part of a brand new platform, based on the .NET Framework, and is fully object-oriented.

### ***WSDL***

Web Service Description Language (WSDL) is an XML format for describing Web Services. WSDL allows Web Service providers and users of such services to work together easily by enabling the separation of the description of the abstract functionality offered by a service from concrete details of a service description such as "how" and "where" that functionality is offered.

### ***XML***

Extensible Markup Language (XML) is a markup language that provides a format for describing structured data. XML is a World Wide Web Consortium (W3C) specification and is a subset of Standard Generalized Markup Language (SGML).

### ***XSD***

XML Schema Definition (XSD), a Recommendation of the World Wide Web Consortium (W3C), specifies how to formally describe the elements in an Extensible Markup Language (XML) document. This description can be used to verify that each

item of content in a document adheres to the description of the element in which the content is to be placed.