JOURNAL OF SOFTWARE MAINTENANCE AND EVOLUTION: RESEARCH AND PRACTICE
J. Softw. Maint. Evol.: Res. Pract. 2007; **19**:297–314
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/smr.354

Special Issue

The design and use of WSDL-Test: a tool for testing Web services

Harry M. Sneed[1] and Shihong Huang[2,*,†]

[1]*Anecon GmbH, Vienna, Austria*
[2]*Department of Computer Science & Engineering, Florida Atlantic University, 777 Glades Road, Boca Raton, FL 33431-0991, U.S.A.*

SUMMARY

Web services are becoming increasingly important to many businesses, especially as an enabling technology for systems that adopt a service-oriented architecture approach to their development. However, testing Web services poses significant challenges. This paper describes the design and use of WSDL-Test, a tool designed specifically for this purpose. A key feature of WSDL-Test is its ability to simulate the actual usage of Web services in a controlled environment. This enables WSDL-Test to generate requests and validate responses in a rapid and reliable manner. To illustrate the use of WSDL-Test, the paper also discusses our experience using the tool on a real-world online eGoverment application. Copyright © 2007 John Wiley & Sons, Ltd.

Received 12 February 2007; Revised 23 July 2007; Accepted 31 July 2007

KEY WORDS: Web services; WSDL; XML; testing; automation

1. INTRODUCTION

Web services are becoming increasingly important to the IT business, especially since the advent of service-oriented architecture (SOA). IT users are looking for a way to increase the flexibility of their IT systems so as to be able to react quickly to changes in their environment. If a competitor comes up with a new marketing approach, they have to be able to follow that approach in a short

*Correspondence to: Shihong Huang, Department of Computer Science & Engineering, Florida Atlantic University, 777 Glades Road, Boca Raton, FL 33431-0991, U.S.A.
†E-mail: shihong@cse.fau.edu

time. Adaptability of the IT systems has become critical to the survival of a company. If a new law is legislated, such as the Sorbane Oxley Act, companies have to be able to implement it within weeks. Changes to laws and regulations cannot be postponed. They have to be implemented by a given deadline, which is often only a short time away.

Under such time pressure, it is no longer possible to plan and organize long-running projects. It is necessary to design and assemble a working solution within a limited time. This requirement for immediate response presupposes the existence of reusable components, which can be glued together within a standard framework to support a customized business process. The standard framework is an SOA such as that offered by IBM, Oracle, and SAP [1]. The components are the Web services; the overlying business process can be defined with the business process execution language (BPEL) [2]. The glue for binding the business process to the Web services as well as to link the Web services to one another is the Web service description language (WSDL) [3]. The Web service components themselves are derived from various sources. Some are bought, some are taken from the open-source community, some are newly developed, and others are taken from the existing software systems, i.e., they are recycled to be reused in the new environment. Normally, this entails wrapping them [4].

Regardless of where they come from, no one can ensure that the Web service components will work as one might expect. Even those that are bought may not fit exactly to the task at hand. The fact that they are not compatible can lead to serious interaction errors. The recycled components may be even worse. Legacy programs tend to contain many hidden errors, which in a given context counterbalance each other. However, when moved to another environment to perform a slightly different function, the errors suddenly emerge to the surface. The same can happen with open-source components. Perry and Kaiser have demonstrated that the correctness of a component in one environment will not hold for another environment. Therefore, components have to be retested for the every environment in which they are reused [5].

In the case of self-developed services, the reliability problem is the same as with all new software. They have to be subjected to extensive testing at all levels—at the unit level, at the component level, and, finally, at the system level. Experience with new systems shows that the error rate of newly developed software varies between 3 and 6 errors per 1000 statements [6]. These errors have to be located and removed before the software goes into production. A significant portion of these errors is due to false assumptions the developer has about the nature of the task and the behavior of the environment. Such errors can only be uncovered by testing in the target environment—with data produced by others with a different perspective on the requirements. This is the primary rationale for independent testers.

No matter where the Web services come from, they should go through an independent testing process, not only individually but also in conjunction with one another. This process should be well defined and supported by automated tools, so that it is quick, thorough, and transparent. Transparency is of particular importance in testing Web services so that test cases can be traced and intermediate results can be examined. Owing to the volume of test data required, it is also necessary to automatically generate the inputs and to automatically validate the outputs. By generating varying combinations of representative test data, a high rate of functional coverage is attained. By comparing the test results with expected results, a high degree of correctness is ensured [7].

2. RELATED WORK

As Web services have become increasingly important to pervasive computing, significant efforts have been spent on the testing of Web services from both academia and industry. This section briefly comments on some of the related efforts.

2.1. Current research

Coyote [8] is an example of earlier work on providing an XML-based object-oriented testing framework that incorporates concepts from object-oriented application framework to test Web services rapidly. In order to address the insufficient information, such as lacking of dependence information, provided by WSDL file of Web services, four extensions of WSDL have been proposed in [9] to facilitate Web services testing. Considering applications areas for the semantic Web, Narayanan *et al.* [10] proposed a method to enable markup and automated reasoning technology to describe, simulate, compose, test, and verify compositions of Web services. They used DAML-S DAML+OIL ontology for describing the capabilities of Web services. A similar work of using DAML-S for service description to find the semantic match between a declarative description of the service being sought and a description of the service being offered is described in [11].

Nguyen [12] described the technical challenges of testing Web applications and presented the results of Web-testing strategies at several Fortune 500 companies. He outlined some of the aspects that need to be considered to improve the return on investment of Web testing. In the context of re-engineering existing Web sites into Web services, Jian and Stroulia [13] focus on characterizing the interaction between Web services and client browsers. These interactions are expressed in XML specifications that are syntactically and semantically close to WSDL.

2.2. Existing tools

There is no lack of tools for testing Web services. In fact, the market is full of them. The problem is not so much with the quantity, but with the quality of the tools. Most of them are recent developments that have yet to mature. They are also difficult to adjust to the local conditions and require users to submit data via the Web client user interface. Testing through the user interface is not the most effective means of testing Web services, as has been pointed out by R. Martin in a recent contribution to the IEEE Software Magazine. He suggests using a testbus to bypass the user interface and to test the services directly [14]. This is an approach that has been followed by the authors.

A typical tool on the market is the Mercury tool 'Quicktest Professional'. It allows the users to fill out a Web page and to submit it. It then follows the request from the client workstation through the network. This is done by instrumenting the SOAP message. The message is traced to the Web service that processes it. If that Web service, invokes another Web service, then the link to that service is followed. The contents of each WSDL interface are recorded and kept in a trace file. In this way, the tester is able to trace the path of the Web service request through the architecture and to examine the message contents at different stages of processing [15].

Parasoft offers a similar solution. However, rather than starting the request from a Web client, it generates requests from the business process procedures written in BPEL. This creates, on the one hand, a larger volume of data, while, on the other hand, simulating real-world conditions. It is expected that most requests for Web services will come from the business process scripts that are driving the business processes. BPEL language has been developed for that purpose; hence, it is natural to test with it. What is missing in the Parasoft solution is the capability of verifying the responses. They have to be inspected visually [16].

One of the pioneers in Web testing is the Empirix. The e-Tester tool from Empirix allows testers to simulate the business processes using the Web clients. Their requests are recorded and translated into test scripts. The testers can then alter and vary the scripts to mutate one request into several variations for making a comprehensive functional test. The scripts are in Visual Basic for Applications, so it is easy for any person familiar with VB to work with them. With the scripts it is further possible to verify the response results against the expected results. Unexpected results are sorted out and fed to the error-reporting system [17].

Other testing companies such as Software Research Associates, Logica and Compuware are all working on similar approaches. Hence, it is only a question of time until the market is flooded with Web service testing tools. After that it will take some time before the desired level of tool quality is reached. Until this is the case, there is still some potential for customized solutions such as the one described in this paper.

What is new in the approach that is presented in this paper is the generation of data from the schema definition. What is also new is the use of the same assertion language both for generating request parameters and for verifying response results. There are other tools such as the Oracle Test Suite that use assertions to verify responses. However, they do not generate requests from the schema nor do they both generate requests and verify responses using the same script. This is as far as we know original to this approach.

3. THE WSDL-TEST APPROACH

The WSDL-Test tool takes a slightly different approach than the other commercial Web service testing tools. It starts with a static analysis of the schema of the WSDL description. From this analysis two objects are generated. One is a WSDL request with random data and the other is a test script. The test script allows the user to manipulate the arguments in the Web service request. It also allows the user to verify the results in the Web server response. The test driver is a separate tool which reads and dispatches the Web service request and which receives and writes the Web service response.

3.1. The motivation for developing this tool

It is often the circumstances of a project that motivate the development of a tool. In this case the project was to test an eGovernment Web site. The general user, the citizen, was to access the Web site through the standard Web user interface. However, the local governments had IT systems, which also needed to access the site in order to obtain information from the central state database. For this purpose it was decided to offer them a Web service interface. Altogether nine different services were defined, each with its own request and response formats.

The user interface to the eGovernment Web site was tested manually by human testers simulating the behavior of potential users. For the Web services, a tool was needed to simulate the behavior of the user programs by automatically generating typical requests and submitting them to the Web service. Since the responses from the Web service are not readily visible, it was also necessary to automatically validate the responses. Thus, the motivation for developing the tool could be summarized as follows:

- Web services cannot be trusted; therefore, they must be subjected to intensive testing.
- All requests with representative argument combinations should be tested.
- All responses with all representative result states should be validated.
- To test Web services, it is necessary to generate WSDL requests with specific arguments for verifying target functions.
- To verify the correctness of Web services, it is necessary to validate the WSDL responses against expected results.

3.2. Generating a random request from the WSDL schema

All tests are a test against something. There has to be a source of the test data and there has to be an oracle to compare the test results against [18]. In the case of WSDL-Test, the oracle is the WSDL schema. That schema is either generated automatically from the interface design or the developer writes it manually. As a third and more advanced alternative, it can be created from the BPEL process description. Regardless of how it is created, the schema defines the basic complex data types in accordance with the rules of the XML schema standard. Complex data types can include other complex data types so that the data tree is represented with single and multiple occurrences of the tree nodes. The schema then defines the base nodes, i.e., the tops of the tree and their sequence. These are the actual parameters. An excerpt from the type definitions of an eGovernment schema is shown in Figure 1.

Following the parameter description comes the message descriptions that identify the names and the component parts of each message, whether it is a request or a response. After that follows the port type definitions. Each service operation to be invoked is listed out with the names of its input and output messages. These message names are references to the messages defined before, which again are references to the parameters defined before that. After the port types come the bindings describing the SOAP prototypes composed of service operations. At the end, the Web service interface is given a name.

A WSDL schema is a tree structure where the SOAP prototypes refer to the service operations that in turn refer to the logical messages, which in turn refer to the parameters which refer to the various data types. The data types may in turn refer to one another. Parsing this tree is called tree walking. The parser selects a top node and follows it down through all of its branches collecting all of the subordinate nodes on the way down. At the bottom of each branch, it will find the basic data types such as integers, booleans, and strings. WSDL-Test goes a step further by assigning each basic data type a set of representative data values. For instance, integer values are assigned a range from 0 to 10 000 and string values are assigned varying character combinations. These representative data sets are stored in tables and can be edited by the user prior to generating the test data.

```
<definitions>
  <types>
    <schema>
      <complexType name="getProfile">
        <sequence><element name = "ZbsWsRequest_1"></sequence>
      </complexType>
      <complexType name="ZbsWsRequest">
         <sequence>
           <element name="applikationId" type="string" nillable="true"/>
         </sequence>
      </complexType>
    </schema>
  </types>
  <message name = "ZbsWebService_getProfile">
    <part name="parameters" element="getProfile"/>
  </message>
  <portType name="ZbsWebServiceInterface">
    <operation name="getProfile">
      <input message = "Interface_getProfile"/>
      <output message = "getProfileResponse"/>
      <fault name="ZbsWsException" message = "ZbsWsException"/>
    </operation>
  </portType>
  <binding name="ZbsWebServiceInterfaceBinding"
           type="tns:ZbsWebServiceInterface">
    <soap:binding transport=http://schemas.xmlsoap.org/soap/http
                  style="document"/>
  </binding>
</definitions>
```

Figure 1. eGovernment schema excerpt.

Thus, the hierarchical structure of the WSDL interface is as follows:

```
Web service interface
      →SOAP prototypes
            →service operations
                  →logical messages
                        →parameters
                              →data types
                                    →elementary data types
                                          →representative values
```

The task of the data generator is to walk the WSDL schema tree down to the level of the basic data types and to select representative values for that type. The values are selected randomly from the set of possible values. From these values an XML data group is created as follows:

```
<Account>
   <Account_Number>100922</Account_Number>
   <Account_Owner> Smith</Account_Owner>
   <Account_Balance> 999.50</Account_Balance>
   <Account_Status> 2</Account_Status>
</Account>
```

In this way a WSDL service request file with sample data is generated and stored for future use. Simultaneously, a test script is created, which allows the tester to override the values originally generated. This script is a template with the data element names and their values:

```
Account: WSDL
    assert new.Account_Number = "100922"
    assert new Account_Owner = "Smith"
    assert new Account_Balance = "999.50"
    assert new Account_Status = "2";
end;
```

By means of altering the script, the tester can now alter the values of the service request. Since a script is also generated for the output data, the tester is given a template for verifying the service responses.

3.3. Writing pre-condition assertions

The test scripts for WSDL-Test are sequences of pre-condition assertions defining possible states of the Web service request. A state is a combination of given values for the data types specified in the WSDL interface definition. The values are assigned to the individual data elements, but their assignment may be mutually dependent so that a particular combination of values can be determined by the tester:

```
assert new. Account_Status = "y"
        if (old.Account_Balance < "0") ;
```

For assigning test data values there are six different assertion types:

1. the assignment of another existing data value from the same interface;
2. the assignment of a constant value;
3. the assignment of a set of alternate values;
4. the assignment of a value range;
5. the assignment of a concatenated value;
6. the assignment of a computed value.

The assignment of another existing data value is done by referring to that value. The value referred to must be within the same WSDL:

```
assert new.Account_Owner = old.Customer_Name;
```

The assignment of a constant value is done by giving the value as a literal in that statement. All literals are enclosed in quotes:

```
assert new. Account_Balance = "0";
```

The assignment of a set of alternate values is made by means of an enumeration. The enumerated values are separated by an exclamation point '!':

```
assert new.Account_Status = "0" ! "1" ! "2" ! "3";
```

According to this assertion, the values will be assigned alternately starting with 0. The first account occurrence will have the status 0, the second the status 1, the third the status 2, and so on.

The assignment of a value range is for the purpose of boundary analysis. It is made by giving the lower and upper bounds of a numeric range:

```
assert new.Account_Status = ["1" : "5"];
```

This will cause the assignment of the values 0, 1, 3, 5, 6 in alternating sequence. The tool produces values one less than the lower bound, the lower bound, the upper bound, one more than the upper bound and the geometric median.

The assignment of a concatenated value is done by joining two or more existing values with two or more constant values in a single string:

```
assert new.Account_Owner =
"Mr." | Customer_Name | "from" | Customer_City;
```

The assignment of a computed value is given as an arithmetic expression in which the arguments may be existing values or constants:

```
assert new. Account_Balance = old.Account_Balance / "2" + "1";
```

The assert assignments can be conditional or unconditional. If they are conditional, they are followed by a logical expression comparing a data variable of the WSDL interface with another data variable of the same interface or with a constant value:

```
assert new.Account_Owner = "Smith"
    if ( old.Account_Number = "100922"&
          old. Account_Balance > "1000" ) ;
```

The tester adapts the assertion statements in the script generated from the WSDL script to provide a representative test data profile including equivalence classes and boundary analysis as well as progressive and degressive value sequences. The goal is to manipulate the input data so that a wide range of representative service requests can be tested. To achieve this, the tester should be familiar with what the Web service is supposed to do and to assign the data values accordingly (see Figure 2).

3.4. Overriding the random data

Once the assertion scripts are available, it is possible to overwrite the random data in the Web service request by the asserted data. This is the task of the XMLGen module. It matches the WSDL file generated by the WSDLGen module with the assertion script written by the tester. The data names in the assertion script are checked against the WSDL schema and the assertions compiled into symbol tables. There are different tables for the constants, the variables, the assignments, and the conditions.

After the assertion script has been compiled, the corresponding WSDL file is read and the data values are replaced by the values derived from the assertions. If the assertion refers to a constant, the constant replaces the existing value of the XML data element with the name corresponding to that in the assertion. If the assertion refers to a variable, the value of the XML data element with

```
file: ZBS-WS;
  if ( object = "ZbsWsRequest" );
     assert new.applikationId = old.applikationId;
     assert new.assertionId = old.assertionId;
     assert new.komponentenId = "4711";
     assert new.version = "1";
  endObject;
  if ( object = "ZbsWSProfile");
     assert new.attributeName = old.attributeName;
     assert new.typ = 21 ! 22 ! 23;
     assert new.value = "Sneed";
  endObject;
  if ( object = "ZbsWsAuthentificationAssertion" );
     assert new.assertionId = "Kati";
     assert new.version = "1";
     assert new.notAfter = "notAfter";
     assert new.notBefore = "notBefore";
  endObject;
end;
```

Figure 2. Sample of assign data values.

that variable name is moved to the target data element. Alternate values are assigned one after the other in ascending order until the last value has been reached, then it starts again with the first value. Range values are assigned as the boundary values plus and minus one.

In the end, a sequence of Web service requests exists with varying representative states combining the original generated data with the data assigned by the assertion scripts. By altering the assertions, the tester can alter the states of the requests, thus ensuring maximum data coverage.

3.5. Activating the Web services

Having generated the Web service requests, it is now possible to dispatch them to the server. This is the task of the test driver. It is a simulated BPEL process with a loop construct. The loop is driven by a list of Web services ordered by the sequence in which they should be invoked. The tester can edit the list to alter the sequence in accordance with the test requirements.

From the list the test driver takes the name of the next Web service to be invoked. It then reads the generated WSDL file with the name of that Web service and dispatches a request to the specified service. When testing in synchronic mode, it will wait until a response is received before dispatching the next request. When testing in asynchronous mode, it will dispatch several requests until it comes to a wait command in the Web service list. At this point it will wait until it has received responses for all of the services dispatched before continuing with the next request.

The responses are accepted and stored in separate response files to be verified later by a post-processor. It is not the job of the driver to create requests or to check the responses. The requests are created by the pre-processor; the driver only dispatches them. The responses are checked by the post-processor; the driver only stores them. In this way, the role of the test driver is reduced to that of a simple dispatcher. BPEL procedures are very appropriate for that, since they have all of the necessary features for invoking Web services within a predefined workflow (see Figure 3).

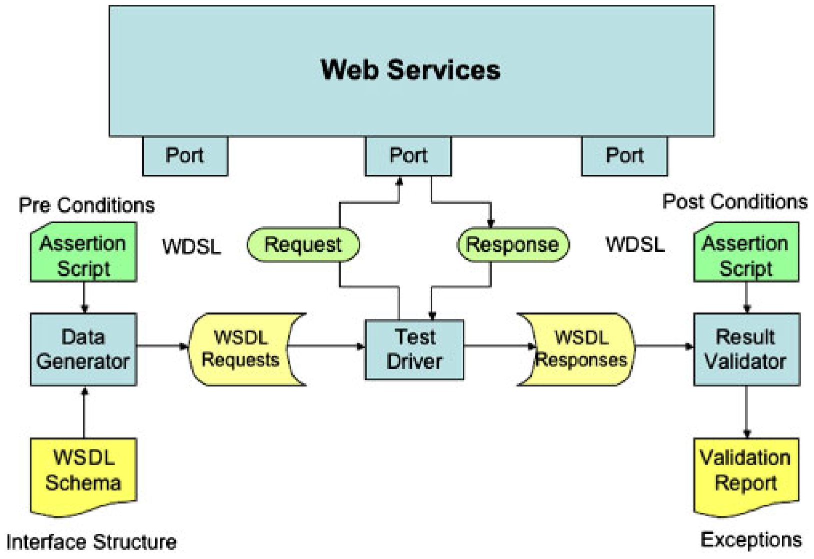

Figure 3. Web service test driver.

3.6. Writing post-condition assertions

The same assertion language is used for verifying the Web service responses as is used for constructing the Web service requests. Only, here, the assertions have an inverse meaning. Data are not assigned from existing variables and constants, but compared with the data values of previous responses or with constant values.

```
assert new.Account_Owner = old.Account_Owner;
```

implies that the value of the data element Account_Owner should match the value in the last response.

```
assert new.Account_Balance = "33.50";
```

is obviously a simple comparison.

```
assert new.Account_Status = "0" ! "1" ! "2" ! "3" ;
```

means that the value of the Account_Status must match at least one of the alternate values.

```
assert new.Account_Balance = ["100.00" : "500.00"]
```

implies that the Account_Balance in the Web service response must be within the range of 100–500.

The computed assertion, e.g.,

```
assert new.Account_Balance = old. Account_Balance - 50;
```

first computes a value and then compares that value with the actual value in the response. As in the case of the pre-conditions, the post-conditions can be unconditional or conditional.

If they are conditional then they are qualified by a logical *if* expression, comparing two variables in the Web service response or comparing a variable in the response with a constant value:

```
assert new. Account _Status = "3"
  if ( old.Account_Balance < "1000" ) ;
```

In all cases, if the assertion is not true, an error message is recorded, displaying both the expected and the actual value. Provided there is an assertion check for each attribute of the WSDL response, the verification of the response will be 100%. It is not, however, required to check every attribute. The tester may decide to restrict the check to only critical variables. If so, the data coverage will be less. Data coverage is measured in terms of the number of asserted results relative to the sum of all results.

3.7. Validating the responses

The module XMLVal fulfills the task of verifying the Web service results. For this it must first compile the post-condition assertions into internal tables of variable references, constants, enumerations, and ranges. In doing so, it checks the data names and types against the names and types declared in the WSDL schema to ensure consistency.

Having succeeded in compiling the assertion scripts, the tool then uses the compiled table to check the Web service response. First, the expected values are stored in a table with a key for each object occurrence. Second, it parses the WSDL result file matching the objects there with the objects in the assertion tables. If a match is found, the attributes of that object are extracted and their values compared with the expected values. If they do not match the verification condition, the data names and values are written out in a list of non-matching results. It is then the task of the tester to explore why the results do not match.

In addition to listing out the assertion violations, the XMLVal tool also produces a statistic on the degree of data coverage and the degree of correctness (see Figure 4).

4. COMPONENTS OF THE WSDL-TEST TOOL

The WSDL-Test tool was put together from several components. These include a GUI shell, an XML file reader and writer, an assertion compiler, an XSD tree walker, a WSDL analyzer, a table processor, an error handler, random and selective request data generators, a request validator, a validation report generator, and a WS test driver. Some of these components were taken from existing tools; others were newly created. The algorithms for processing XML schemas such as tree walking are published in the literature [19] and can be reused. The assertion language and the assertion compiler were adopted from a previous tool and extended. Thus, a prototype version could be made operational in less than a month. Later the tool was refined and enhanced. In its original form, the tool was constructed with the shell implemented with Borland Delphi and the core implemented with Borland C++. The shell and core are connected via an XML parameter file. The tool uses no database, it only uses temporary work files and was designed to run in a Microsoft Windows environment.

```
+-------------------------------------------------------------------------------------+
|                             WSDL Response Validation Report                         |
|                                                                                     |
| File:   ZBS-WS.wsdl                                                  Params: Y Y Y Y  |
| Object: ZbsWsAuthentificationAssertion                                 Date: 26.02.06 |
| Type  : WSDL                                                      System: WebService |
|                                                                                     |
| Key Fields of Record(new,old)                                                       |
+-------------------------------------------------------------------------------------+
| New:ZbsWsAuthentificationAssertion                                                  |
| Old:ZbsWsAuthentificationAssertion                                                  |
+---------------------------------------------+---------------------------------------+
| Non-Matching Fields                         | Non-Matching Values                   |
+---------------------------------------------+---------------------------------------+
| RecKey:27013                                |                                       |
| New: AssertionId                            | Marta                                 |
| Old: AssertionId                            | Kati                                  |
+---------------------------------------------+---------------------------------------+
| RecKey:27022                                |                                       |
| New: Version                                | 2                                     |
| Old: Version                                | 1                                     |
+---------------------------------------------+---------------------------------------+
+---------------------------------------------+---------------------------------------+
|  Total Number of old Responses checked:            10                               |
|  Number of old Responses found in new File:        10                               |
|  Number of old Responses not in new File:          00                               |
|  Number of new Responses found in old File:        10                               |
|  Number of new Responses not in old File:          00                               |
|  Total Number of Attributes checked:               70                               |
|  Total Number of non-Matching Attributes:          07                               |
|  Percentage of matching Attributes:                90 %                             |
|  Percentage of matching Responses:                100 %                             |
+-------------------------------------------------------------------------------------+
```

Figure 4. Response validation report.

4.1. The user interface

The Windows interface of WSDL-Test is designed to accept parameters from the user, to select files from a directory, and to invoke the backend processes. The parameters are:

- the name of the software product under test;
- the name of the software system under test;
- the name of the specific test object;
- the name of the test run itself;
- the types of the old and new files (csv, xml, wsdl);
- the separator characters (;/,/space);
- the comparison options.

The three directories from which files can be selected are:

- the assertion directory with the assertion text files;
- the old file directory with the test inputs—csv, sql, xml, and wsdl files;
- the new file directory with the test outputs—csv, xml, and wsdl files.

In addition, there is an output Directory where the protocols and reports are collected. It is essential that the file names are the same in all four directories, since this is how they are associated. Only the extension may vary. Thus, the actual response 'Message.wsdl' is compared with the expected response 'Message.wsdl' using the assertion script 'Message.wsdl' to produce the report 'Message.wsdl'. The assertion scripts have the extension .asr. The names of all files belonging to a particular project are displayed together with their types for the user to select from.

4.2. The assertion compiler

The assertion compiler was designed to read the assertion scripts and to translate them into internal tables that can be interpreted at test time. A total of nine tables can be generated for each file or object. These are:

1. A header table with information about the test object.
2. A key table with an entry for up to 10 keys used to relate the old and new files.
3. An assertion table with an entry for up to 80 assertion statements.
4. A condition table with an entry for each pre-condition to be fulfilled.
5. A constant table with an entry for each constant value to be compared or generated.
6. An alternates table with an entry for each alternate value an attribute may have.
7. A concatenation table with an entry for each concatenated value.
8. A computational table with the operands and operators of the arithmetic expressions.
9. A replacement table with up to 20 fields, whose values can be substituted by other values.

For SQL, XML, and WSDL, the tables are created for each object within a file. The header table identifies the test object and describes the file attributes. In case of SQL, XML, and WSDL, the header table also identifies the internal objects contained in the file. The key table contains the keys of the old and new records plus their positions and length, as well as constants to be used in lieu of the keys. The assertion table contains the names, types, positions, and lengths of the attributes to be generated or compared. The condition table contains the operands and logical operators for checking the pre-conditions. The constant table is a list of all numeric and string constants to be used either as test data or as compare values. The alternates table is an enumeration of the alternate values an attribute may take on. The concatenated table is a set of the substrings to be joined into a common string value. The computational table contains the operators and operands of the arithmetic expression. The replacement table is a list of those values assigned by the user to replace existing values in the old files.

The assertion compilation has to take place before a file can be generated or validated. It is of course possible to compile many assertion scripts at one time before starting with the generation and validation of files. The results of the assertion compilation are written out in a log file that is displayed to the user at the end of each compile run.

4.3. The WSDL request generator

The request generator consists of three components: (1) a schema pre-processor, (2) a random request generator, and (3) a data assigner. XML schemas are normally generated by some tools. When they are generated, many tags with many attributes are written together into one line as depicted below:

```
<complexType><sequence><element ........./>
</sequence></complexType>
```

To make the XML text more readable the schema pre-processor splits up such long lines into many indented short lines.

```
<complexType>
   <sequence>
        <element ...................../>
   </sequence>
</complexType>
```

This simplifies the processing of the schema by the subsequent components.

The random request generator then creates a series of WSDL requests using constant values as data. The assigning of constants depends on the element type. String data are assigned from representative strings, and numeric data by means of adding or subtracting constant intervals. Date and time values are taken from the clock. Within the request, the data may be given as a variable between the tags of an element or as an attribute to an element. Often the element content remains constant and the attribute values vary. This poses a problem since attributes do not have types. They are per definition strings. Hence, here only string values are assigned. The end result is a request in XML format with randomly assigned data elements and attributes.

The data assigner reads in the WSDL requests that were randomly generated and writes the same requests out again with adjusted data variables. In the assertion scripts the user assigns specific arguments to the request elements by name. By compiling the assertion scripts, the names and arguments are stored in symbol tables. For each data element of the request, the data assigner accesses the symbol tables and checks whether that data element has been assigned an asserted value or set of values. If so the random value of that element is overwritten by the asserted value. If not, the random value remains. In this way, the WSDL requests become a mixture of random values, representative values and boundary values. Through the assertions the tester can control what arguments are sent to the Web service.

The WSDL requests are stored in a temporary file where they are available to the request dispatcher. To distinguish between requests, each request is assigned a test case identifier as a unique key.

4.4. The WSDL response validator

The response validator consists of only two components: the response pre-processor and the data checker. The response pre-processor works in a similar way as the schema pre-processor. It unravels the responses that come back from the Web service to make them easier to process by the data checker. The responses are taken from the queuing file of the test dispatcher.

The data checker reads the responses and identifies the results either by their tag or by their attribute name. Each tag or attribute name is checked against the data names in the output assertion table. If a match is found, the value of that element or attribute is compared against the asserted value. If the actual value varies from the asserted value, a discrepancy is reported in the response validation report. This is repeated for every response.

In order to distinguish between responses, the test case identifier assigned to the request is also inherited by the response so that each response is linked to a particular request. This enables the tester to include the test case identifier in the assertions and to compare responses based on the test case.

4.5. The WSDL request dispatcher

The WSDL request dispatcher is the only Java component. It takes the generated requests from the input queue file, packs them into a SOAP envelope and sends them to the targeted Web service. If there is a problem in reaching the Web service it will handle the exception. Otherwise it will simply record the exceptions and go on to the next request. The responses are taken out of the SOAP envelope and stored in the output queue file, where they are uniquely identified by the test case identifier.

The request dispatcher also records the time when each test case is dispatched and the time when the response is returned. These are then the start and end times of that test case. By instrumenting the methods of the Web services with probes recording the time they are executed, it is possible to link specific methods to specific test cases. This makes it easier to identify errors in the Web services. It also enables the identification of features since a test case is equivalent to a path through one or more Web services. This is a subject for future research [20].

5. EXPERIENCE WITH WSDL-TEST IN AN EGOVERNMENT PROJECT

The experience with WSDL-Test in an eGovernment project has been reported on at the WSE 2005 workshop. Nine different Web services were tested there, with an average of 22 requests per service. Altogether 47 different responses were verified. Of these, 19 contained at least one erroneous result. Thus, of the more than 450 errors found within the project as a whole, some 25 were detected in the Web services [21].

In the mean time, the tool has been applied to a second project in the same environment. Up until now more than 40 errors have been uncovered by comparing the results. The project is still going on.

It would appear that the tool is an appropriate instrument for testing Web services, as long as the WSDL interfaces are not overly complex. If they are too complex, the task of writing the assertions becomes difficult and errors occur there. At this point the tester cannot be sure whether an observed error is caused by the Web service or by a wrongly formulated assertion. A similar experience was reported on from the U.S. ballistic missile defense project some 30 years ago. Some 40% of the errors reported there were actually errors in the testing procedures [22]. Looking at the testing technology used in that project by the RXVP test laboratory, it can be seen that basic test methods—setting pre-conditions, checking post-conditions, instrumenting the software, monitoring the test paths and measuring test coverage—have hardly changed since the 1970s. Only the environment has changed.

6. SUMMARY AND FUTURE WORK

This paper has reported on the design and use of WSDL-Test, a tool for supporting Web service testing. The tool WSDL-Test generates Web service requests from the WSDL schemas and adjusts them in accordance with the pre-condition assertions written by the tester. It dispatches the requests and captures the responses. After testing, it then verifies the response contents against the post-condition assertions composed by the tester.

The tool is still under development, but it has already been employed in an eGovernment project to expedite the testing of Web services offered by a German State Government. Future work will be in the direction of linking this tool to other test tools in supporting other test activities.

Testing software systems is a complex job that has yet to be fully understood. There are many facets of the task, such as specifying test cases, generating test data, monitoring test execution, measuring test coverage, validating test results, tracking system errors, and so on. The tool WSDL-Test addresses only two of these issues—generating test data and validating test results. Another tool Test Analysis analyzes the requirement documents to extract the functional and non-functional test cases. These abstract test cases are then stored in a test case database. It would be necessary to somehow use these test cases to generate the pre- and post-condition assertions. That would entail bridging the gap between the requirement specification and the test specification. The greatest barrier to achieving this is the informality of the requirement specs. The question is one of deriving formal, detailed expressions from an abstract, informal description. It is the same problem as faced by the model-driven development community.

Another direction for future work is test monitoring. It would be useful to trace the path of a Web service request through the system. This requires altering the server components to record what requests they are currently working on. By treating a trace file it would be possible to monitor the execution sequence of the Web services, as there are often many services involved in the processing of one request.

There still remains the basic question of to what degree the test should be automated. It may not be so wise to try and automate the whole Web testing process, but to rely instead on the skills and creativity of the human tester. Automation often tends to hide important problems. The issue of test automation versus creative testing remains a major topic in the testing literature [23].

The limitations to the approach presented in this paper are of both a technical and a theoretical nature. A technical limitation has to do with the restricted scope of the assertion scripts. Comparisons can only be made between variables defined within the WSDL schema. No data can be addressed other than those contained within the requests and responses. For instance, database attributes are out of scope for both request generation and response validation.

A theoretical limitation is the language used to assign arguments and verify results. It is currently designed to be as simple as possible to allow ordinary testers to use it, but it does so at the cost of expressive power. It does not allow the user to nest assertions or to define complex data relationships. Also, since value domains are not part of the WSDL schema, it remains up to the tester to enumerate representative values and to define value ranges in the assertion script. This can result in a significant manual effort. Assertion script templates can be generated, but the filling out of the actual test data has to be done manually. One way to alleviate this problem would be to include domain definitions in the WSDL. Another way would be to store the data values in external tables, which are referenced by the assertions. These and other issues will have to be addressed in future work.

REFERENCES

1. Kratzig D, Banke K, Slama D. *Enterprise SOA Coad Series*. Prentice-Hall: Upper Saddle River NJ, 2004; 6.
2. Juric M. *Business Process Execution Language for Web Services*. Packt Publishing: Birmingham, U.K., 2004; 7.

3. Sneed H. Integrating legacy Software into a Service oriented Architecture. *Proceedings of the 10th European Conference on Software Maintenance and Reengineering CSMR 2006*, 22–24 March 2006. IEEE Computer Society Press: Silver Spring MD, 2006; 5.
4. Tilley S, Gerdes J, Hamilton T, Huang S, Müller H, Smith D, Wong K. On the business value and technical challenges of adapting web services. *Journal of Software Maintenance and Evolution*: *Research and Practice* 2004; **16**(1–2):31–50.
5. Perry D, Kaiser G. Adequate testing and object-oriented programming. *Journal of Object-Oriented Programming* 1990; **2**(5):13–19.
6. Musa J, Ackerman AF. Quantifying software validation: when to stop testing? *IEEE Software* 1989; **6**(3):19–27.
7. Berg H, Boebert W, Franta W, Moher T. *Formal Methods of Program Verification and Specification*. Prentice-Hall: Englewood Cliffs, NJ, 1982.
8. Tsai WT, Paul R, Song W, Cao Z. Coyote: An XML-based framework for Web services testing. *Proceedings 7th IEEE International Symposium on High Assurance Systems Engineering*, HASE 2002, 25–26 October 2002. IEEE Computer Society: Silver Spring MD, 2002; 173–174.
9. Tsai WT, Paul R, Wang Y, Fan C, Wang D. Extending WDL to facilitate Web services testing. *Proceedings 7th IEEE International Symposium on High Assurance Systems Engineering*, HASE 2002, 25–26 October 2002. IEEE Computer Society: Silver Spring MD, 2002; 171.
10. Narayanan S, Mcllraith S. Simulation, verification and automated composition of Web services. *Proceedings of the 11th International Conference on World Wide Web*, WWW'02, 2002. ACM Press: New York NY, 2002.
11. Paolucci M, Kawamura T, Payne TR, Sycara K. Semantic matching of Web services capabilities. *Proceedings of the First International Semantic Web Conference*, ISWC 2002, 9–12 June 2002 (*Lecture Notes in Computer Science*, vol. 2342/2002). Springer: Berlin/Heidelberg, 2002.
12. Nguyen HQ. Web application testing beyond tactics. *Proceedings 6th IEEE International Workshop on Web Site Evolution*, WSE 2004, 11 September 2004. IEEE CS Press: Los Alamitos CA, 2004.
13. Jiang Y, Stroulia E. Towards reengineering Web sites to Web-services providers. *Proceedings of the 8th European Conference on Software Maintenance and Reengineering*, CSMR 2004, 24–26 March 2004. IEEE Computer Society Press: Silver Spring MD, 296–305.
14. Martin R. The test bus imperative: Architectures that support automated acceptance testing. *IEEE Software* 2005; **22**(4):65–76.
15. Editor. Mercury Interactive simplifies functional testing. *Computer Weekly* 2006; **15** (April): 22.
16. Editor. Parasoft supports Web service testing. *Computer Weekly* 2006; **15** (April): 24.
17. Empirix Inc. e-Test suite for integrated Web testing. www.empirix.com [21 July 2007].
18. Howden W. *Functional Program Testing and Analysis McGraw-Hill Series in Software Engineering and Technology*. McGraw-Hill College: New York NY, 1987; 123.
19. Bradley N. *The XML Companion* (3rd edn). Addison-Wesley: Reading MA, 2001.
20. Sneed H. Reverse engineering of test cases for selective regression testing. *Proceedings of the 8th European Conference on Software Maintenance and Reengineering*, 24–26 March 2004. IEEE Computer Society Press: Silver Spring MD, 2004.
21. Sneed H. Testing an eGovernment Website. *Proceedings 7th IEEE International Symposium on Web Site Evolution*, WS E2005, 26 September 2005. IEEE Computer Society Press: Silver Spring MD, 2005.
22. Ramamoorthy C, Ho S. Testing large software with automated software evaluation systems. *IEEE Transactions of Software Engineering* 1975; **1**:46–58.
23. Nguyen HQ. Testing Web-based applications. *Software Testing & Quality Engineering* 2000; **2**(3):23–30.

AUTHORS' BIOGRAPHIES

Harry M. Sneed is a test engineer in Anecon GmbH in Vienna and is a lecturer at the University of Linz in Austria, teaching Software Evolution. He has tremendous industrial experience in software testing and software maintenance and evolution. He has been working as Programmer/Analysis, Project Leader and Research Consultant in various large-scale industrial projects, including U.S. Navy, Volkswagen Foundation, Hungary, Union Bank of Switzerland, and three ESPRIT projects. He has published more than 180 technical articles both in English and German Science journals and proceedings, and 17 books in German. He received the best Industrial Paper award in ICSM 2002 in Montreal. He was the General Chair of ICSM 2005 in Budapest. He is Co-Chair of the GI-Fachgruppe for software Management and serves in the GI committee for Metrics, Test and Reengineering. In 1998, he was elected to the technical Committee of the IEEE Computer Society, and in 2005 he was appointed as a fellow of the German Informatics Society.

Shihong Huang is an Assistant Professor in the Department of Computer Science & Engineering at Florida Atlantic University. She has a PhD from the University of California, Riverside. Her research interests include reverse engineering, program comprehension, software systems redocumentation, and software maintenance and evolution. She was the General Chair of the 24th ACM International Conference on Design of Communication (SIGDOC 2006), and is Program Co-Chair of the 9th IEEE International Symposium on Web Site Evolution (WSE 2007).