Towards a Practical and Effective Method for Web Services Test Case Generation

Zhong Jie Li, Jun Zhu
IBM China Research Lab
{lizhongj,zhujun}@cn.ibm.com

Liang-Jie Zhang
IBM Watson Research Center
zhanglj@us.ibm.com

Naomi M. Mitsumori
IBM Global Business Services
naomi@us.ibm.com

Abstract

This paper proposes a method for Web services test case generation, which is centered on a practical test data generation framework that has higher probability to penetrate the service implementation logic. This presented framework leverages both the information contained in the WSDL/XSD files and the information provided by testers as well as enables testers to customize the fields to be tested, field constraints and data generation rules. The proposed method can generate test cases that are effective in detecting defects and efficient in reducing test time. This paper also presents preliminary empirical evidence to illustrate the value of the method.

1. Introduction

Web services has been used to realize service-oriented architecture (SOA) as foundational realization technology. As many enterprises are now deploying numerous Web services, and services ecosystems grow increasingly complex connecting many parties, the importance of quality of Web services is inarguable. It is no surprise that Web services testing is the first thing that comes to many people's mind when SOA testing is mentioned.

Nowadays, there are quite a few commercial products [1-5] and academic works [6,7] on Web services testing, including test data generation. Basically these tools provide graphical user interfaces to allow users import WSDL definitions, create and edit test cases manually, and run the test cases and check results. Some tools can analyze the data format information contained in the WSDL file, generate test data variations and produce a lot of test cases. The leveraged data format information includes data type, value and structure. Variations can be generated based on equivalence partition, boundary value analysis and so on.

These methods have one major deficiency when applied in practice as we observed before developing the proposed approach in this paper. They only use the information contained in the WSDL files. In reality, however, a WSDL definition usually does not have much information about the real type and value space of a field. This is either because the Web services designer has limited time to explore different types of XML schema, or has intentionally left it flexible for future data type change. For example, the parameters of an **add** operation are all of **string** type, which gives no clue for a test generation algorithm to search in the numeric type value space. It can only generate random strings based on this type definition. These test cases are ineffective in detecting defects as most of the inputs are filtered by a programming API (e.g. a regular expression) before real business logic begins in the service implementation. We are not saying that test data of **string** type should not be used for **int** field - that can be useful – our suggestion is that they should not occupy majority of the test data as one string value is already a good equivalence partition. Too many values in the same equivalence partition just mean waste of test run time without increasing defect discovery rate.

In this paper, we propose a method that overcomes this deficiency and can generate effective test data for Web services functional testing. In addition to using the original information in the WSDL specification, we also allow users to customize several things: the field to be tested, the actual data type, the additional value constraint, data variation rules, etc. The method can generate a smaller size of test data variations. Irrelevant or redundant test data can be effectively filtered. Reduced test data size will reduce test execution time. This is important for keeping a rational set of regression test cases. Most importantly, the size reduction doesn't degrade the defect discovery

AST'09, May 18–19, 2009, Vancouver, Canada
978-1-4244-3711-5/09/$25.00 © 2009 IEEE

capability, as the generated test data are more focused on the variations that are directly related to the service implementation logic.

The paper is organized as follows. Section 2 gives an overview of the method. Section 3 describes the approach in detail. Section 4 presents a preliminary application of the tool. Section 5 introduces related works. Section 6 concludes the paper with future work predictions.

2. Method overview

This paper proposes a method for Web services test data generation. The method is centered around the following idea: leverage both the information contained in the WSDL/XSD files and the information provided by testers, cross-check these two kinds of information for validity, allow for user-customization of tested fields, field constraints and data generation rules, and generate effective test data that have higher probability to penetrate the service implementation logic. The method is shown in Figure 1.

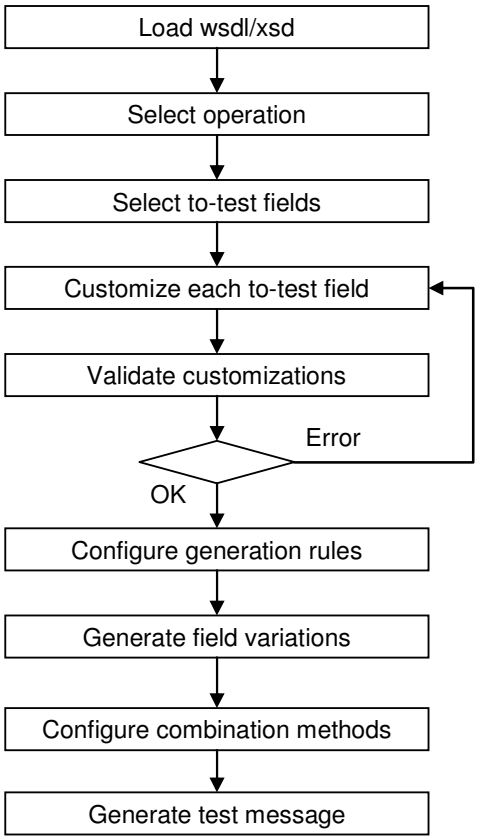

Figure 1. Method overview

The method works as follows. Firstly, the WSDL/XSD files are loaded. All the contained operations will be presented to the user. The user then selects an operation for which he or she wants to generate test data. The input message structure defined for this operation will be shown. It can contain many fields of various types (as supported by the XSD specification), organized in a tree structure. The user selects a set of fields that he or she wants to include into this round of test data generation. The tester can choose all the fields or only some of them, based on Web services requirements and his own knowledge of what is important and what is not. The field selection can also be done (semi-)automatically using code analysis technique to calculate which fields really matter for correct business processing. For example, we analyze the Web services implementation code, build the data flow, and identify how a parameter field is used in the data flow. For instance, it can be used in a condition, an assignment, an invocation to another method or a database operation. These are some examples of usages where programming mistakes can be made and thus deserve testing.

Now the user browses through these selected fields and customizes a field if necessary. In performing the customization, the user can specify a sub-type, which is a further restriction on the base type that is defined in the original WSDL/XSD files. For example, **int** is a subtype of **string**. He or she can modify the original constraint to make it more realistic in reflecting its actual semantics. The tester must input a default value, which will be used as the baseline for data variations. Then he or she can input exclusion special values to indicate that these values are not "special" for this specific field. For example, @ is not a special value for an email field.

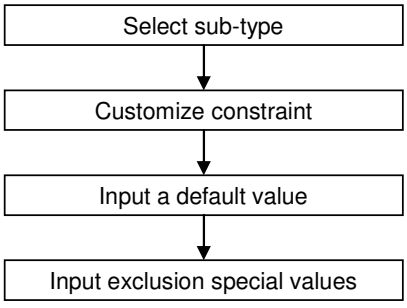

Figure 2. Customization of fields to be tested

After the user has customized all the to-test fields, the final state can be validated in terms of its completeness and correctness. A possible error is that a selected field is not provided with a default value, or a

required field is de-selected. If an error happens, the user needs to fix the error. If the validation passes, the user can then configure generation rules. There can be a list of pre-defined rules, from which the user can select among. One example rule is: for **int** type, generate two negative variations based on the default value, one smaller than the lower bound, the other greater than the upper bound. Next, based on the configured rules, test data can be generated automatically. For each selected field, there will be multiple variations generated. Variations of different fields can be combined in different ways to form a complete input message. The combination method depends on the test strategy.

One possible test strategy is to test each field one by one using different variations of this field and at the same time keeping the other fields to their default values. With this strategy, it is easier to reveal the responsible field when an error occurs during testing. The final message can then be used to actually test the Web services implementation.

The advantages of the proposed method include:

1) The method allows a user to give a subtype and customize field constraint. So the user can influence the test generation and make it generate data variations that are more focused in the right value space. Take the "Add" Web service as an example, the user can specify an **int** subtype based on the **string** type for the two parameters. Then the test generation will focus on **int** value variations. These variations have better chance to penetrate the real business logic of the service implementation. Comparatively, most random **string** values will be filtered at the very entrance of the implementation. It should be noted that the test generation can still generate nonnumeric **string** values but this is not the focus.

2) The method can be used to generate a smaller size of test data variations. By allowing the user to select to-test fields, select sub-type, and modify constraint, the variations will be more focused. Irrelevant message fields can be effectively filtered. Reduced test data size will reduce test execution time. This is very important for keeping a rational set of regression test cases.

3) The generation rules can be configured or added to bring in more intelligence and best practices in automatic test generation. For example, we can add such a rule: generate data variations that are consistent with the "language" field. If the language value is English, the "date" field uses mm-dd-yyyy format. If the language value is Chinese, the "date" field uses yyyy-mm-dd format.

4) Different combination methods can be used to further improve the test coverage. For example, if two fields are related to each other, combination of their different variations can improve the test coverage.

Overall, the proposed method can generate test data that are effective in detecting defects and efficient in reducing test time.

3. Implementation

The proposed method has been implemented in a tool called WSTD-Gen (Web Services Test Data Generator), whose major user interface is shown in Figure 3.

3.1 Use flow

We'll follow the described method in section 2 to explain the GUI elements and workflows.

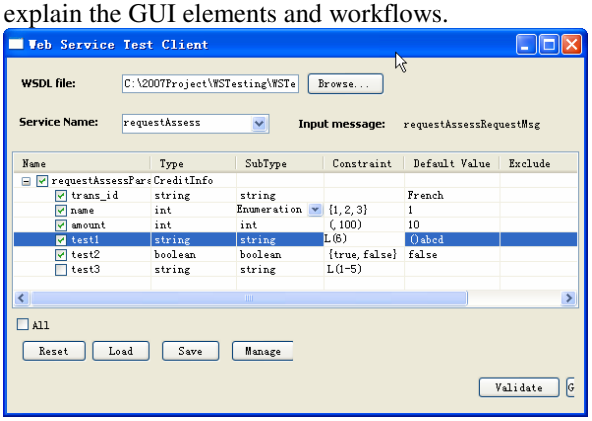

Figure 3. WSTD-Gen screenshot

1) Load WSDL/XSD files

User selects a WSDL file. Both the WSDL file and the associated XSD files will be automatically loaded. Figure 3 shows a Web service and one operation named requestAssess. The input message has a type "requestAssessRequestMsg". It has a tree structure. The following information can be taken from the WSDL/xsd files: field, field name, type, subType, constraint. If there is no restriction defined on a basic type, the subType will be the same as type. An example constraint that can be defined in XSD is the "amount" field with a constraint of (-∞,100). XSD can also specify the occurrence constraint of a field through minOccur and maxOccur attribute. In this example, the field trans_id is a required field (minOccur = 1). It must be filled in the input message. When a WSDL file is loaded, the mandatory fields are all automatically checked in the tree, indicating they must get a value from the user. If user tries to uncheck it, a warning message will show up.

2) Select one operation to test

One Web service can define multiple operations. Users need to select one operation at one time to generate test data for this operation. The example shows an operation by default once the WSDL file is loaded.

3) Select to-test fields

As mentioned earlier, the mandatory fields as specified in the WSDL/XSD files have been selected on WSDL loading automatically. For other fields that are not required (minOccur = 0), the user needs to select them if he or she wants to include them in the test message. The selection criteria can be varied: fields that matters for business process decisions (conditions and assignments) in the Web services implementation, fields that will be saved into databases, and others. Such fields can be (semi)automatically selected in some ways given a specific criterion. One method is to use code analysis technique to track parameter usage inside the Web services implementation. WSTD-Gen has put this feature in the requirement list for future enhancement.

4) Customize each to-test field

a) Select a sub-type.

As we mentioned, many WSDL/XSD definitions specify a ***string*** type for most fields, although XSD has a lot of predefined types: ***string***, ***boolean***, ***date***, ***dateTime***, ***time***, ***integer***, ***long***, ***int***, ***short***, ***byte***, ***decimal***, ***float***, ***double***, etc. There can be multiple reasons for such a practice. One reason is that this allows easier changes of Web services implementation. A ***string*** type can safely hold values of various types.

The Web services implementation will use some filtering logic to guard against invalid values. The drawback is that the Web services provider must bear the burden of data validation in the business logic. This is also not good for testers and automatic test generation tools as test data variations can only be made based on the ***string*** type. If we know it's in fact an ***int*** type, Max value, Min value can be tested. It must be tested that a loan Web service will respond with the right message if the loan request amount exceeds a required value in the specification. A hundred of random values are no better than one pinpointing value.

b) Customize constraint.

Table 1 shows the type constraints supported by XSD.

Table 1. XSD constraint specification

Constraint	Description
enumeration	Defines a list of acceptable values
fractionDigits	Specifies the maximum number of decimal places allowed. Must be equal to or greater than zero
Length	Specifies the exact number of characters or list items allowed. Must be equal to or greater than zero
maxExclusive	Specifies the upper bounds for numeric values (the value must be less than this value)
maxInclusive	Specifies the upper bounds for numeric values (the value must be less than or equal to this value)
maxLength	Specifies the maximum number of characters or list items allowed. Must be equal to or greater than zero
minExclusive	Specifies the lower bounds for numeric values (the value must be greater than this value)
minInclusive	Specifies the lower bounds for numeric values (the value must be greater than or equal to this value)
minLength	Specifies the minimum number of characters or list items allowed. Must be equal to or greater than zero
Pattern	Defines the exact sequence of characters that are acceptable
totalDigits	Specifies the exact number of digits allowed. Must be greater than zero
whiteSpace	Specifies how whitespace (line feeds, tabs, spaces, and carriage returns) is handled

If specified in the original WSDL/XSD files, they can be used directly in test data generation. Users can still make changes if only the modification is compatible with the original constraints. If not specified at all in the original definitions, users can input their constraints based on the later-provided subtype using the same set of constraint systems.

c) Input a default value.

Default value is the baseline of variations. It needs to be provided for each selected to-test fields. An input message composed of the selected fields with default values is the baseline test message. Typically this is the message used in a basic positive test case. With this test case, testers can check that the service functions correctly and responds with a correct output message. Usually, this message cannot be generated automatically as it is well-formed and has clear

business meaning. Most importantly, it has dependency on basic test data. For example, to test a **findBook** operation, an **addBook** operation must be invoked and the book information used in **findBook** must be the same as that in the **addBook**.As a convenience method, the default values can be filled in automatically by referring to existing real test message in an XML file or some other sources.

d) Input exclusion special values.

Inserting special characters is a common strategy to generate variations. For example, for a ***string*** type field, special characters like ~!@# can be used together with alphabetic characters as variations. However, whether or not a character is "special" depends on the field semantics. For example, @ is normal in an email address. So we need to indicate this exclusion before test generation.

5) Validate customizations

The user customizations need to be validated against the original constraints in the WSDL/XSD definition so that user-introduced error could be avoided. Here are two example validation items: a) each selected field must have a default value, b) each modification of constraint must be compliant with the original one. If there are errors, users are alerted to fix them and run validation again.

6) Configure generation rules

Generation rules are used by the tool to generate test data variations. Figure 4 shows the configuration GUI. The column headers tell the column usage. Generation rules can be different for different data types, so they are organized by data types - each row specifies all kinds of test data variation generation rules for a specific data type (group): ***String***, ***Boolean***, ***Number***, ***Date***, ***Time***, ***DateTime***, etc, as listed in the first column of this table.

The first column is "SubType" instead of "Type", because the "SubType" represents the actual data type of a field, which is provided by the user in "Customize each to-test field".

The second column is "Constraint" of the "SubType".

The third column is an example "Default" value.

Each of the other columns represents a category of variation generation rules. In each category, related variation generation rules are listed.

The first row of the generation rule table illustrates an example generation rule for the ***String*** subtype. It lists example values for different categories or attributes associated with the ***String*** subtype. Each value represents a generation rule. The default value is "abcd". The "Length" category includes a variation with a length smaller than the minLength, and another variation with a length over the maxLength. MinLength and maxLength are the allowed minimum and maximum string length of this variable, respectively, as defined in the constraint or pattern of this variable in the respective WSDL or XSD file. In this case, the constraint is L(3-5), meaning that the minimum string length is 3 and the maximum is 5. So one variation value is "ab", the other is "abcdef". Note that the Constraint "L(3-5)" and Default value "abcd" are not tied to any particular field of the test message. They are used only to illustrate the variation generation rules to users by examples. A particular field may have a default value "xyzw" and variations "xyz" and "xyzwuv".

The "Special char" category includes all the variations derived by replacing an original letter with a special character. The example variation shows one that replaces the character 'a' in the default value "abcd" with another character '~', resulting in a value "~bcd".

The "Wrong type" category includes all the variations that have a different type: integer, float, etc.

The "Case" category includes case variations: changes between lower case and upper case.

The "Enumeration" category includes all the enumeration values that are provided in the subType.

The "Wrong value" category includes all the variations with the correct type but incorrect value. If an enumeration subtype is defined, a value not in the enumeration value set is invalid.

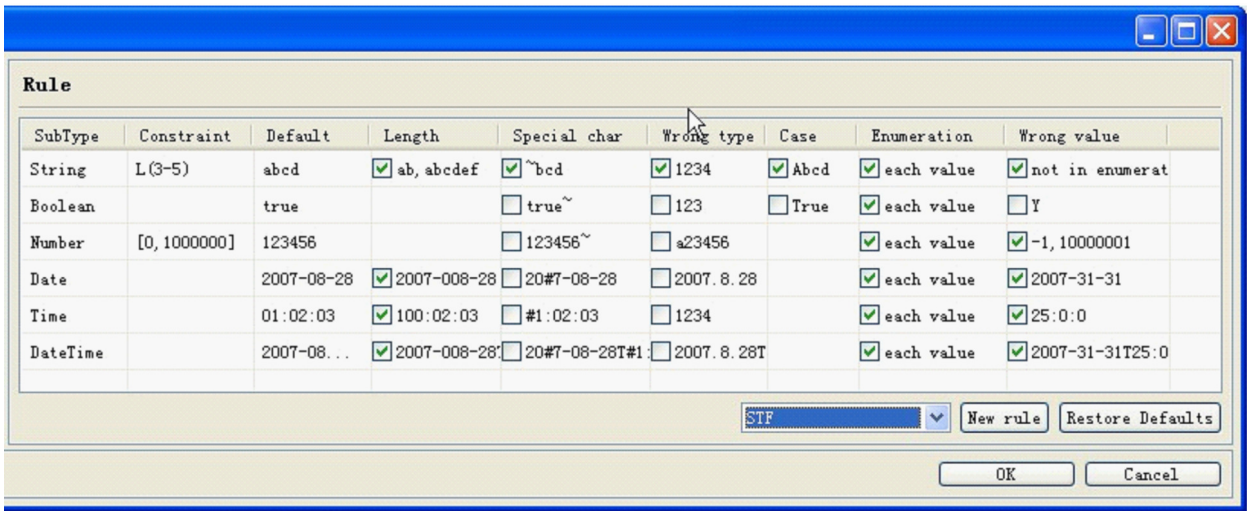

Figure 4. Test generation rules

The rule representation shown in Figure 4 is easier to understand than a logical expression coded in a rule specification language like stringLength(variable)<minLength. At the backend, the rules are specified more formally, but this is only for advanced users, not the end users. In the GUI shown in Figure 4, end users can view, select, unselect defined rules but cannot add new rules.

In addition to the examples listed in Figure 4 there can be other rules for the ***String*** data type. There can be rules for the other data types. There can be rules for associating different fields together, rules for calculating one field from some other fields. Users are provided with the flexibility to select among predefined rules to apply to the user's specific test purpose. These are not supported in current release of the tool and will represent future works.

7) Generate field variations

Based on the user customizations and generation rules, field variations can be generated. For example, "test1" is a field of ***string*** type whose constraint is "L(6)", default value is "()abcd". The generated variations are shown in the following XML segment in Figure 5.

```
- <test1>
    <default>()abcd</default>
    <specialChar>~)abcd,!)abcd,@)abcd,#)abcd,$)abcd,%)abcd,^)abcd,*)abcd,()abc
      abcd,{)abcd,})abcd,[)abcd,])abcd,>)abcd,<)abcd, )abcd</specialChar>
    <wrongType>466272,4.66272,6)abcd</wrongType>
    <length>()abc</length>
    <case>()abcd,()ABCD</case>
  </test1>
```

Figure 5. Test data variations for an example field

8) Configure combination methods

For each selected field, there will be multiple variations generated. Variations of different fields can be combined in various ways to generate a complete input message. The combination method depends on the test strategy. One possible test strategy is to test each field one by one using different variations of this field and at the same time keeping the other fields to their default values. With this strategy, it is easier to reveal the responsible field when an error occurs during testing. Another method is an exhaustive combination of variations of two fields to achieve a so-called pair-wise coverage. This kind of coverage is especially useful for inter-dependent fields. For example, customerType and amount can jointly determine the discount rate. Therefore, it may be preferred that some variations for customerType {gold, silver, bronze} should be combined with some variations for amount {1000, 10000, 100000}.

9) Generate test messages

Once the combination method is determined, the full messages can be generated. The complete messages can then be used to test the Web services implementation. The following XML segment in Figure 6 shows a complete message example under the first kind of combination method. The "test1" field has a special character variation; the other fields are all filled default values.

```
<?xml version="1.0" encoding="UTF-8" ?>
- <requestAssessRequestMsg>
  - <requestAssessParameters>
      <trans_id>French</trans_id>
      <name>1</name>
      <amount>10</amount>
      <test1>~)abcd</test1>
      <test2>false</test2>
    </requestAssessParameters>
  </requestAssessRequestMsg>
```

Figure 6. An example test message

3.2 Tool architecture

Figure 7 is the architecture of WSTD-Gen. The components are explained below.

1) GUI. This component is responsible for user interaction. Users read the message structure, make customizations, configure generation rules, and perform other activities related to test data generation.

2) WSDL/XSD parser. This component parses WSDL/XSD/XML files. The parsed result as memory models are used by GUI component to present to users. Existing XML files holding real Web services messages are also parsed and put into the "Existing real data" pool for later use as default value in test data generation. "Business semantics" contains semantic information related to the Web services definition. Some of the additional field constraints can be extracted from such information. Furthermore, relations between fields and service invocation preconditions can also be found and used in test data generation. The usage of such information for test data generation is covered in the generation rule configuration step. So far in the current release of WSTD-Gen, advanced use of business semantic data has not been supported. Currently we only support simple use of business data like that provided in the XML Schema Standard Type Library: http://www.codesynthesis.com/projects/xsstl/.

3) Configuration validator. User customizations and generation rule selections are validated by this component. Possible errors alert the user. The final correct result (configurations, generation rules) are saved.

4) Variation generator. This component takes the above artifacts as inputs, and produces data variations for each selected to-test fields.

5) Message composer. Based on the variations and the chosen combination method, this component composes the complete input messages, which are then put into a message pool for the usage of test tools.

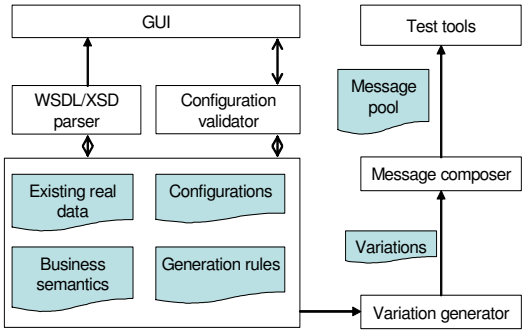

Figure 7. Tool architecture

4. Preliminary application

We applied the prototype WSTD-Gen in a real Web service that wraps a call center query function. Table 2 lists some key facts of one operation of the web service.

Table 2. Test generation for a web service

Total fields of the input message	128
Fields selected to test	23
Primitive data types in WSDL/XSD specification	2 (string, any)
Sub-types inputted by tester	8 (int, datetime, five kinds of string sub-type with length restriction, enumeration on the previous data type)
Generation rules applied	All the predefined rules for string, number, and datetime
Test cases auto-generated	514
Test generation time	10 minutes
Test cases manual designed	293
Manual design time	3 Days

This operation has a big input message with 128 fields. If we use complete test data generation that includes all these fields, the resulting tests will be unmanageably huge. Given the pushing service testing schedule, the tester (together with the Web services developer) only selected 23 fields to focus test on. These fields represent the data heavily manipulated in the business logic.

The tester also customized the data type. Originally there are only two data types, the only evidence of the real data type is the field name – but it can be misleading. In order to get some clues on how to write test data, the tester asked for the help from the architect and the developer, and then determined eight sub-types.

With this information, they can generate effective test data. The tester decided to use all the test generation rules built in the tool, and finally generated 514 test cases within 10 minutes. Before using the tool, the tester had identified 293 test cases manually within 3 days. The tester was happy with the tool and asked for more functions.

This Web service has 12 operations. There are over 20 such services in a single one project. Thus the estimated test case number for manual writing is: 300*12*20+ (72,000+). This is a formidable number. WSTD-Gen can be a valuable tool for both reducing human labor and improving test coverage.

5. Related works

Currently there are quite a lot of tools with Web services testing capability in market or open source community. To only list a few, Parasoft SOATest, Mindreef SOAPScope, CrossCheck Networks SOAPSonar, SOAPUI, and IBM Rational Tester for SOA Quality (RTSQ) [1-5].

Basically these tools provide graphical user interfaces that allow users import WSDL definition, create and edit test cases, run the test cases and see testing result. A test case consists of an input message that will be used as the parameter to invoke the service under test, and an output message that is expected from the service. The actual output message is compared to the expected one. Based on the comparison, a verdict is made on whether the service function is correct or not.

For most of these tools, there is no automatic test message or data generation capability. This will lead to time-consuming test case design process. Consider a simple **add** function that takes two parameters *x* and *y*. To adequately test this function, many variations on the parameters need to be tried: *x*=1, *y*=1; *x*=-1, *y*=1; *x*=0, *y*=1; *x*=-1, *y*=-1; *x*=a maximum value for the ***int*** type, *y*=1; *x*=non-integer value, *y*=1;…. This is only a simple case that involves only two parameters. Usually, a real Web service operation uses large messages with complex structures. Tens or hundreds of fields in a single message is common. It can be imagined how much effort it will be for manual test case design.

Some of them can generate test cases automatically from the WSDL definition. SOAPSonar utilizes XSD-Mutation, a patent-pending automation technique, to generate a set of test cases, both positive and negative. The test mutations may occur at the data type, data value, message structure, or protocol binding level. An example mutation is a buffer overflow type boundary conditions for an unbounded ***string*** type field. The test cases generated by SOAPSonar are mostly random test cases and ineffective in detecting defects as it only refers to the data structure and type information defined in the WSDL. In reality, a WSDL definition usually has no much information about the actual type and value space of a field. For example, the parameters of an **add** operation can be all of ***string*** type, without any clue for a test generation algorithm to search in the numeric type value space. It can only generate random strings based on this type definition. These test cases are ineffective in detecting defects as most of the inputs are filtered in the service implementation before real business logic begins.

RTSQ can generate a random test message from a WSDL definition, e.g. “abc” for each ***string*** type field. The primary usage of such test cases is performance testing, where message values are not important. It is inappropriate for the functional testing of web services.

In the object-oriented world, there are tools that can automatically generate test cases from method declarations. For example, Parasoft Jtest [8] can do Java class unit test generation – test each method with various boundary values of each input parameter (boundary values for ***string*** type: null, empty, special characters, etc), and observe whether runtime exception occurs. It is not applicable to Web services test generation because 1) there is the same type problem of WSDL definition as mentioned earlier 2) further schema-compliance is needed: the input message must comply with its schema definition. WSDL and XSD standards allow value space restriction on specific types. For example, for the ***int*** type, max and min value can be specified; for ***string*** type, max and min length can be specified. Such information cannot be defined in Java method declaration, thus not considered in any test generation method.

[6] reported a method of Web services test case generation by analyzing the message data types according to standard XML schema syntax. A knowledge base is established where each build-in simple data type is associated with default facets definition and sets of candidate values based on the test strategies such as random values and boundary values. Once the WSDL file is parsed, the analyzer will extract the data, create instances of their corresponding data type, obtain and record the facet values with the data instances. Each facet is linked to a facet generator, for examples, for a data of type ***string***, lenGen() to generate the length of a string. Test generator coordinates the facets generators and composes the result data. Again, this method only refers to the data structure and type information defined in the WSDL. Therefore, the generated test cases are mostly random test cases and ineffective in detecting defects.

[7] reported a method of using data value perturbation (DVP) to generate a variety of test cases. Data value perturbation relies on ideas from boundary value testing. Tests are created by replacing each value with each boundary value, in turn, for the appropriate type. This is similar with the approach described in [6].

6. Conclusion and future works

In this paper, we propose a Web services test case generation method. It is practical in that it doesn't aim fully automated test generation from the WSDL file and gives appropriate control to the user. It is effective in that it leverages additional inputs besides the WSDL information; it allows users to customize the data type and value with programming and domain knowledge; it allows users to select test generation rules (by data type) according to the test strategy within their organization. All this results in much fewer, more focused test cases that have a bigger chance to uncover defects.

A tool named WSTD-Gen has been built. With the tool's help, Web services developers can generate and maintain a set of regression test cases with little effort and ensured coverage. WSTD-Gen can be interfaced with any test execution tool given some mechanism to adapt different message formats.

In the future, we plan to enhance the method and tool in the following aspects: automatic code analysis to decide important fields to test, advanced rules (e.g. cross-field generation rule), use of business semantics information, etc. In addition, how to leverage existing Web services design or implementation to decide the real type and constraint is another interesting topic. With such capabilities, users' effort in customization will be reduced.

7. References

[1] SOATest. http://www.parasoft.com/soatest

[2] SOAPScope. http://home.mindreef.com/

[3] SOAPSonar. http://www.crosschecknet.com/

[4] SOAPUI. http://www.soapui.org/

[5] IBM Rational Tester for SOA Quality (RTSQ). http://www.ibm.com/developerworks/rational/library/07/0327_kelly/index.html

[6] Xiaoying Bai, Wenli Dong, Wei-Tek Tsai, Yinong Chen. WSDL-based automatic test case generation for Web services testing. IEEE International Workshop on Service-Oriented System Engineering (SOSE) 2005. 207-212.

[7] J. Offutt and W. Xu. Generating Test Cases for Web Services Using Data Perturbation, ACM SIGSOFT Software Engineering Notes 29, No. 5, 1–10 (2004).

[8] JTest – Java unit testing & code compliance. http://www.parasoft.com/jtest.