

Swiss Cheese Test Case Generation for Web Services Testing

W. T. Tsai, X. Wei, Y. Chen, R. Paul*, B. Xiao

Computer Science and Engineering Department

Arizona State University, Tempe, AZ 85287-8809, U.S.A

*Department of Defense, Washington DC, U.S.A

Abstract

Current Web services testing techniques are unable to assure the desired level of trustworthiness, which presents a barrier to WS applications in mission and business critical environments. This paper presents a framework that assures the trustworthiness of Web services. New assurance techniques are developed within the framework, including specification verification via completeness and consistency checking, test case generation, and automated Web services testing. Traditional test case generation methods only generate positive test cases that verify the functionality of software. The proposed Swiss Cheese test case generation method is designed to generate both positive and negative test cases that also reveal the vulnerability of Web services. This integrated development process is implemented in a case study. The experimental evaluation demonstrates the effectiveness of this approach. It also reveals that the Swiss Cheese negative testing detects even more faults than positive testing and thus significantly reduces the vulnerability of Web services.

Keywords: Web services, Web services testing, test case generation, vulnerability, model checking.

1. Introduction

Web services (WS) have emerged in e-commerce and e-business applications in the recent years. Current WS are based on UDDI server that provides directory services similar to the telephone yellow book. A UDDI server is not responsible for the quality of the services it refers. Thus, the trustworthiness or vulnerability of WS presents a major concern for the users and a barrier to widening the application of WS. Traditional dependability techniques, such as correctness proof, fault-tolerant computing, testing, and evaluation, could be used to improve the trustworthiness of WS. However, these techniques have to be redesigned to handle the dynamic and the open platform features of WS. This paper exploits WS-based testing and verification techniques that can lead to trustworthy WS.

The main differences between WS and traditional applications are caused by the following four aspects: Loosely coupled, trustworthiness, lack of specification, and Dynamic integration.

A number of studies and attempts have been made to address the WS testing and verification problems. In [5] Davidson distinguished between two kinds of WS testing: Internet-based and Intranet-based WS testing. He also listed several testing techniques for WS including: proof-of-concept testing, functional testing, regression testing, load/stress testing, and monitoring. In [14], it was suggested that WS testing should include: basic WS functionality; SOAP messages; WSDL files; publishing, finding, binding capabilities of an SOA; asynchronous capabilities of WS; the SOAP intermediary capability; the quality of service of WS; dynamic runtime capabilities; SOAP and WS interoperability; and WS performance and load testing. In [2], Clune and Chen suggested that both clients and service providers must be involved in WS testing, and many issues must be addressed during WS development including: security, interoperability, UDDI registration, and performance considerations. Similarly, in [8], Myerson stated that all three parties of WS including clients, providers, and brokers, need to be involved in WS testing. These discussions support the collaborative testing concept we propose for WS testing.

Commercial tools have been developed for WS testing, including e-test by Empirix at <http://www.empirix.com/>, eValid from Software Research at www.soft.com, xmlpsy at <http://webservices.xmlpsy.com/>, and SOAPtest from Parasoft. These tools often debug XML files, monitor WS testing, capture-and-replay tests and debug SOAP. IBM's Websight is a testing and debugging tool under its WS framework WebSphere [1]. Websight traces and visualizes the execution process of WS and thus helps the programmer find syntax and semantic errors in the WS under test.

In the past a few years, we have developed WS testing techniques that support trustworthy computing. In [13], rapid testing techniques were developed, including group testing, regression testing, pattern-based verification. In [10], an enhanced WSDL interface was

developed to include dependency information, functional description, invoking, and concurrent sequence specifications so that test cases can be generated based on WSDL descriptions. In [11], a variety of test generation techniques were developed to test WS in an enhanced UDDI-based service broker. These test cases could be arranged hierarchically to test domains of related WS. An early form of WS check-in and check-out processes was also proposed to increase the confidence of WS used. In [12], WS scenarios were developed in stages. In the first stage, the individual service scenarios are developed; in the second stage, interactions among WS were modeled; and finally the overall scenarios were developed combining previously developed scenarios. These scenarios were then translated into test cases to be performed by organized distributed agents.

Based on these studies, we proposed the WebStar (Web Services Testing, Reliability Assessment, and Ranking) framework that assures the trustworthiness and reduces the vulnerability of WS by rigorous positive and negative testing, reliability assessing, and ranking. A WS-based reliability model was proposed in [15]. Ranking of WS and test cases were studied in [16].

This paper focuses on test case generation and testing techniques for WS assurance. These techniques not only perform positive testing that verify the required functionality but also perform negative testing that assures the robustness of WS under unspecified inputs or attacks. Negative testing is particular important for WS, because WS can be composed of WS from different service providers without the availability of the source codes.

The rest of the paper is structured as follows. Section 2 outlines the overall WS development process including specification checking, test case generation, and testing techniques. Section 3 introduces a topology-based Web Service test case selection method: the Swiss Cheese Model. Section 4 presents a case study and experiments that follow the proposed development process. Finally, section 5 concludes the paper.

2. Development and Assurance of Web Services

OWL-S language facilitates WS specification with a core set of markup language constructs for describing the properties and capabilities of WS in an unambiguous and computer-interpretable form. OWL-S supports the automation of WS tasks including automated WS discovery, invocation, interoperation, composition, and execution monitoring.

The first step of our development process is to create WS specification written in OWL-S. With a translator, the specifications written in other specification languages

such as WSDL and WS-CDL could be used too. The next step is to perform specification check. Different methods can be applied to perform the specification check. We have developed the Completeness and Consistency (C&C) analysis [13] and model checking techniques based on BLAST [1][5]. In this paper, we will focus on the C&C analysis. If the specification fails the check, it has to be modified, refined and tested again.

After several iterations of test and refinement, we obtain the enhanced specifications, which are used for test case generation. Boolean expression analysis method is used to extract the full scenario coverage of Boolean expressions [13], which are then applied as the input to the Swiss Cheese Automated Test Case Generation [17] tool, which, in turn, generates both positive and negative test cases.

Finally, the test cases are stored in the test case database before being applied to test the WS developed.

The key technologies applied in this development process include:

- Completeness and consistency (C&C) analysis of the WS specification: This module checks whether all conditions in OWL-S specification are consistent and whether all conditions have been covered and handled properly by the specification.
- Verification patterns: Testing temporal logic properties can be facilitated by the verification patterns technique [14]. This approach can generate many test cases by recognizing patterns in system behavior and generate the corresponding test cases by composition. This approach has been used successfully to test medical devices and DoD applications in our previous projects.
- Boolean Expression Extraction for Test case generation: Test case generation techniques can be greatly enhanced by this comprehensive formal C&C checker followed by test case generation based on Boolean expressions [13]. An important distinction of this approach is that test case generation is based on quantitative Hamming distance. All previous approaches, including MC/DC and MUMCUT [3], were based on user experience and intuition. Exploring the topological hypercube structure of Boolean expressions can easily reveal the faults not discoverable by previous approaches. Furthermore, these two mechanisms can be completely automated and thus saving significant effort and time.
- After Boolean expressions are extracted, the Swiss Cheese test case generation tool can be applied to obtain both positive and negative test cases, as to be discussed in the next section. Swiss Cheese approach is useful for Web service testing because it solves three traditional difficulties in Web service testing:

lack of code, dynamic integration and high insurance requirements. As a specification-based black box testing approach, Swiss Cheese approach can general and select test cases with Web service specifications only, which are available for most of the commercialized Web services. The dynamic integration of Web services requires test cases to be rapidly generated because the integrated Web service may be recomposed very fast. With the automated test case generation and selection tool based on the Swiss Cheese model, the generation process for a mid-size Web service can be finished in only a few seconds. What's more, addressing both positive and negative testing enables Swiss Cheese approach to provide high insurance that is necessary for commercialized safety-critical Web services like bank or air plane ticket reserving services.

3. Topology Based Web Service Test Case Selection Method--- Swiss Cheese Model

Swiss Cheese model presents a systematic approach to generate and select the most powerful test cases that can detect faults in WS more effectively. Before discussing the, we introduce the notations and terminology used in the model first.

3.1 Notations and Terminology

Let $C = \{c_j\}$ ($j = 1, \dots, n$) be a set of *condition variables* of the system under consideration, which represents the system status or states; T_i be a *term* representing a product of condition variables:

$$T_i = c_{i_1} \bar{c}_{i_2} \dots \bar{c}_{i_j} \dots c_{i_k} \quad (i=1, \dots, m)$$

where c_j can take two values, "true" and "false". The condition variables that do not appear in a term are regarded as "don't care", i.e., they do not affect the truth value of the term. The \bar{c}_j represents the negation of condition variable c_j . Condition variables are referred as *literals*, following the same convention used in MUMCUT [3][18][19].

A term is said to be a *min-term* if it contains every conditional variable or its negation, that is, each variable takes an explicit appearance of either "1" or "0". For example, if there are three condition variables in the Boolean expression, the term $c_1 \bar{c}_3$ is not a min-term because variable c_2 is missing. This term can be split into two min-terms: $c_1 c_2 \bar{c}_3$ and $c_1 \bar{c}_2 \bar{c}_3$. Each min-term represents a cell in Karnaugh map (K-map).

A *Boolean expression specification* S can be represented as the sum of terms: $S = \sum_{i=1}^m T_i$.

3.2 Swiss Cheese Model

The Swiss Cheese (SC) Model defines the test case generation process as follows:

1. Take the OWL-S specification as input;
2. Convert the OWL-S into the scenarios consisting of sequences of conditions and the relationship among the conditions;
3. Completeness and Consistency (C&C) check;
4. Extract a Boolean expression from each scenario;
5. Represent a Boolean expression by a standard K-map;
6. Compute the Hamming Distance (HD) of each cell in the K-map;
7. Compute the Boundary count (BC) of each cell in the K-map; Define Detection Distance $DD = (HD, BC)$
8. Create a Swiss Cheese map (SC-map) by (1) using a color code to represent the truth values of the cells. Non-white color for true and white color for false; (2) tagging Detection Distance DD of each cell ;
9. Rank the test cases: Each cell corresponds to a test case and the DD of the potency of the test case. The dictionary order among the DD values ranks the test cases.

In this section, we will focus the most innovative part of the approach, e.g., steps 6 through 9. In the next section, we will use a case study to show the complete process including all steps.

This approach is called SC Model because the appearance of SC-map looks like a piece of bubbled Swiss cheese. SC Model selects most critical test cases according to two criteria: *Hamming distance* (HD) and *Boundary count* (BC). Hamming distance, named after R.W. Hamming, is usually defined as the number of different bits between two strings. HD of two binary numbers or two min-terms in our study is defined as the number of bits between the two terms in which the two differ. For example, the HD between two min-terms "001111" and "010011" is three. In a SC-map, each cell corresponds to a min-term. Therefore HD can be extended to define the relative distance between two cells. In our model, the HD of a cell is defined as following:

- The HD of a cell C is 0, if the cell is with non-white color and at least one of the neighboring cells has a white color. Cells with $HD = 0$ form the boundary of a Boolean expression.
- For positive test case generation, HD of a cell C is $p+1$, if at least one of the neighboring cells has a HD of p and there is no neighboring cells whose HD is less than p , where $p \geq 0$.
- For negative test case generation, HD of a cell C is $n-1$, if at least one of the neighboring cells has a HD of

n and there is no neighboring cells whose HD is greater than (or absolute value less than) n, where $n \leq 0$. The purpose of defining negative HD is to emphasize the negative testing. If we consider the absolute value, the two cases are identical. Thus, we will imply the positive test generation unless explicitly mentioned.

HD is the primary factor that reflects the fault detection potency of the test case corresponding to the cell. This is because the following two principles:

1. Each cell in the SC-map corresponds to at least one test case.
2. The larger the absolute value of HD is, the lower the fault detection potency of the corresponding test cases will have.

The first principle 1 is straightforward and the second principle can be proved in the following two cases:

Case 1: The principle is true for cells with $HD = 0$ because those cells are all boundary cells. This has been proved in traditional boundary testing theory that boundary cells are most powerful.

Case 2: Let S represents a Boolean expression, represented in the sum of several terms. Each term is the product of literals. From the study of Chen and Lau, [3][18][19], we know that eight types of faults are most typical in testing of Boolean expression. The *literal negation fault* (LNF) is one of those eight typical faults, defined as “A literal in a particular term in the Boolean expression is replaced by its negation”. In other words, it is caused by the reversal of the truth value of a literal in a Boolean expression. Suppose the probability of one literal negation fault is p , the fault detection potency of cells with $HD = 1$ is proportional to p . This is because any cell with $HD = 1$ can be considered as a result of one literal negation fault occurred on one of its neighbor cell with $HD = 0$. For example, Figure 1 shows the SC-map corresponding to the Boolean expression: $\overline{abcd} + \overline{abc}d + \overline{ab}cd + \overline{a}bcd + abcd + abc\overline{d}$. From the figure we can see that 6 cells correspond to positive test cases ($HD = 0$), marked in gray pattern. The remaining 10 cells correspond to negative test cases ($HD < 0$), in white color. The cell \overline{abcd} ($HD = -1$) can be considered as the result of literal negation fault on the $HD = 0$ cell $abcd$ (3rd literal negation) or \overline{abcd} (1st literal negation).

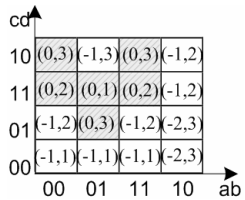


Figure 1. SC-map example with (HD , BC) values

Similarly, the fault detection potency of cells with $HD = 2$ is proportional to p^2 , because any cell with $HD = 2$ can be considered as a result of two different literal negation faults concurrently occurred on one of the cells with $HD = 0$. For example, the cell \overline{abcd} can be considered as the result of two different literal negation faults concurrently occurred on the $HD = 0$ cell $abc\overline{d}$ (2nd and 3rd literal negation) or \overline{abcd} (1st and 3rd literal negation). Because $0 < p \ll 1$, $p^2 \ll p$. Similarly we can draw the conclusion that the fault detection potency of cells with $HD = n$ is proportional to p^n . Because $p^n < p^{n-1} < p$, we prove the conclusion that “The larger the absolute value of HD is, the lower the fault detection potency of the corresponding test cases will have”. This conclusion is also supported by the case study introduced in Section 4.

In a SC-map, a color change means a change of the truth value. This is because in this paper we only use two kinds of color code: gray and white. gray cells mark those cells where the Boolean expression value is true ($HD \geq 0$) and white cells represent otherwise ($HD < 0$).

There is a secondary factor that reflects the fault detection potency: Boundary Count (BC). For a cell with $HD = p$, the BC of this cell is defined as following:

- # of neighbor cells with $HD = p - 1$, if $p \geq 0$
- # of neighbor cells with $HD = p + 1$, if $p < 0$

BC also reflects the fault detection potency because we found that even two cells have the same HD , they may have different fault detection potency if their BC are different. With the same HD , the larger the BC of a cell has, the higher the potency of fault detection is. For example, in Figure 1, the cell \overline{abcd} ($HD = -1$, $BC = 1$) and $\overline{abc}d$ ($HD = -1$ and $BC = 2$) have the same HD , but different BC. The corresponding fault detection potency is also different. The cell $\overline{abc}d$ is more powerful in fault detection because it can detect two different literal negation faults (the 3rd literal negation on the cell $abcd$ and the 1st literal negation on the cell \overline{abcd}), while the cell \overline{abcd} can only detect one literal negation fault (the 4th literal negation on the cell \overline{abcd}).

Compared with HD , BC is a secondary factor because in the research, we found that a cell with smaller HD and smaller BC can usually detect more faults than a cell with larger HD and larger BC. The case study in Section 4 supports this interesting hypothesis.

Because both HD and BC reflect the fault detection potency, we define the term Detection Distance (DD) to represent the pair (HD , BC) which reflects the potency of the fault detection. We defines a series of selection criteria based on the (HD , BC) pair [17][21][22].

Generally speaking, a test case is more powerful if the absolute value of its HD is smaller, but its BD is larger.

Above we discuss the fault detect potency of HD and BC . In this following section, we will use examples to illustrate the conclusion we obtained: a test case is more powerful if the absolute value of its HD is smaller, but its BD is larger. Those eight types of typical faults in testing of Boolean expression (proposed by Chen and Lau [3][18][19]) are used as an evaluation criterion of the fault detection potency.

To further analyze the ability of fault detection, we show the statistical data of LRF (Literal Reference Fault) detection, where the literal is a value of a condition variable in a term of Boolean expression. LRF means that a literal is replaced by another literal. For example $S = ab + \bar{a}bcd + \bar{a}bcd + \bar{a}bcd$ is changed to $S' = ad + \bar{a}bcd + \bar{a}bcd$ by substituting d for b in the first term. The fault is often referred to as a Variable Reference Fault (VRF) in many

related studies [20]. Table 1 shows all the possible fault cases S' of LRF regarding S , where $S = ab + \bar{a}bcd + \bar{a}bcd + \bar{a}bcd$. Corresponding SC-map is shown in Figure 2.

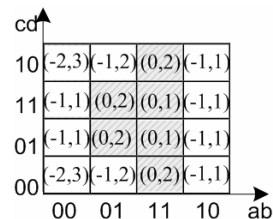


Figure 2. SC-map of the original expression

Note that the test cases (cells in SC-map) with a higher HD do not necessarily detect all the faults of LRF. That is why we still need to generate test cases with smaller HD but large BC .

Table 1 The Cases of LRF

Term	Literal	Figures of Faulty Boolean Expression	
1 st	1 st	$S' = cb + \bar{a}bcd + \bar{a}bcd + \bar{a}bcd$	$S' = \bar{c}b + \bar{a}bcd + \bar{a}bcd + \bar{a}bcd$
		$S' = db + \bar{a}bcd + \bar{a}bcd + \bar{a}bcd$	$S' = \bar{d}b + \bar{a}bcd + \bar{a}bcd + \bar{a}bcd$
	2 nd	$S' = ac + \bar{a}bcd + \bar{a}bcd + \bar{a}bcd$	$S' = \bar{a}c + \bar{a}bcd + \bar{a}bcd + \bar{a}bcd$
		$S' = ad + \bar{a}bcd + \bar{a}bcd + \bar{a}bcd$	$S' = \bar{a}d + \bar{a}bcd + \bar{a}bcd + \bar{a}bcd$

Table 2 shows the statistical data of fault detection of cells with different (HD , BC). Cells $\bar{a}bcd$, $abc\bar{d}$, $\bar{a}bcd$ and $\bar{a}bcd$ have the same HD with cells $\bar{a}bcd$ and $abcd$. But because the former cells have higher BC than the latter, they have much higher average fault

detection rate (25%) compared with (0%). It is observed that test data corresponding to cells with higher BC ($BC \geq 2$) are able to identify more faults than test cases corresponding to cells with lower BC and internal nodes in a topological structure of Boolean expression.

Table 2 Statistical Data of Fault Detection of Cells with Different (HD , BC)

Cells	1 st Term		2 nd Term		Average Rate	(HD, BC)
	1 st	2 nd	1 st	2 nd		
$\bar{a}bcd$	50%	50%	50%	50%	50%	(0,2)
$abc\bar{d}$	50%	50%	50%	50%	50%	(0,2)
$\bar{a}bcd$	0	0	0	0	0	(0,2)
$\bar{a}bcd$	0	0	0	0	0	(0,2)
$\bar{a}bcd$	0	0	0	0	0	(0,1)
$abcd$	0	0	0	0	0	(0,1)
Average fault identifying rate of cells with $BC = 2$						25%
Average fault identifying rate of cells with $BC = 1$						0%

4. A Case Study: Testing a Best Buy Stock WS

Swiss Cheese Model has been applied to select test cases in several industrial applications. This section we

illustrate this model using a Best Buy Stock (BBS) WS as an example. In this example, the WS under development and test consist of a server and multiple clients, residing

in different locations. A client can send requests to the server and the server responses to the requests.

4.1 The original specification

Table 3 lists the functional requirements of the BBS WS. The server offers two functions and the clients can access these two functions.

Table 3 Requirements of Best Buy Stock WS

Event	Requirements
#1 A client queries a stock's price	A client can query any stock's price. If the queried stock name is not empty and the requested stock information is available, the server WS sends the requested stock price to the requesting client.
#2 20 minutes have past since last stock price checking	<p>The Auto-Buy Web Service automatically checks stock prices every 20 minutes. If the information of all stocks is complete, the prices of some stocks increase $\geq 5\%$ within the past 20 minutes, and all stock owners' addresses are available, the server WS will send messages to the all stock owners, reminding them to sell the stocks whose prices increase $\geq 5\%$, or buy the stocks to sell at an even higher price.</p> <p>If the information of all stocks is complete, the prices of some stocks decreases $\geq 10\%$ within the past 20 minutes, and all stock owners' addresses are available, the server WS will send messages to the stock owner, reminding them to buy the stocks whose prices decrease $\geq 10\%$ or sell them to avoid further financial loss.</p> <p>If the advancing volume or declining volume of some stocks increases $\geq 100\%$ in the past 20 minutes compared to the same period of yesterday , the server WS will also send messages to the all stock owners.</p>

4.2 BBS Test Case Selection

The BBS is specified in OWL-S and it can be formalized in the ACDATE & Scenario modeling specification language [12][14]. Then the completeness and consistency (C&C) analysis [14][17] is applied on the formalized BBS specification to detect and eliminate the potential incompleteness and inconsistency. The complete and consistent BBS WS specification includes four scenarios. For example one of the scenarios is the "Stock Price and Trading Volume Auto-Checking" scenario, whose partial OWL-S specification is shown in Figure 3.

The scenario includes the following seven conditions

```

<owl:Class rdf:ID="If-Then-Else">
  <rdfs:subClassOf rdf:resource="#ControlConstruct"/>
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty rdf:resource="#components"/>
      <owl:allValuesFrom rdf:resource="#ControlConstructBag"/>
    </owl:Restriction>
  </rdfs:subClassOf>
</owl:Class>

<owl:ObjectProperty rdf:ID="ifCondition">
  <rdfs:comment> The if condition of an if-then-else</rdfs:comment>
  <rdfs:domain rdf:resource="#If-Then-Else"/>
  <rdfs:range rdf:resource="#BBS_Server.SomeStocksPriceIncreaseGreaterThanOrEqualTo_5Per";#Condition"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="then">
  <rdfs:domain rdf:resource="#If-Then-Else"/>
  <rdfs:range rdf:resource="#ControlConstruct"/>
  <rdfs:process rdf:resource="#BBS_Server.ServerInformClientSomeStocksPriceIncreaseSharply"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="else">
  <rdfs:domain rdf:resource="#If-Then-Else"/>
  <rdfs:range rdf:resource="#ControlConstruct"/>
  .....//More If-Then-Else OWL-S specifications representing other condition combinations
</owl:ObjectProperty>

```

Figure 3. "Stock Price and Trading Volume Auto-Checking" Scenario partial OWL-S specification

From the ACDATE scenarios, a tool [14][17] can be used to automatically extract the scenario expression corresponding to the "Stock Price and Trading Volume Auto-Checking" Scenario, which is:

$$S = c_3 * c_4 + c_3 * c_5 + c_3 * c_6 + c_3 * c_7$$

- c_1 : Queried stock name is not empty
- c_2 : Requested stock information is available
- c_3 : The information of all stocks is complete
- c_4 : The prices of some stocks increase $\geq 5\%$ within the past 20 minutes
- c_5 : The prices of some stocks decreases $\geq 10\%$ within the past 20 minutes
- c_6 : The advancing volume of certain stocks increases $\geq 100\%$ in the past 20 minutes compared to the same period yesterday
- c_7 : The declining volume of certain stocks increases $\geq 100\%$ in the past 20 minutes compared to the same period yesterday.

The SC-map of the above Boolean specification is three dimensional, as shown in Figure 4. In the SC-map, the cells that can satisfy expression S are defined to have $HD = 0$ and those cells that make the expression S be false are defined to have $HD < 0$. Each cell in the

SC diagrams corresponds to a test case that checks if the scenario expression S is satisfied, and values in the cells are the (HD, BC) pair. Test cases with $HD \geq 0$ are positive test cases, testing the specified functions of the Web service; and test cases with $HD < 0$ are negative test cases, testing the missing branches of the Web service specification. The information compacted in the SC diagrams can be unpacked in Table 4, which includes both positive and negative test cases. For example, the cell $(c_3 = 1, c_4 = 0, c_5 = 0, c_6 = 0, c_7 = 1)$ in the SC-map corresponds to the test case (10001) in the table. Considering the topological hypercube structure of Boolean expressions boundary count (BC) in the last column can be obtained.

Among all the 32 candidate test cases, 5 are identified to be the most critical ones. They are listed as following:

1. 10001 ($HD = 0$ && $BC = 2$)
2. 10010 ($HD = 0$ && $BC = 2$)
3. 10100 ($HD = 0$ && $BC = 2$)
4. 11000 ($HD = 0$ && $BC = 2$)
5. 10000 ($HD = -1$ && $BC = 4$)

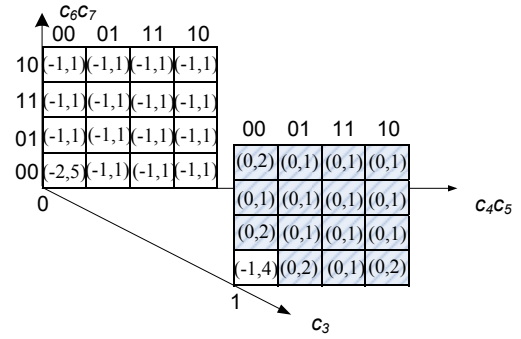


Figure 4. SC-map of the Boolean expression

Table 4 Identified Test Cases

	Test Case (TC)	# of TC	BC
$HD = 0$	10001,10010,10100,11000	4	2
	10011,10101,10111,10110,11100,11101,11111,11110,11001,11011,11010	11	1
$HD = -1$	10000	1	4
	00001,00011,00010,00100,00101,00111,00110,01100,01101,01111,01110,01000,01001,01011,01010	15	1
$HD = -2$	00000	1	5
Total		32	

Table 5 Test Cases Identified for Event 1 Handler Specification of BBS Web Service

Rank	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
# of Failed WSs	24	22	21	18	14	14	13	13	11	10	10	10	10	9	9	9
# of TC in Table 3	3	4	2	1	10	11	12	9	16	7	6	13	14	15	8	5
HD	0	0	0	0	0	0	0	0	-1	0	0	0	0	0	0	0
BC	2	2	2	2	1	1	1	1	2	1	1	1	1	1	1	1
Rank	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
# of Failed WSs	3	3	3	3	3	3	3	3	3	3	3	3	3	2	2	2
# of TC in Table 3	32	27	31	28	29	26	21	20	23	22	24	18	25	30	19	17
HD	-2	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
BC	5	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

To validate the effectiveness of the SC Model, these 32 test cases are used to test 60 experimental BBS Web Services designed based on the same specification. A number of them contain different faults that return can return incorrect values. Obviously, the more faults a test case detect, the more potent the test case is. The test results are shown in Table 5. The 1st row lists the rank of each test case generated by the SC model. The 2nd row counts the number of failed Web Services when testing all the 60 BBS Web Services using the test case with the

rank shown in the 1st row. The 3rd row refers to the index of test cases as shown in Table 4. The 4th and 5th rows give the Hamming Distance (HD) and Boundary distance (BD) of test cases with the rank equivalent to the 1st row and with the index equivalent to the 3rd row.

The validation result is excellent. Among those 5 most critical test cases identified by Swiss Cheese Model, 4 are validated to be REALLY most important – rank 1st to 4th in terms of the number of experimental programs that return incorrect value, and the other one ranks 9th. SC

Model has also been applied to two embedded systems --- Car Alarm System (CAS) and Communication Processor Project for Motorola --- as well as industrial control systems --- Configurable Business Logic Project for Intel. The test data also validate the effectiveness of SC Model in test case selection.

5. Summary and Conclusion

This paper presents an innovative approach for test data selection. It is based on the existing idea of considering the boundary test data as the most potent ones. It extends the idea by adding the next levels of

potency of test data. These additional test data can be applied when the most potent test data cannot detect all the faults. Our previous work and current experiments have shown that the faults that cannot be covered by the boundary test data are not unusual. The hierarchical selection of test data improves the fault coverage significantly. The new model also has its theoretical significance. The fault detection potency of test data is related to two parameters: the Hamming distance and the boundary count. The dictionary order on the parameter ranks the potency of the test data.

References

- [1] D. Beyer, A. Chlipala, T. Henzinger, R. Jhala, and R. Majumdar, "Generating Tests from Counterexamples", Proceedings of the 26th International Conference on Software Engineering (ICSE'04), Scotland, UK, May 2004, pp. 326 - 335.
- [2] E. Clarke, O. Grumberg, and D. Peled, *Model Checking*, MIT Press, 2002.
- [3] T. Y. Chen and M. F. Lau, "Test Cases Selection Strategies Based on Boolean Specifications", *Software Testing, Verification and Reliability*, Vol. 11, No. 3, Sep. 2001, pp.165-180.
- [4] J. Clune and L. Chen, "Testing Web Services: Methods for Ensuring Server and Client Reliability" at <http://www.syscon.com/websphere/>
- [5] N. Davidson, "Testing Web Services" at <http://www.webservices.org/>, in October 2002.
- [6] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre, "Lazy Abstraction", In Proceedings of the 29th Annual Symposium on Principles of Programming Languages, 2002, pages 58-70.
- [7] G. Holtzman, "The Spin Model Checker," *IEEE Transactions on Software Engineering*, Vol. 23, No. 5, May 1997, pp. 279-295.
- [8] J. Myerson, "Testing for SOAP Interoperability" at <http://www.webservicesarchitect.com/>, Feb 2002.
- [9] Wim De Pauw, et al., "Websight Visualizing the Execution of Web Services", Workshop on Testing, Analysis and Verification of Web Services, Boston, MA, July 2004.
- [10] W. T. Tsai, R. Paul, Y. Wang, C. Fan, and D. Wang, "Extending WSDL to Facilitate Web Services Testing", *Proc. of IEEE HASE*, 2002, pp. 171-172.
- [11] W. T. Tsai, R. Paul, Z. Cao, L. Yu, A. Saimi, and B. Xiao, "Verification of Web Services Using an Enhanced UDDI Server", *Proc. of IEEE WORDS*, 2003, pp. 131-138.
- [12] W. T. Tsai, R. Paul, L. Yu, A. Saimi, and Z. Cao, "Scenario-Based Web Service Testing with Distributed Agents", *IEICE Transactions on Information and Systems*, 2003, Vol. E86-D, No. 10, 2003, pp. 2130-2144.
- [13] W.T. Tsai, Lian Yu, Feng Zhu and Ray J. Paul, "Rapid Verification of Embedded Systems Using Patterns", *COMPSAC 2003*: 466-471.
- [14] W. T. Tsai, R. Paul, L. Yu, X. Wei, and F. Zhu, "Rapid Pattern-Oriented Scenario-Based Testing for Embedded Systems", to appear in book *Software Evolution with UML and XML*, edited by H. Yang, 2004.
- [15] W. T. Tsai, D. Zhang, Y. Chen, H. Huang, Ray Paul, and Ning Liao, "A Software Reliability Model for Web Services", the 8th IASTED International Conference on Software Engineering and Applications, Cambridge, MA, November 2004, pp. 144-149.
- [16] W. T. Tsai, Y. Chen, Zhibin Cao, Xiaoying Bai, Hai Huang, and Ray Paul, "Testing Web Services Using Progressive Group Testing", *Advanced Workshop on Content Computing*, Zhenjiang, China, November 2004, pp. 314-322.
- [17] W. T. Tsai, X. Wei, Y. Chen, B. Xiao, R. Paul, and H. Huang, "Developing and Assuring Trustworthy Web Services", *ISADS*, 2005
- [18] T. Y. Chen and M. F. Lau, "Two test cases selection strategies towards testing of Boolean specifications", in Proceedings of the 21st Annual International Computer Software and Applications Conference (COMPSAC'97), pp.608-611.
- [19] T. Y. Chen, M. F. Lau and Y. T. Yu, "MUMCUT: A fault-based strategy for testing Boolean specifications", in Proceedings of 1999 Asia-Pacific Conference on Software Engineering (APSEC'99), Dec. 1999, pp. 606-613.
- [20] E. J. Weyuker, T. Goradia, and A. Singh, "Automatically generating test data from a Boolean Specification", *IEEE Transactions on Software Engineering*, 1994, Vol.20, No. 5, pp.353-363.
- [21] W. T. Tsai, Y. Chen, R. Paul, H. Huang, X. Zhou and X. Wei, "Adaptive Testing, Oracle Generation, and Test Case Ranking for Web Services", *COMPSAC*, 2005.
- [22] W. T. Tsai, X. Wei, Y. Chen, R. Paul and L. Yu, "Swiss Cheese Model for Software Testing Based on Boolean Expressions and Hamming Distance", submitted to *HASE 2005*.