# Built-in structural testing of web services

Marcelo Medeiros Eler, Marcio Eduardo Delamaro, Jose Carlos Maldonado, Paulo Cesar Masiero
Instituto de Ciencias Matematicas e de Computacao Universidade de Sao Paulo
P.O. 668 – Sao Carlos – Brasil – 13560-970
Email:{mareler, delamaro, jcmaldon, masiero}@icmc.usp.br

***Abstract*—Testing Service Oriented Architecture applications is a challenging task due to the high dynamism, the low coupling and the low testability of services. Web services, a popular implementation of services, are usually provided as black box and using testing techniques based on implementation is limited. This paper presents an approach to support the use of the structural testing technique on web service testing. The approach improves web service testability by developing web services with built-in structural testing capabilities. Testers can run test cases against such web services and obtain a coverage analysis on structural testing criteria. A set of metadata provided with the testable web service helps testers to evaluate the coverage reached and the quality of their test cases. An implementation of the approach is presented using a service called JaBUTiWS that performs instrumentation and coverage analysis of Java web services. We also present a usage scenario of the approach.**

## I. Introduction

Service Oriented Architecture (SOA) is an architectural style that uses services as the basic constructs to support the development of rapid, low-cost, loosely-coupled and easily integrated applications even in heterogeneous environments [1]. Web services are an emerging technology to integrate applications using open standards based on XML that have become a well adopted implementation of SOA requirements.

Testing SOA applications is a challenging task due to the complex nature of web services, the high dynamism, the low coupling and the low testability of services [2]. Testability is the degree to which a system or service supports the establishment of test criteria and the performance of tests to determine whether those criteria have been met. It is also an important quality indicator since its measurement leads to the prospect of facilitating and improving a service test process [3], [4].

Service-oriented software has low testability because it is more difficult to setup and trace the execution of a test set when the system elements are on different places across the network [3]. Moreover, web services have low testability because they are usually seen as black box since they are only provided with their interfaces. Designing web services with high testability is an important task for developers since it would increase the quality of composite services and reduce the cost of developing and testing [4], [5].

Canfora and Penta claim that there are five perspectives for testing a service: developer, provider, integrator, certifier and user [5]. The testability of web services is different for each perspective. The developer/provider has full access to artifacts related to the service implementation and for him/her the service has high testability, because any available testing technique can be applied. For the other perspectives, on the other hand, web services have low testability because they have only access to interfaces and specifications. In this paper we also user the perspective tester that represents anyone who wants to test a web service from an external perspective (integrator, certifier, user).

Software Components and web services have many similarities. Both of them are self-contained composition units and are accessed by explicit published interfaces [6]. According to Weyuker, software components should be tested before using it even if it had been tested during development time [7]. Brenner et. al. believe that the same care should be taken for web services and they claim that web services should be tested during runtime to assure their confiability across the time [8].

The testablity of web services is an important key to allow clients to perform a suitable testing activity. We found many approaches in the literature for web service testing. Most of the research is based on testing web services through their interfaces' specification. Some authors have proposed enriching the WSDL file with semantic markup and other information to improve the web service testability and to facilitate the derivation of richer test cases [9]–[11]. Other authors suggested that developers make available additional information about services (often called metadata in many contexts) to help the conduction of tests based on other information than signatures. These metadata can be models with the internal structure of the service, test cases, testing scripts and/or details of the internal implementation [12], [13].

A few approaches have been developed for exploring structural testing of web services, but they are mainly based on workflow testing or control and data-flow graphs generated from the WSDL and/or BPEL specifications [14]–[19]. Most of these approaches still consider web services as black boxes.

In this context, the purpose of this paper is to present the BISTWS (Built-in Structural Testing of Web Services) approach that supports the use of the structural testing technique on web service testing. The approach improves SOA testability by providing web services with structural testing facilities (testable web services). A testable web service has operations to provide a structural coverage analysis report based on a test session. Testable web services can also be provided with metadata to help testers to better understand the web service under test and to improve their test set.

We believe that developers/providers would be interested in providing testable web services since testability is a quality

978-0-7695-4273-7/10 $26.00 © 2010 IEEE
DOI 10.1109/SBES.2010.15

indicator [3]–[5] and it would be a competitive advantage. We also believe that testers would be interested in using testable web services since a coverage analysis report allows to evaluate the quality of their test case by indicating how much they are exercising the web service under test (in a single or in a composition context).

BISTWS is generic and in this paper we present an implementation of the approach using Java web services. We also show an usage scenario in which a developer/provider created a web service and used the proposed approach to generate a testable web services. On the other side, we also show the activities performed by a tester to execute a test session and get a coverage analysis report.

This paper is organized as follows. In Section 2, the BISTWS approach is presented in detail. In Section 3, an overview of the web service for structural testing JaBUTiWS that allows a particular instantiation of the BISTWS approach is presented. In Section 4, an usage scenario of BISTWS with JaBUTiWS is presented. In Section 5, related work is discussed. Section 6 presents the conclusions of this work.

## II. THE BISTWS APPROACH

BISTWS is a conceptual approach devised to introduce structural testing into SOA development. The main idea is to improve web services testability by introducing structural testing facilities into their code and interface. This kind of web service is called testable web service. A testable web service can trace its own execution when a test session is carried out and can generate a structural coverage analysis based on the traces generated.

The feasibility of the approach requires the contribution of many stakeholders. The developer/provider must agree on instrumenting the code of the web service to introduce structural testing capabilities. The tester must set the web service under test to a test session mode, run a set of test cases and use operations to query for a structural coverage analysis report. Additionally, the developer/provider should provide a set of metadata with the testable web service to help testers to improve their test set.

Instrumenting code to allow structural testing is not an easy task. The instrumentation adds extra code to trace the execution of the code under test to generate information about what parts (data, paths, nodes) were exercised. A set of test requirements that should be met by test cases should also be generated during the instrumentation. A structural coverage analysis use test requirements and trace files to produce a report indicating how much the test requirements were covered by the test cases executed during a test session.

Performing structural instrumentation and coverage analyzis manually requires much effort and it would be error prone. For that reason BISTWS relies on a web service called TestingWS that automates the approach. TestingWS is generically defined as a structural testing service that is able to instrument a web service and perform coverage analysis based on trace files and test requirements.

Figure 1 shows an illustration of the BISTWS approach. The approach defines how developers/providers should use TestingWS to produce a testable web services and which type of metadata should be provided to help testers understand the web service under test and to improve their test set. The approach also shows how testers can use the structural testing facilities of testable web services to execute a test session, get a coverage analysis report and evaluate the results.

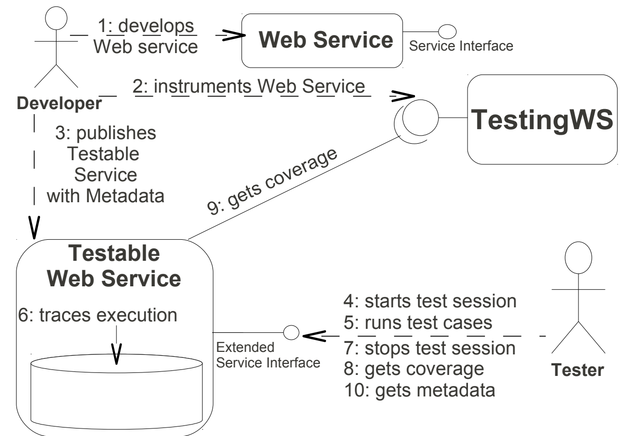


Fig. 1. BISTWS approach.

TestingWS and each step of the approach is presented in details as following. The steps of the approach are split into Developers' and Tester's perspective. The developer perspective in this paper also represents the provider perspective and the tester perspective represents the integrator, certifier and user.

### A. *TestingWS*

TestingWS plays the role of a generic structural testing web service that is able to produce testable web services and to perform coverage analysis. TestingWS have two operations: one to receive a regular web service and return a testable web service and another to generate a coverage analysis report based on trace files generated during the execution of testable web services. The description here is general and specific details of TestingWS's operations and activities (instrumentation and coverage analysis) must be defined for each implementation.

The operation to generate a testable web service should perform the following activities:

1) insert instructions into the web service implementation to give to the instrumented web service the capability to trace its own execution.
2) analyze the web service implementation and generates a set of test requirements for the structural criteria adopted (data-flow and/or control-flow criteria, for example).
3) insert operations to define test sessions (`startTrace` and `stopTrace`) and to return coverage analysis (`getCoverage`) into the instrumented web service. These are important operations since the testing code

inserted into the web service during the instrumentation may bring overhead to its execution and it can be avoided or mitigated if the testing code is executed only during a test session.
4) insert operations to handle metadata: `getMetadataTags`, that returns a list of tags with the identification of available metadata (coverage, test case, test requirements); and `getMetadata`, that returns the metadata required.
5) insert an operation to perform relative coverage (`getRelativeCoverage`). This operation compares the coverage score reached by the tester with the one reached by the developer. It can give to the tester an idea on how close he/she is to the developer's coverage.

The operation to generate a coverage analysis report receives a trace file and an identification of a testable web service. TestingWS retrieves the test requirements of the testable web service that were stored during instrumentation and uses the trace file received to calculate which requirements were met during the test execution related to the trace received.

*B. Developer's perspective*

The following steps presents the activities of the developer to create a testable web service using the BISTWS approach.

**Step 1 - Develops service:** The developer creates a web service using any programming tools and languages. He/She also develops a set of test cases using any available testing technique.

**Step 2 - Instruments service:** The developer wants to provide a web service with high testability to its clients and submits the web service developed to be instrumented by TestingWS. The developer receives a testable web service.

**Step 3 - Publishes testable service with metadata:** The developer create a set of metadata and publishes the testable web service with metadata on a web service container. The type and structure of the metadata that should be published with testable web services should be defined by particular implementations of TestingWS. The developer must publish the testable version of the web service to allow clients to test it and get a coverage analysis during runtime.

*C. Tester's perspective*

The following steps presents the activities to test a testable web service and to get a structural coverage analysis to evaluate the test set executed.

**Step 4 - Starts test session:** The tester invokes the `startTrace` operation to begin a test session. The `startTrace` operation takes `userID` and `sessionID` as input parameters. These parameters are used by the testable web service to identify which trace was generated during which test session and for which user.

**Step 5 - Runs test cases:** The tester runs a test set against the testable web service during a test session.

**Step 6 - Traces execution:** The testable web service traces which instructions, branches and data were exercised during a test session every time an operation is called. The trace generated is identified by `userID` and `sessionID` and is locally stored by the testable web service.

**Step 7 - Stops test session:** The tester invokes the `stopTrace` operation to stop a test session. The `stopTrace` operation also takes `userID` and `sessionID` as input parameters.

**Step 8 - Gets coverage:** The tester calls the `getCoverage` operation using the test session identifiers (`userID` and `sessionID`) as input parameters to obtain a structural coverage analysis.

**Step 9 - Gets coverage:** The testable web service executes the `getCoverage` operation and delegates to the TestingWS the task of performing coverage analysis. The testable web service access the specified trace and send it to TestingWS. TestingWS performs the coverage analysis and replies to the testable web service that returns the report to the tester. The tester does not need to know in fact that TestingWS is performing the analysis.

The structural coverage analysis is done as follows. TestingWS uses the trace received to determine which test requirements generated during instrumentation for each structural criteria were covered during that particular execution (test session). Performing this kind of analysis manually requires much effort and it should be done by a tool. We recommend that TestingWS also plays the role of the analyzer, since it has access to the test requirements generated during instrumentation.

The coverage analysis can be presented in many ways and can reveal internal details of the web service that are not exposed through its interface. We suggest four types of report: coverage analysis by the whole service, by interface operation, by class and by method (considering an object-oriented implementation). Developers should decide, during instrumentation, which types of coverage may be reported on coverage analysis.

Steps 4 to 9 comprises the main activities of BISTWS approach, because they are related to a test session. Figure 2 shows a UML Sequence Diagram of a test session in BISTWS, from the `startTrace` to the `getCoverage` operation. The diagram shows the interaction among the Tester and the Testable Web Service (TWS) and the TWS and the JaBUTiWS. The tester does not interact with JaBUTiWS directly.

**Step 10 - Gets metadata:** Testers use the coverage analysis to evaluate the quality of their test cases concerning structural criteria, but in many cases they do not have enough information to decide whether the coverage reached is good or whether it needs improvement. This happens because web services are usually provided as black box and testers do not have detailed information about the implementation. They do not know if test cases are missing or there are infeasible requirements. BISTWS mitigate this situation using metadata in many ways:

1) Testers can use the `getRelativeCoverage` operation to compare the coverage analysis achieved with the coverage analysis provided as metadata by the developer. We assume that the developer has access to the web

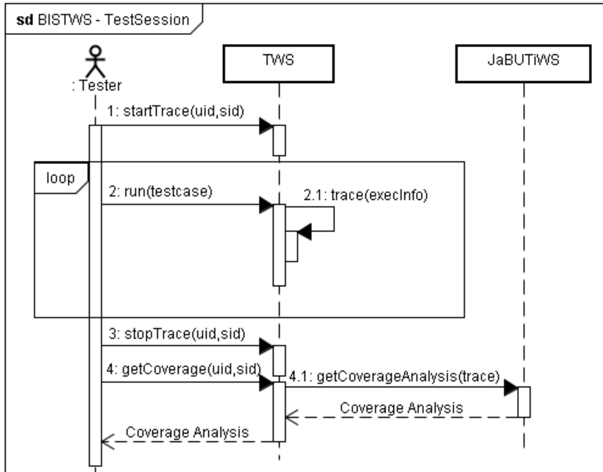


Fig. 2. A UML Sequence Diagram of a test session in the BISTWS approach

service implementation and can cover all feasible test requirements. Testers can be satisfied with being close to the developer's coverage even if they have not reached 100% coverage for some criterion.

2) Testers can reuse the whole test set provided with the testable web service as metadata and evaluate the coverage achieved.
3) Testers can study the developer's test case to realize what test cases are missing on their test set. Suppose that a testable web service is being tested from the perspective of a composition and the composition handles invalid entries. In such case the testable web services would never be invoked with invalid entries. The tester can realize that test cases for invalid entries are the only test cases that are missing in comparison with the

### D. Governance Rules

The feasibility of the BISTWS approach lays on some governance rules since they establish rights and duties of each actor involved [20]. Developers should follow these rules:

- The developer should agree on sending the implementation (source-code or binary-code) of the web service to TestingWS. In many cases, the developer could not be comfortable with providing this artifact, but he/she needs to trust the confidentiality provided by TestingWS.
- The developer should provide a set of metadata with the testable web service to help testers on evaluating and improving the coverage achieved.
- The developer should use the metadata structure provided by TestingWS to generate the metadata required.
- The developer should not change the testable web service for a non-testable version. Each update should be submitted to TestingWS instrumentation. The metadata should be updated too.

TestingWS should follow these governance rules:

- TestingWS should receive a web service and generate a testable web service with structural testing capabilities.
- TestingWS should keep internal information of web services stored locally to perform coverage analysis using the trace file received. These information can be the test requirements, the instrumented code or models that provides data to perform coverage analysis based on a trace file.
- TestingWS should provide the metadata structure to the developer.
- TestingWS should provide supporting tools that helps developers and testers to perform their activities using the BISTWS approach.
- TestingWS should handle non-functional issues (authentication, availability, concurrency, confidentiality and security).

## III. JaBUTiWS: a TestingWS implementation

The BISTWS approach is conceptual and we present here a particular implementation of the approach. In previous work we developed a testing web service called JaBUTiWS that supports structural testing of Java programs [21]. We extended JaBUTiWS to comply with the BISTWS and the TestingWS requirements. In this section we show the details of this implementation.

### A. The previous JaBUTiWS implementation

JaBUTiWS (Available at www.labes.icmc.usp.br/~jabutiservice) is a structural testing web service that was developed based on the tool JaBUTi (Java Bytecode Understanding and Testing) developed by Vincenzi et. al. [22] to support structural testing of object-oriented programs written in Java. JaBUTi implements some testing coverage criteria that are used in the context of unit testing: all-nodes, all-edges and all-uses. One of the advantages of JaBUTi is that it does not require the Java source code to perform its activities because all static and dynamic analysis are based on the Java bytecode.

The architecture of the JaBUTiWS is presented in Figure 3 and it comprises four components: 1) Axis2 engine; 2) JaBUTiWS Controller; 3) Database (DB); and 4) JaBUTiCore. *Axis2* is a Java-based implementation for both the client and server sides to send, receive and process SOAP messages. The *Controller* component implements the operations published on the WSDL interface. It is a controller that receives messages, accesses the Database and calls JaBUTiCore operations to perform instrumentation and coverage analysis. The *Database* stores testing projects' information, including test requirements, test cases and trace files. The *JaBUTiCore* component wraps the core classes of the JaBUTi tool that handle instrumentation and coverage analysis.

A comprehensive set of operations was defined to provide the structural testing service that would be useful for the service clients. JaBUTiWS is a stateful web service and needs to follow a sequence of execution. First the tester creates a project and sends the object program to be instrumented. The tester then gets the instrumented program and runs it against test cases. A trace file with execution analysis is generated and then sent to the JaBUTiWS. JaBUTiWS uses the trace to

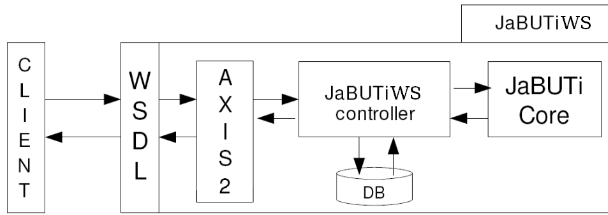


Fig. 3. Architecture of the JaBUTiWS.

analyze which requirements were covered and which were not to generate a coverage analysis according to the implemented criteria.

JaBUTiWS instrumentation is done using BCEL (Byte Code Engineering Library) and the test requirements generated in this phase are written in a XML file. JaBUTiWS does not need to store the instrumented implementation because it performs the coverage analysis using a trace file and the XML file with the test requirements generated during instrumentation.

### *B. The JaBUTiWS implementation to comply with TestingWS requirements*

We have extended JaBUTiWS to support BISTWS and to comply with the TestingWS requirements. We extended the JaBUTiWS operations to include an operation to receive a common web service and transform it into a testable web service according to BISTWS recommendations. Details concerning the generation of testable services, the coverage analysis and the metadata handling are presented as following.

*1) Testable web services generation:* JaBUTiWS has an operation to generate testable web services. This operation takes a deployment package (.war) as input parameter and produces a deployment package with a testable service. The instrumentation is done as follows:

1) JaBUTiWS instruments the web service using the original instrumentation with BCEL and gives to the testable web service the capability to write to a trace file details (instructions, branches and attributes exercised) of its own execution.
2) JaBUTiWS analyzes the web service implementation and generates a set of test requirements into a XML file for these criteria: all-nodes, all-edges and all-uses.
3) JaBUTiWS inserts into the web service the six operations suggested by the BISTWS approach: `startTrace`, `stopTrace`, `getCoverage`, `getRelativeCoverage`, `getMetadataTags` and `getMetadata`.

The operation to create a testable web service also takes the access level as an input parameter to determine which details may be presented in the coverage analysis report.

*2) Coverage Analysis:* JaBUTiWS performs coverage analysis using a trace file with execution details and a set of test requirements generated during instrumentation. The coverage analysis can be done for the whole service and for its operations (level 1), for classes (level 2) and for methods (level 3).

*3) Metadata handling:* We define, in this particular implementation of BISTWS, two types of metadata that Developers should produce and publish with the testable web service: the test set used to test the web service and the coverage analysis obtained. Figure 4 shows a snippet of a XML test case metadata. The root element is `<testcases>`. Each operation (`<operation>`) is under the root element with a attribute `name`. Each operation defines a set of test cases (`<testcase>`). Each test case has identification, a set of input parameters, an output value and a description (rationale) of the test case. In this case the description is hidden to safe space.

```
1  <testset>
2    <operation name="checkID">
3      <testcase id="checkID-1">
4          <input name="id" type="xs:string">29935661806</input>
5        <expected>true</expected>
6      </testcase>
7      <testcase id="checkID-2">
8          <input name="id" type="xs:string">11111111111</input>
9        <expected>false</expected>
10     </testcase>
11     <testcase id="checkID-3">
12         <input name="id" type="xs:string">12355454454</input>
13       <expected>false</expected>
14     </testcase>
15     <testcase id="checkID-4">
16         <input name="id" type="xs:string">12121</input>
17       <expected>false</expected>
18     </testcase>
19   </operation>
20 </testset>
```

Fig. 4. Snippet of a test case metadata

Figure 5 shows a snippet of a XML coverage metadata. The root element is `<coverage>`. Each type of coverage (`<coveragebyservice>`, `<coveragebyoperation>`, `<coveragebyclass>` and `<coveragebymethod>`) is declared under the root element. The coverage by service defines the service name and the coverage for each criterion under the service element. The coverage by operation defines the operation name and the coverage for each criterion under the operation element. The same structure of coverage by service and coverage by operation is applied to the coverage by class and coverage by method but it is not shown in Figure 5.

*4) Tool Support:* We have developed a tool called WSMTS (Web Service Monitoring and Testing System) to help developers and testers on their activities. Developers can use WSMTS to invoke JaBUTiWS to generate a testable web service and can use WSMTS to create the metadata required by JaBUTiWS implementation of BISTWS. Testers can use WSMTS to access testable web services, create and run test cases, get coverage analysis and get metadata. The full description of WSMTS is not in the scope of this paper. We present some illustrations of WSMTS user interface in Section V and here we highlight its main features and their relation with the BISTWS steps:

- **Testable web service generation (step 2)**. WSMTS is used to invoke JaBUTiWS to generate a testable web service.

```
1  <coverage>
2    <coveragebyservice>
3      <service name="IDChecker">
4        <All-Nodes req="58" cov="20">34.0</All-Nodes>
5        <All-Edges req="76" cov="21">27.0</All-Edges>
6        <All-Uses req="195" cov="40">20.0</All-Uses>
7      </service>
8    </coveragebyservice>
9
10   <coveragebyoperation>
11     <operation name="net.id.Checker.isCNPJValid(Ljava/lang/String;)Z">
12       <All-Nodes req="37" cov="0">0.0</All-Nodes>
13       <All-Edges req="52" cov="0">0.0</All-Edges>
14       <All-Uses req="139" cov="0">0.0</All-Uses>
15     </operation>
16     (...)
17   </coveragebyoperation>
18     (...)
19 </coverage>
```

Fig. 5. Snippet of a coverage analysis metadata

- **Metadata generation (step 3)**. Developers can use WSMTS to create a set of test cases and export them to XML. They can also invoke a testable web service using the test cases developed and get a coverage analysis in a XML format. The XML structure used by WSMTS complies with the JaBUTiWS's requirements to metadata structure.
- **Testing project creation**. Testers can create testing projects and select testable web services to test. Testers must provide the name and the endpoint address of the testable web service. WSMTS access the WSDL file of the provided testable web service and automatically extract the published operations that should be tested.
- **Test set design**. Testers can use the WSMTS interface to create test cases to the operations of the testable web service. WSMTS automatically identifies the operations of the testable web service under test using a WSDL parser.
- **Test session execution (steps 4 to 9)**. Testers can set the testable web service to a test session mode and execute a test set. When the testable web service under test is in a test session mode, WSMTS automatically invokes the `startTrace` before and `stopTrace` after the test set execution. WSMTS also invokes the `getCoverage` operation of the testable web service after the test session and presents the results of each test case and the coverage analysis achieved.
- **Metadata visualization (step 10)**. Testers can use WSMTS to extract metadata information from the testable web service. If the metadata is structured as JaBUTiWS requires, WSMTS shows the metadata in formated tables. Otherwise, XML document is presented as it is.

## IV. Usage Scenario

The JaBUTiWS implementation of the BISTWS approach is presented in this section by a simple example. Consider a scenario in which a developer created a web service called ShippingWS and uses JaBUTiWS to transform it into a testable web service before its publication. A tester selected this service to use into a service composition, implemented some test cases, got a coverage analysis report and used the metadata available to evaluate the report achieved. This scenario is presented in details following the BISTWS steps and perspectives.

### A. *Developer's perspective*

ShippingWS is a web service with an operation to query addresses based on a zip code and an operation to calculate a shipping price based on a source zip code, a destination zip code, a weight and a type of shipping service. There are two types of shipping services: fast (type 1) and basic (type 2). If the object weights more than 500g, the shipping price of fast shipping is used, even if basic shipping is selected. The return value is 0 if weight or service is invalid (weight$\leq$ 0 or 3$\leq$ service$\leq$ 0) and -1 if source or destination zip code is invalid.

**Step 1 - Development:** The developer implemented ShippingWS using the Axis2 library and the Eclipse platform. Figure 6 presents the class diagram of the ShippingWS web service. The interface of ShippingWS has two operations: `calcShippingPrice` and `getAddress`. The Shipping class implements the ShippingWS interface and uses the following class: PriceMgr and ZipMgr. The developer used the test cases forms of WSMTS to created the set of test cases shown in Table I to test ShippingWS.

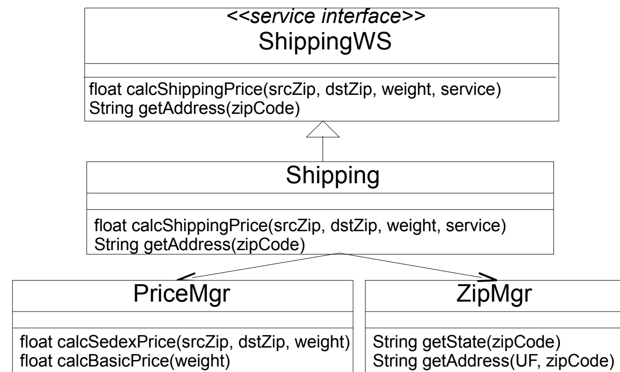


Fig. 6. Class Diagram of ShippingWS.

**Step 2 - Instrumentation:** Figure 8 shows the WSMTS entry form used by the Developer to instrument ShippingWS using JaBUTiWS. The developer informed what is the endpoint address of JaBUTiWS, the location of the .war file of ShippingWS implementation, the implementation of the ShippingWS interface and the destination of the .war file of the testable version of ShippingWS. The developer also selected the level of details that may be reported on coverage analysis. In this case, the developer set the access level 3 that allows testers to get coverage analysis by service, by interface operations, by classes and by methods. WSMTS used these information to invoke JaBUTiWS and get a testable version of ShippingWS.

Figure 7 shows the ShippingWS interface after the instrumentation. According to the recommendations of BISTWS,

TABLE I
DEVELOPER'S TEST CASES TO TEST SHIPPINGWS

| TC-ID | Input | Output | Rationale |
|---|---|---|---|
| **CalcShippingPrice** | **srcZip, dstZip, weight, service** | **Expected value** | **Description** |
| 01 | 13566580, 19500000, 0.3, 1 | 11.9 | Valid parameters using service 1 |
| 02 | 13566580, 13566580, 0.4, 2 | 6.85 | Valid parameters using service 2 and lighter than 500g |
| 03 | 13566580, 13566580, 0.7, 2 | 11.2 | Valid parameters using service 2 and heavier than 500g |
| 04 | 13566, 13566580, 0.7, 2 | ZipFault | Invalid source zip using service 2. |
| 05 | 13566580, 130, 0.7, 2 | ZipFault | Invalid destination zip using service 2. |
| 06 | 13566580, 13566580, -0.5, 2 | InputFault | Invalid weight using service 2 |
| 07 | 13566580, 13566580, 0.5, 3 | InputFault | Invalid service |
| **GetAddress** | **zipCode** | **Expected value** | **Description** |
| 01 | 13566580 | Miguel Alves Margarido Street | Valid parameters |
| 02 | 1340, | Invalid zip code | Invalid parameters |

ShippingWS have gotten operations to support structural testing and handle metadata.

<<service interface>>
ShippingWS

float calcShippingPrice(srcZip, dstZip, weight, service)
String getAddress(zipCode)
String[] getMetadataTags()
XML getMetadata(String tag)
void startTrace(userID, sessionID)
void stopTrace(userID, sessionID)
XML getCoverage(userID, sessionID, reportType)
XML getRelativeCoverage(userID, sessionID, reportType)

Fig. 7. ShippingWS interface after instrumentation.

Fig. 8. Testable Web Service Generation

**Step 3 - Publication:** The developer ran the test set created using WSMTS and get a coverage analysis. The developer used WSMTS to export the test set and the coverage analysis achieved to a XML format that complies with the specifications of the metadata required by JaBUTiWS. Thus, the developer published the testable version of ShippingWS and its metadata on a Tomcat container.

## B. *Tester's perspective*

An integrator wanted to use a web service with the same specifications of ShippingWS. ShippingWS was found into a registry of web services and a tester was required to guarantee that ShippingWS is working perfectly. The following steps show the activities performed by a tester to test ShippingWS.

**Steps 4 to 9 - Test session:** The tester access the interface specification of ShippingWS and used the test cases forms of WSMTS to create a set of test cases. Figure 9 shows the test cases created to test ShippingWS. Notice that there are test cases for both operations of ShippingWS. The TC-ID column represents the identifier of the test case. The input values of the test case are from the first column after TC-ID to the column right before Expected value. The Expected Value is the oracle of the test case.

calcShippingPrice — Save — Close

| TC-ID | sourceZip | destZip | weight | service | Expected Value |
|---|---|---|---|---|---|
| calcShippingPrice-1 | 13564390 | 03356001 | 0.4 | 1 | 13.4 |
| calcShippingPrice-2 | 60150190 | 50060001 | 0.4 | 2 | 11.5 |
| calcShippingPrice-3 | 60150190 | 50060001 | 0.7 | 2 | 19.1 |

getAddress — Save — Close

| TC-ID | zipCode | Expected Value |
|---|---|---|
| getAddress-1 | 63100.000 | St Antonio Street |

Fig. 9. Test cases created to test ShippingWS

After creating the test cases, the tester used WSMTS to set ShippingWS to a test session mode. The tester used WSMTS to run the test cases presented in Figure 9 and the following activities were executed at this point:

1) WSMTS called the `startTrace` operation of ShippingWS.
2) WSMTS ran each test case of the selected test case.
3) WSMTS called the `stopTrace` operation of ShippingWS.
4) WSMTS called the `getCoverage` operation of ShippingWS and presented a coverage analysis report. Table II shows the coverage analysis achieved with the tester's test cases. Each line at the coverage report shows the number of test requirements covered over the number of test requirements. Notice that the tester could not reach 100% of coverage in any criterion for any entity (service,

operation, class, method). We do not show the screen of the WSMTS here to save space.

5) WSMTS presented a report with the status of each test case executed (passed or failed).

TABLE II
STRUCTURAL COVERAGE ANALYSIS

| Service | All-nodes | All-edges | All-uses |
|---|---|---|---|
| ShippingWS | 25/42(60%) | 9/51(57%) | 58/90(64%) |
| **Operation** | **All-nodes** | **All-edges** | **All-uses** |
| calcShippingPrice | 8/13(62%) | 8/18(44%) | 22/34(65%) |
| getAddress | 2/3(67%) | 1/2(50%) | 7/9(78%) |
| **Class** | **All-nodes** | **All-edges** | **All-uses** |
| PriceMgr | 7/13(54%) | 10/16(63%) | 15/25(60%) |
| Shipping | 10/16(63%) | 9/20(45%) | 29/43(67%) |
| ZipMgr | 8/13(62%) | 10/15(67%) | 14/22(64%) |
| **Method** | **All-nodes** | **All-edges** | **All-uses** |
| PriceMgr.calcBasicPrice | 4/7(57%) | 6/9(67%) | 11/16(69%) |
| PriceMgr.calcSedexPrice | 3/6(50%) | 4/7(57%) | 4/9(44%) |
| Shipping.calcShippingPrice | 8/13(62%) | 8/18(44%) | 22/34(65%) |
| Shipping.getAddress | 2/3(67%) | 1/2(50%) | 7/9(78%) |
| ZipMgr.getAddress | 4/7(57%) | 5/8(53%) | 8/13(62%) |
| ZipMgr.getState | 4/6(67%) | 5/7(71%) | 6/9(67%) |

**Step 10 - Metadata usage:** The tester was not confident that the coverage reached was enough. The tester used WSMTS to get the metadata provided by ShippingWS and performed the following activities:

1) Coverage Analysis comparison: The tester saw the coverage analysis reached by the developer's test cases execution and realized that there were not infeasible requirements. The tester realized that it would be possible to reach the maximum coverage improving the test set used.
2) Test cases study: The tester studied the developer's test cases and realized that there were no test cases for invalid input parameters in his/her original test set. The tester thus improved the test set with test cases for invalid entries and this time he/she could achieve 100% of coverage for all criterion.

Figure 10 shows an illustration of the last configuration of the WSMTS project to test ShippingWS. TestShippintWS is associated to the ShippingWS testable web service. The test set (test-ShippingWS-1) was created to test ShippingWS. ShippingWS also has two metadata: coverage.xml and test-cases.xml. The execution of the test set produced a test set result (test-ShippingWS-1-exec-1.xml) and a coverage analysis report (test-ShippingWS-1-execCov-1.xml). The right side of the picture shows the coverage analysis achieved after the last test session.

### C. Discussion

The example presented is simple but we could learn many things during its execution. We could realize that the adoption of BISTWS using JaBUTiWS have little impact on developers and testers tasks. Developers can easily transform web services into a testable version since all transformation and coverage analysis are automatically performed by JaBUTiWS. It is even easier when the developer uses a supporting tool like WSMTS. Their main effort is to produce the set of metadata in the required XML format, but this task is straightforward when the developer has already created a set of test cases and uses supporting tools.

The example also shows that is easy to get structural testing information from a testable web service. There is no difference between using one or another version of the web service considering the regular operations. The difference is that the tester needs to invoke the operation `startTrace` before and the operation `startTrace` after a test session if he/she wants to get a coverage analysis invoking `getCoverage` operation. In the example presented, these operations were automatically invoked by WSMTS.

We could also realized that BISTWS has some limitations. BISTWS is highly dependent of TestingWS and the testing code of the instrumented web services brings overhead to the architecture. The dependency of TestingWS could be solved by giving to the testable web service the capability of performing coverage analysis instead of calling TestingWS to perform this task. It depends on the implementation of TestingWS and the instrumentation phase, but we believe that, in general, that solution would bring a greater overhead to the testable web service code.

The overhead issue also depends on the implementation of TestingWS. In the JaBUTiWS implementation, for example, the testing code is only activated to generate a trace file when a testable web service is set to a test session mode (`startTrace`). However, the testing code is also executed when a test session is not being carried out. The difference is that no trace file is generated outside a test session. Moreover, other clients may also be affected by the overhead when they were using a testable web service that was set to a test session mode by some client.

We have done a performance analysis to evaluate how much is the overhead inserted by BISTWS in a SOA architecture using testable web services generated by JaBUTiWS. We created a set of test cases and executed them fifity times against a web service and its testable version. Considering the testable version, we executed the test set inside and outside a test session. We measured the average time to execute the whole test set in the three situations mentioned above and summarize the results on Table III. Notice that the overhead of the testable web service outside a test session is 2.65% and inside a test session is 5.26%.

The response time presented in Table III is the average of fifty executions and in general the response time of a testable web service was greater than the response time of a regular web service. However, there were specific executions in that the response time of the testable version execution was smaller than the response time of the regular version. This happens because in some executions the overhead due to the network was greater than the overhead brought by the testing code. Thus we consider that the testing code overhead is worthless for non critical SOA applications.

Another issue we found is related to the metadata. There can

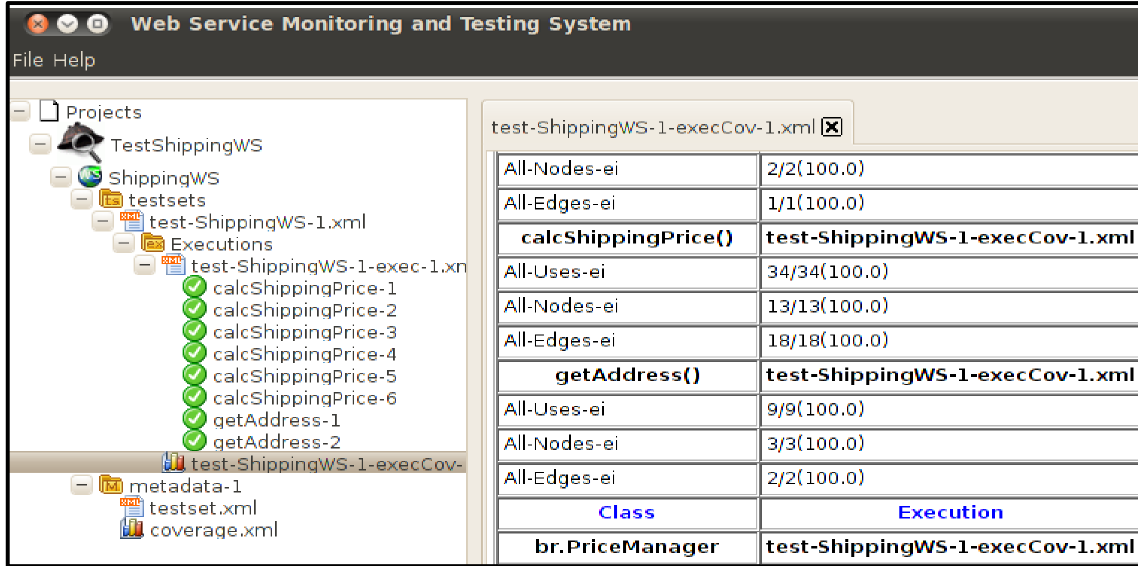


Fig. 10. ToolSupport

TABLE III
OVERHEAD ANALYSIS

| Web service Version | Average time | Overhead |
|---|---|---|
| Non-testable | 2070 | 0% |
| Testable | 2125 | 2.65% |
| Testable (in test session mode) | 2179 | 5.26% |

be testable web services without any metadata or the metadata suggested may not be enough to help testers improve their test set. We are performing a study regarding this issue to come out with a better solution soon.

Despite these drawbacks, we believe that the proposed approach does not characterize a very hard architecture and the overhead is minimum. The high availability of an implementation of TestingWS is not very difficult to accomplish and, if it becomes unavailable sometime, the only operation of a testable web service that would not work is the `getCoverage` operation. All other operations does not depend on TestingWS. Even if at this moment testers have few information to improve their test set, at least they have a perspective on how they are exercising the web service under test.

## V. Related Work

A few proposals for structural testing of web services have been found in the literature, but we only discuss those ones that are more related to our work. Bartolini et. al [15] proposed a web service called TCov. The developer should manually instrument the service or composition of services to be tested and insert calls to TCov to record information of execution. Every time the instrumented service runs, details of the execution will be recorded in TCov. Thus, the client using the service can run test cases and query TCov to obtain recorded data. For any coverage analysis, the client should use the data collected from TCov and do it manually or either use an existing tool or develop one to do this. In our approach, the instrumentation is done automatically as well as the coverage analysis and BISTWS also supports testers with metadata.

Endo et. al. [17] proposed applying the model PCFG (Parallel Control Flow Graph) for testing Web Services compositions represented in BPEL. A PCFG is a model to capture the control flow, data flow and communication in parallel programs based on message exchange. A PCFG is composed of several processes that have their own control flow (CFG - Control Flow Graph). The ValiBPEL tool automates the approach and use structural testing criteria such as all-nodes, all-edges and all-uses.

Karam et al. [23] introduced a formal model to map the structure of compositions of web services under testing. Each service of the composition is called a node and transactions between web services are the edges, so graph techniques are used to derive test cases to cover structural criteria, as all-nodes and all-edges, for example.

The work of Karam et. al. [23] and Endo et. al. [17] is similar to our approach as both use structural criteria for testing web services. The focus of their work, however, is only on supporting the developer on testing a BPEL process before publishing it, because it is necessary to have access to the code to derive the model proposed. Moreover, the emphasis is on testing the composition, and the web services used in the composition are seen as black-box.

## VI. Concluding Remarks

This paper presented the BISTWS approach to apply structural testing in the context of SOA testing. The main idea is to support developers on creating testable web services with structural testing facilities to their clients. Testers can set the

testable web service to a test session mode, run test cases and get a coverage analysis report on structural testing criteria. Testers can also use the metadata of the testable service to evaluate and improve the coverage achieved.

The approach proposed is generic and introduces the idea of improving web service testability through structural testing capabilities. The implementation of the approach defines which programming languages will be supported on generating testable web services. The particular implementation also affects how the instrumentation is done, which structural criteria are supported and which operations are inserted into the testable web service.

We have also shown an implementation of the approach using a web service for structural testing of Java web services called JaBUTiWS. We also presented a short usage example. JaBUTiWS only generates web services in Java but they still comply with web services standards. Applying the BISTWS approach to other languages requires the implementation of a tool to instrument the web service and to insert the operations to easy the structural testing from an external context.

JaBUTiWS does not handle non-functional issues and does not support integration testing currently. Compositions written in Java can be instrumented by JaBUTiWS, but the web services used in the composition would not be instrumented. Even if the web services used in the composition are testable web services, the structural criteria and the coverage analysis implemented by JaBUTiWS would not cover the integration among the testable web services.

As future work we intend to design other implementations of TestingWS. We plan to convert the ValiBPEL tool [17] into a web service and integrate it with JaBUTiWS. This combination would allow the generation of testable workflow BPEL. It could also be explored the integration between the workflow and the testable web services used as clientes. We will also perform further evaluation of the BISTWS approach to formally evaluate the advantages of using a testable web service instead of using a regular web service. Moreover, we plan to investigate how structural testing facilities could help to monitor and to certificate web services.

There are also some improvements we want to do in the JaBUTiWS implementation: use test requirements as metadata and allow testers to relate which test requirements were covered by which test case and which test requirements still need to be covered; implement non-functional requirements, such as concurrency control, authentication and security issues; perform a detailed study on how and which metadata would be useful to help testers improve their test set after achieving a coverage analysis report.

## VII. Acknowledgements

The authors would like to thank the Brazilian funding agencies: FAPESP (process 2008/03252-2), CAPES and CNPq for their financial support.

## References

[1] M. P. Papazoglou, P. Traverso, S. Dustdar, F. Leymann, and B. J. Krämer, "Service-oriented computing: A research roadmap," in *Service Oriented Computing*, 2005.

[2] M. H. Mustafa Bozkurt and Y. Hassoun, "Testing web services: A survey," Department of Computer Science, King's College London, Tech. Rep. TR-10-01, January 2010.

[3] L. O'Brien, P. Merson, and L. Bass, "Quality attributes for service-oriented architectures," in *Proc. of the Int. Workshop on Systems Development in SOA Environments*, 2007, p. 3.

[4] W. T. Tsai, J. Gao, X. Wei, and Y. Chen, "Testability of software in service-oriented architecture," in *Proc. of the 30th Annual Int. Computer Software and Applications Conf.*, 2006, pp. 163–170.

[5] G. Canfora and M. Penta, "Service-oriented architectures testing: A survey," pp. 78–105, 2009.

[6] H.-G. Gross, *Component-Based Software Testing with UML.* Springer, 2005.

[7] E. J. Weyuker, "Testing component-based software: A cautionary tale," *IEEE Softw.*, vol. 15, no. 5, pp. 54–59, 1998.

[8] D. Brenner, C. Atkinson, O. Hummel, and D. Stoll, "Strategies for the run-time testing of third party web services," in *SOCA '07: Proc. of the IEEE Int. Conference on Service-Oriented Computing and Applications*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 114–121.

[9] X. Bai, W. Dong, W.-T. Tsai, and Y. Chen, "Wsdl-based automatic test case generation for web services testing," in *Proc. of the IEEE Int. Workshop on Service-Oriented System Engineering*, 2005, pp. 215–220.

[10] C. Keum, S. Kang, I.-Y. Ko, J. Baik, and Y.-I. Choi, "Generating test cases for web services using extended finite state machine," in *Proc. of the 18th Int. Conf. on Testing Communicating Systems*, 2006, pp. 103–117.

[11] W. T. Tsai, R. Paul, Y. Wang, C. Fan, and D. Wang, "Extending wsdl to facilitate web services testing," in *Proc. of the 7th IEEE Int. Symposium on High Assurance Systems Engineering*, 2002, p. 171.

[12] X. Bai, Y. Wang, G. Dai, W.-T. Tsai, and Y. Chen, "A framework for contract-based collaborative verification and validation of web services." in *Proc. of the 10th Int. Symposium on Component-Based Software Engineering*.

[13] R. Heckel and L. Mariani, "Automatic conformance testing of web services," in *Proc. of the 9th Int. Conf. on Fundamental Approaches to Software Engineering*, 2005, pp. 34–48.

[14] C. Bartolini, A. Bertolino, E. Marchetti, and I. Parissis, "Data flow-based validation of web services compositions: Perspectives and examples," pp. 298–325, 2008.

[15] C. Bartolini, A. Bertolino, S. Elbaum, and E. Marchetti, "Whitening soa testing," in *Proc. of the 7th Joint Meeting of the European Software Engineering Conf.*, 2009, pp. 161–170.

[16] W.-L. Dong, H. Yu, and Y.-B. Zhang, "Testing bpel-based web service composition using high-level petri nets," in *Proc. of the 10th IEEE Int. Enterprise Distributed Object Computing Conf.*, 2006, pp. 441–444.

[17] A. T. Endo, A. S. Simão, S. R. S. Souza, and P. S. L. Souza, "Web services composition testing: a strategy based on structural testing of parallel programs," in *Proc. of the Testing: Academic and Industrial Conf. - Pratice and Research Techniques*, 2008.

[18] L. Li, W. Chou, and W. Guo, "Control flow analysis and coverage driven testing for web services," in *Proc. of the IEEE Int. Conf. on Web Services*, 2008, pp. 473–480.

[19] L. Mei, W. Chan, and T. Tse, "Data flow testing of service-oriented workflow applications," in *Proc. of the 30th Int. Conf. on Software Engineering*, 2008, pp. 371–380.

[20] A. Bertolino and A. Polini, "Soa test governance: Enabling service integration testing across organization and technology borders," in *Proc. of the IEEE Int. Conf. on Software Testing, Verification and Validation*, 2009, pp. 277–286.

[21] M. M. Eler, A. T. Endo, P. C. Masiero, M. E. Delamaro, J. C. Maldonado, A. M. R. Vincenzi, M. L. Chaim, and D. M. Beder, "Jabutiservice: A web service for structural testing of java programs," in *Proc. of the 33rd Annual IEEE Software Engineering Workshop*, 2009.

[22] A. M. R. Vincenzi, M. E. Delamaro, J. C. Maldonado, and W. E. Wong, "Establishing structural testing criteria for java bytecode," *Software Practice & Experience*, vol. 36, no. 14, pp. 1513–1541, 2006.

[23] M. Karam, H. Safa, and H. Artail, "An abstract workflow-based framework for testing composed web services." in *Proc. of the 5th Int. Conf. on Computer Systems and Applications*.