Web Service Slicing: Intra and Inter-Operational Analysis to Test Changes

Article *in* IEEE Transactions on Services Computing · March 2018

DOI: 10.1109/TSC.2018.2821157

CITATIONS
8

READS
205

2 authors:

Animesh Chaturvedi
Indian Institute of Information Technology, Design and Manufacturing Jabalpur
17 PUBLICATIONS **112** CITATIONS
SEE PROFILE

David Binkley
Loyola University Maryland
184 PUBLICATIONS **8,028** CITATIONS
SEE PROFILE

Web Service Slicing: Intra and Inter-Operational Analysis to Test Changes

Animesh Chaturvedi and Dave Binkley

Abstract— We introduce *Web Service Slicing,* a technique that captures a functional subset of a *large-scale web service* using an *interface slice* captured as a *WSDL slice* (a subset of a service's WSDL). An interface (WSDL) slice provides access to an *interoperable slice*, which is a functional subset of the service's code. The technique uses *intra-operational* and *inter-operational analysis* to identify web service changes. With the aid of an *associative code-test mapping,* we leverage the identification of affected operations to reduce the cost of web-service regression testing by extracting a subset of the existing test cases. Used in conjunction with a web service slice, this subset reduces the cost of web-service regression testing by enabling the running of fewer tests. Furthermore, we exploit two approaches: *Operationalized Regression Testing of Web Services* (ORTWS) and *Parameterized Regression Testing of Web Services* (PRTWS). ORTWS effectively tests *intra-operational changes* at the WSDL and WS-code levels, while PRTWS tests *inter-operational changes* involving *inter-operational dependencies* due to *primary parameters*. Finally, we present results obtained using our prototype implementation, *AWSCM* (*Automatic Web Service Change Management*), in two case-study experiments that serve to illustrate the reduction potential of the technique using eight real-world web services.

Index Terms— web services, software analysis, program slicing, software maintenance, and testing tools

1 Introduction

A web service is a composable unit of a web application that supports interoperable machine to machine interactions with other applications. While a web service can stand alone, more often it is part of an integrated web application composed of a collection of web services communicating using a protocol such as SOAP. Generally, web service procedures are referred as *operations* when they are part of a web service's public interface. The operations provided by a web service are discrete pieces of functionality often accessed using remote procedure calls.

The public interface to a web service is commonly described using the XML-based *Web Service Description Language* (WSDL) to define the exposed operations. Thus, a web service consists of two major artifacts: the WSDL used to describe its interface and the web-service code (WS code) used to implement its business logic. Typically, each web service is independently developed and tested. A web service can be implemented in a variety of ways with object-oriented languages such as Java and C++ meshing well with the web interface.

Since their inception, web services have grown in size, which leads to the growth of ubiquitous systems [1]. Such evolving ubiquitous systems require increasing maintenance and evolution effort. A critical part of maintenance is *regression testing,* which is used to revalidate a system after a change.

To address the need for regression testing cost reduction, we employ the well-known concept of program slicing [2][3]. For program *P*, a program slice is a subprogram of *P* that captures a subset of *P's* behaviour. Program slicing has seen prior application in reducing the cost of regression testing [4]. Specifically, we are interested in techniques akin to interface slicing [5], which was introduced before the advent of web services to slice languages where a module includes a public *interface* and a private *implementation*. We employ interface slices of a web service to access subsets of its operations.

In addition to the web service slice, we describe an approach to reducing the cost of web-service regression testing by executing a subset of the service's test cases for a web service slice. Unfortunately, regression testing large systems is often quite expensive [6]; this is clearly true in case of large-scale web services. Fortunately, *change analysis* can be used to reduce regression-testing effort [7][8]. In the case of web services, we can exploit changes made to both the web service code and its WSDL description. We determine the subset service by computing a subset of its WSDL (as interface slice) using automatically captured change impact analysis.

We aim to improve state-of-the-art by applying the fundamental concepts of program slicing and regression testing to web services. We use two concepts: *intra-procedural analysis* and *inter-procedural analysis* [9][10] as two levels of source code analysis. *Intra-procedural analysis* inspects a single procedure, while *inter-procedural analysis* inspects relationships between procedures. These two can be used to compute program slices and retrieve test-case subsets in the context of web service cost reduction.

- *Animesh Chaturvedi is with the Indian Institute of Technology Indore, Indore, MP, India. E-mail: animesh.chaturvedi88@gmail.com.*
- *Dave Binkley is with the Loyola University Maryland, Baltimore, MD, USA. E-mail: binkley@cs.loyola.edu.*

We introduce *web service slicing*, which is built upon *intra-operational analysis, inter-operational analysis*, and *interface slicing*. In this paper, we re-invent existing program analysis and slicing concepts for web-service analysis and slicing. This paper also fills a gap between change analysis and regression testing of web services. We explain how to execute a subset of a web service using a subset of its test cases to reduce costs. This paper makes the following four contributions, which we discuss in the next four sections. These are followed by a discussion of related work and a brief summary.

- Section 2 defines and describes key concepts. It then provides an illustrative example that aims to provide intuition behind the definitions.

- Section 3 presents two techniques for analysing differences between two versions of a web service: *Intra-Operational Change Analysis* and *Inter-Operational Change Analysis*. The first of these is useful in the analysis of a single operation while the latter aids in understanding the interactions between operations.

- Section 4 proposes two regression testing cost reduction techniques: *Operationalized Regression Testing of Web Service* (ORTWS), which is based on *Intra-Operational Change Analysis,* and *Parameterized Regression Testing of Web Service* (PRTWS), which is based on *Inter-Operational Change Analysis*. Based on the underlying change information, both techniques identify a subset of the system's test cases. This section also describes the implementation of these two techniques in our tool AWSCM.

- Section 5 presents case studies involving eight real world web services. These case studies highlight the impact of the proposed techniques and shows how they reduce regression-testing effort.

2 Proposed Definitions

This section provides six foundational definitions: *intra-operational analysis*, *inter-operational analysis*, *subset service*, *web service slicing*, *interoperable slice*, and *interface slice.* At the end of the section, we provide an illustrative example to provide intuition behind the definitions. We first define intra-operational and inter-operational analysis.

Definition 1: An ***intra-operational analysis*** is an intra-procedural analysis that considers analysis of code within the code of operation.

Intra-operational analysis can be used to identify affected operations that are directly modified. In contrast, to identify operations that are affected because they depend on a change in some other operation or procedure, this case requires *inter-operational analysis*.

Definition 2: An ***inter-operational analysis*** is an inter-procedural analysis that considers analysis of *affected operations* depending upon other operations or procedures.

Next, we define *subset service*, which supports our definition for *web service slicing*.

Definition 3: A ***subset service*** is a subset of a web service that provides only some of the original service's functionality.

Definition 4: ***Web service slicing*** is a variation of program slicing that extracts a subset of a web service that provides a subset of the original service's behaviour. The resulting slice is referred as a ***web service slice***.

Web service slicing helps address the problem of large-scale web service maintenance by identifying *subset services,* which capture the changed portions of the system. These *subset services* are useful in the following four situations. First, consider *subset testing,* which aims to reduce maintenance cost by selecting a subset of the test cases. Doing so avoids the running (or re-running) of unnecessary test cases. Second, the formation of a *subset client* aims to access only a limited subset of an existing service's functionalities. Third, *subset calling* aims to capture the subset of a service required to support a specific calling service. Finally, *subset reengineering* aims to re-engineer an existing large service to create a smaller service. Next, we refine web service slicing to consider slicing the service's code and then its interface.

Definition 5: An ***interoperable slice*** is a portion of the web service code that can be accessed as a *subset service* from a remote client that needs a subset functionality.

An interoperable slice is different from web service code slice (or web service program slice) because we only consider slicing with respect to operations accessible from a remote location. A web service is accessible through an interface (e.g., that described by WSDL in a SOAP based web service). Commonly its code is unavailable, which makes an interoperable slice useful. Finally, we define slicing at the interface (WSDL) level.

Definition 6: ***Interface slicing*** is a variation of program slicing that extracts a subset of a given interface's operation that captures a subset of the original interface's behaviour. The resulting slice is referred as an ***interface slice*** containing a subset of interface's operations.

These six definitions are equally applicable to SOAP/WSDL and REST based web services. However, in the remainder of the paper, we restricted our study to SOAP/WSDL based web services. In doing so, we use interface slices, Definition 6, to extract the set of operations from a given WSDL (a given interface) to produce a ***WSDL slice***.

An *interface slice* provides access to a *subset service* by providing an interface to the corresponding *interoperable slice*. Thus, a *WSDL slice* provides the interface to a *subset of a web service* where the corresponding *interoperable slice* captures the corresponding subset of the code. An *interface slice* (a *WSDL slice* in the form of a Subset WSDL [11]) is designed to provide access to an interoperable slice. The re-implementation of the interoperable slice is not needed because the WSDL slice (or Subset WSDL) provides access to subset of the functionality originally available (or deployed). Thus, the subset (or interoperable slice) of the original web service can be accessed over a network through the WSDL slice. This option is pragmatic when the code for the service is unavailable.

An interface slice helps an engineer during development and testing by providing focus on a portion of a large service. The approach is viable because large ser-

vices are decomposable into smaller services. Accessing only a subset provides clarity to the testing process where we focus testing on relevant subsets of a larger web service. Slices of the web service code including *subsets of the service's interfaces,* which help in *subset service formation*, test selection, and test execution. For example, we can use the *Subset WSDL* to select a subset of the test cases that tests the corresponding subset of the code.

The remainder of this section presents a motivational example that aims to provide some intuition for the definitions and how they support our overall approach. The example, shown in Figure 1, illustrates changes using two versions of a Banking Web Service. In Version 1, the operation 'deposit' indirectly depends upon the operation 'balance'. Here, there exists an *inter-operational dependency* of 'deposit' on 'balance'. Suppose the bank decides to change the functionality by deleting the operation 'balance' and inserting the operation 'statement'. This change (see Version 2) affects the operation 'deposit' due to an indirect call to a modified procedure 'showBalance' (via 'creditMoney' and 'updateBalance').

The *intra-operational analysis* uncovers the removal of the operation 'balance' and the addition of the new operation 'statement'. In contrast, *inter-operational analysis* uncovers the indirect changes in the code for the operation 'deposit', which depends on the procedure 'useBalanceOp' whose name was modified to 'updateBalance'.

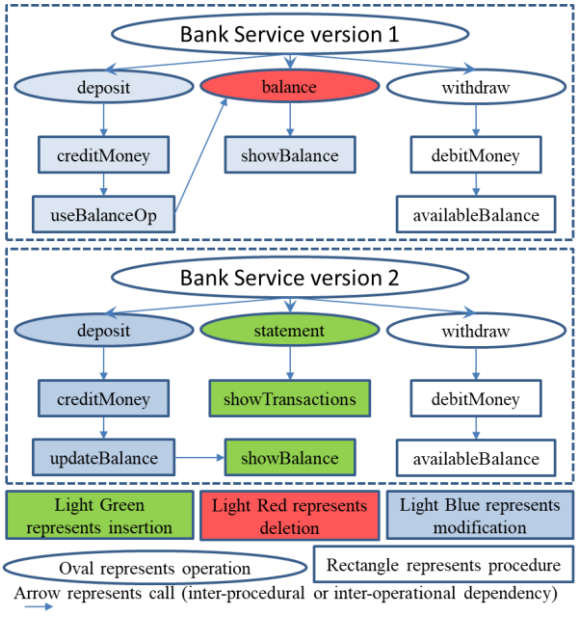

Fig. 1. The evolution of a Banking Web Service modelled as a dependence graph. The figure provides an abstract view of the following three changes. (i) The operation 'balance' was deleted, thereafter, the operation 'statement' along with the procedure 'showTransactions' were inserted. (ii) The procedure "useBalanceOp" is modified by renaming it "updateBalance", this affects (modifies) procedure 'creditMoney' and transitively the operation 'deposit'. (iii) This leads to modification of the procedure "showBalance". All the changes are highlighted with colour that depicts: insertion (light green), deletion (light red), and modification (light blue).

Figure 2 illustrates a web service slice using the Banking example of Figure 1. This slice is a combination of an *interface* (*WSDL*) *slice* (a subset of the WSDL) and its corresponding *interoperable slice* (a subset of the code).

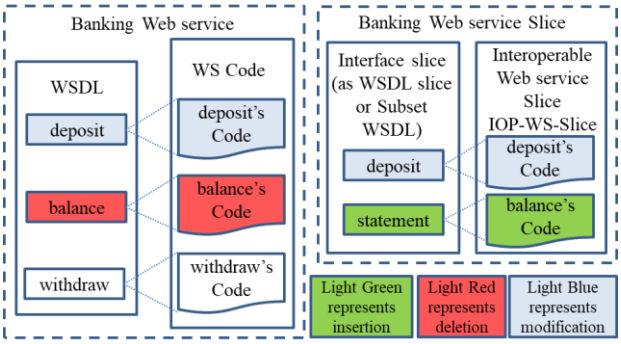

Fig. 2. A web service slice (combining an *interface slice* and an interoperable slice) of the Banking Web Service.

The evolution of a web service brings the need for *change analysis* based *regression testing*. The rest of this paper builds on these definitions using two approaches to change analysis of a web service.

3 Intra / Inter-Operational Change Analysis

This section describes two proposed intra-operational and inter-operational change analysis to analyse behaviour of a changed operation. The two approaches are as follows.

(i) *Intra-Operational Change Analysis* identifies operations whose code underwent changes. For example, in Figure 1, the operation 'statement' is new and thus an example of a changed operation.

(ii) *Inter-Operational Change Analysis* identifies affected operations that depend on changed operations and procedures. For example, the operation 'deposit' depends on the affected procedure 'creditMoney' (which calls the modified procedure 'updateBalance' in Version 2).

Intra-Operational Change Analysis effectively identifies *intra-operational changes* between two versions of the WSDL and WS code. The result is three subsets of the WSDL. The first two, the *Difference WSDL* (DWSDL) and the *Unit WSDL* (UWSDL), capture changes in the WSDL and changes in the WS code, respectively. The third *Subset WSDL*, the *Reduce WSDL* (RWSDL), contains user-selected operations, which enables the user to force the tool to focus on specific parts of the code. The three *Subset WSDL*s are combined to form the *Combined WSDL* (CWSDL). The operations of the *Combined WSDL* are unique (i.e., an operation is included only once, even if it is found in more than one of the three subsets).

Inter-Operational Change Analysis captures *inter-operational changes* at the code level using test data to examine the flow of control and data within an operation. It is applicable to special cases of the operational behaviour where the input parameters of an operation are dependent upon each other. Inter-Operational Change Analysis first identifies test data that leads to code flow changes and then creates a Subset WSDL, referred as the *Parameter WSDL* (PWSDL), which captures these changes. Specifically, the PWSDL captures the insertions and modifications made at the WS code level.

Inter-Operational Change Analysis is analogous to a dynamic database where an operation is analogous to a relation, a message (a web service call) is analogous to a

query, and the parameters of the operation are analogous to the attributes of the relation. Therefore, a testing tool determines the input *query* and the output *result* by the input *parameter values* and output *result of the operation*.

Inter-Operational Change Analysis is inspired by Codd's rules [12], in particular, the notion of a *primary key* in a database management system. Its initial step is to identify the *primary parameters* of an operation manually. Just as the *key attributes* in a relational database uniquely identify other *attributes*, the *key parameters* of an operation uniquely identify the other parameters. In some cases, a collection of parameters can play the role of a *primary parameter*. The parameters that do not participate in the formation of the *primary parameter* and that depends on the *primary parameter* are referred as *non-primary parameters*. When an operation has two or more *candidate primary parameters*, one of them is chosen as the primary parameter and the remainder become *non-primary parameter(s)*.

Table 1 shows the types of changes captured by the *Subset WSDLs*. Current APIs are restricted to regenerate the WSDL from the code or refactored code, whereas we present two approach to generate WSDL due to changes done between two versions of a web service. The discrimination between changes done to the WS interface and code is useful because of the two reasons. First, when the WS interface is changed, then interface call need to be synchronized. Second, when the internal business logic is changed, then the messaging-code to call an operation needs to be synchronized. Thus, both type of changes require maintenance effort to continue the service accessibility. Both kinds of the change analysis is helpful to identify changed operations that needs regression testing.

4 Operationalized and Parameterized Regression Testing of Web Services

This section describes the use of change analysis to support web-service regression testing cost reduction. Our two techniques (ORTWS and PRTWS) depend on an *associative code-test mapping* that maps the code to its corresponding test cases. We discuss this mapping in Subsections 4.1 and 4.2. To support this mapping, we designed test suites such that their test cases are associated with particular portions of the corresponding web service code. Thereafter, we identify changes in web services and then map these changes to their subsets of the test cases using the associative mapping.

The Figure 3 shows that the two complimentary approaches ORTWS and PRTWS; both have two steps. The first step is to identify a set of changed operations and their corresponding *Subset WSDLs*. The second step maps these operations to test-cases of the old test-suite using the associative code-test mapping. Executing the service with this test-case-subset reduces regression-testing effort, and thus testing cost.

Table 1 provides an overview of the intra-operational and inter-operational change analysis used in support of web-service regression testing. The table presents five columns: the type of each change, discriminated by the two levels (WSDL and WS code), the location and purpose of the change, the *Subset WSDL* that captures the change, and how to perform regression testing.

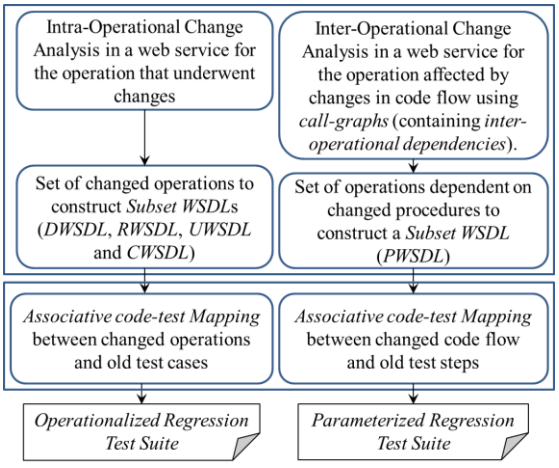

Fig. 3. An overview of ORTWS and PRTWS that use intra-operational and inter operational change analysis, respectively.

TABLE 1 CHANGE BASED WEB-SERVICE REGRESSION TESTING

Type (or label)	Level	Location and purpose	Captured in Subset WSDL	Regression testing technique
Insertion	WSDL	Insertion of an operation in WSDL or Datatype in XSD	DWSDL RWSDL	Test case template require suitable test data
	WS Code	Insertion of code for new operation	UWSDL PWSDL	
Deletion	WSDL	Deletion of operation from WSDL to make it disfunctional to client	None, because these changes are rare and unusual.	Failure of test cases will test the deletion
	WS Code	Deletion of operation from WS code to make it disfunctional at WSDL level		
Modification	WSDL	Modification of XSD	DWSDL RWSDL	Operationalized testing
	WS code	Modification at code lines without affecting WSDL	UWSDL PWSDL	Code flow or Parametrized testing

The next two sub-sections explain how *Intra-Operational Change Analysis* forms the core of ORTWS, which captures a set of changed operations, and how *Inter-Operational Change Analysis* forms the core of PRTWS, which captures changed control (or data) flow in code.

4.1 Operationalized Regression Testing of Web Services (ORTWS)

To begin with, ORTWS captures a set of changed operation based on three kinds of changes: insertions, deletions, and modifications. The idea is to design a test suite such that each test case holds test data to validate an operation or a procedure. The process exploits intra-operational changes occurring in two places – in the WSDL and in the WS code. Figure 4 illustrates the construction of the four Subset WSDLs and the *Operationalized Regression Test Suite*.

ORTWS extracts a subset of the test cases from the existing test suite in the following three ways. First, *DWSDL*, *RWSDL,* and *UWSDL* capture newly inserted operations that need to be tested during regression testing. Newly generated *test-case templates* can be used to test such operations. These templates are empty test cases for the newly inserted operations. For inserted operations, we use a testing tool (such as JMeter or SoapUI) that can automatically generate test-case templates with given signatures (names and parameters) of each new operation found in the WSDL. A tester can then fill each template with suitable test data.

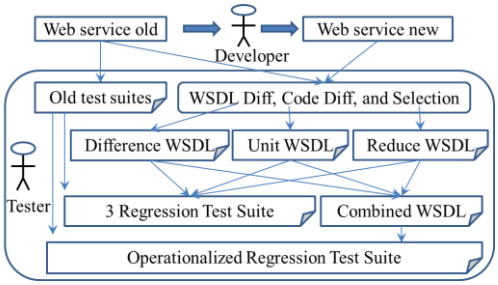

Fig. 4.The *Intra-Operational Change Analysis* based ORTWS constructs four Subset WSDLs (DWSDL, UWSDL, RWSDL, and CWSDL) and an Operationalized Regression Test Suite.

Second, deleted operations are not present in the *Subset WSDL* because they are not present in the new version of the web service. Thus, their respective test cases are expected to fail during regression testing. The failures attest to the deletion of the operation. Thereafter, these test cases can be removed.

Third, modifications at the WSDL level involve changes in the port, binding, or in the XSD where the input, output, or both may be changed. The *DWSDL* and the *RWSDL* capture such changes. Modifications at the WS code level are captured in the *UWSDL*. Based on these three, we extract test cases from the test suite of the old web service using the associative code-test mapping.

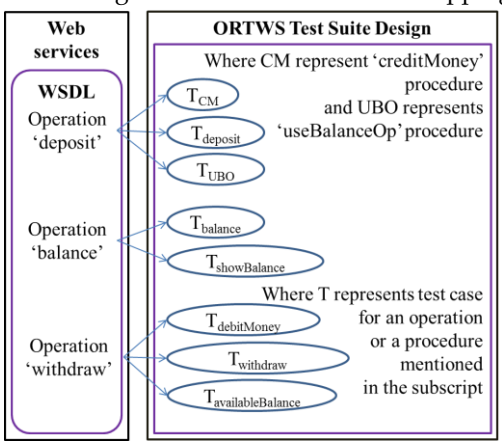

Fig. 5. Example of *associative code-test mapping* between the operations 'deposit', 'balance', and 'withdraw' in a WSDL with its Test cases (T_{CM}, $T_{deposit}$, T_{UBO}), ($T_{balance}$ and $T_{showBalance}$), and ($T_{debitMoney}$, $T_{withdraw}$ and $T_{availableBalance}$) respectively. Each test case is designed in the test suite for the operation or its procedure.

Additionally, some selective operations, chosen by the user, are captured by the RWSDL. Finally, we form the *CWSDL* as the union of the operations in the three subset WSDLs: DWSDL, RWSDL, and UWSDL.

Figure 5, illustrates the process using the Banking Web Service example from Figure 1. It shows the mapping between three web service operations and their test cases. The test cases corresponding to the changed operations are selected to form the *Regression Test Suite*.

ORTWS works well because usually web service operations are loosely coupled. Hence, a web service operation can be tested independently. In the next subsection, PRTWS considers web service operations that invoke another operation or procedure.

4.2 Parameterized Regression Testing of Web Services (PRTWS)

The second approach, PRTWS, identifies a reduced test set by exploiting the breakdown of parameters into *primary parameters* and *non-primary parameters* to focus testing on specific code. The idea is to design a test suite such that each test case holds the test data for a *fixed* value of the primary parameter, and then consider different values for the non-primary parameters. Such test-case examines focused data and control flow within the operation's code.

For example, suppose an operation is too complex to test directly because the test cases for the operation are too complex. To simplify this situation, the tester can design an entire test suite dedicated to a single operation. In this test suite, each test case is designed for a fixed value of the *primary parameter*. This makes it possible to identify test steps that test individual code flows of an operation by executing the operation using a fixed value of the *primary parameter*.

PRTWS is helpful in testing *service interactions*, which occur when an operation of one web service calls an operation of another. PRTWS is also helpful in testing *subset-service interactions* by analysing the subset interaction between two web services.

Figure 6 shows an overview of PRTWS that first identifies inter-operational changes using Inter-Operational Change Analysis based on inter-operational dependencies and inter-procedural dependencies. The next step maps the identified changes to the call graph in order to find a set of change-affected operations from which Inter-Operational Change Analysis generates two outputs: the PWSDL and a Parameterized Regression Test Suite. The PWSDL contains a set of change-affected operations used to identify test cases (made up of test–steps) from the old test suite. These tests are identified with the help of the associative code-test mapping.

In greater detail, the associative code-test mapping between *test steps* and *operation code* is used to map each flow path to one of the test steps. Here, we exploit a finer-grained level associative code-test mapping built from the individual test steps that make up a test case. Each test step is mapped to specific control and data flow within an operation. This mapping enables us to select those test steps required to exercise the identified changes in the code flows. The identified test steps are then combined to form test cases.

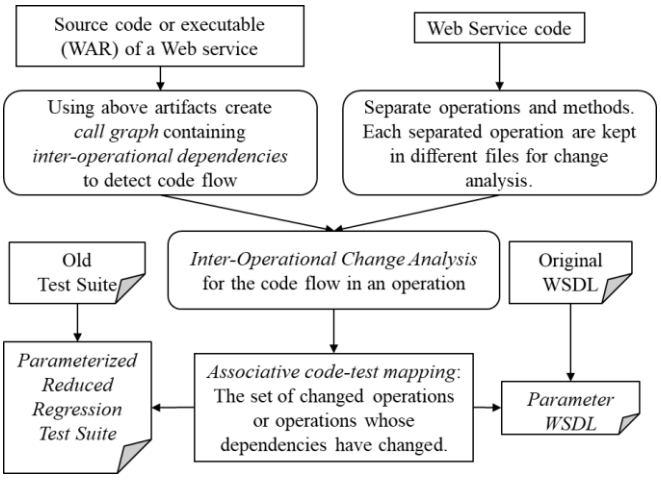

Fig. 6. The Inter-Operational Change Analysis based PRTWS constructs the PWSDL and the Parameterized Regression Test Suite.

To illustrate, Figure 7, continues the Banking Web Service example. It shows the designing of a test suite with test cases for a fixed value of a primary parameter. Each test step involves a different operation code flow using different parameter values. By selecting certain test steps, a tester can test their corresponding code flows.

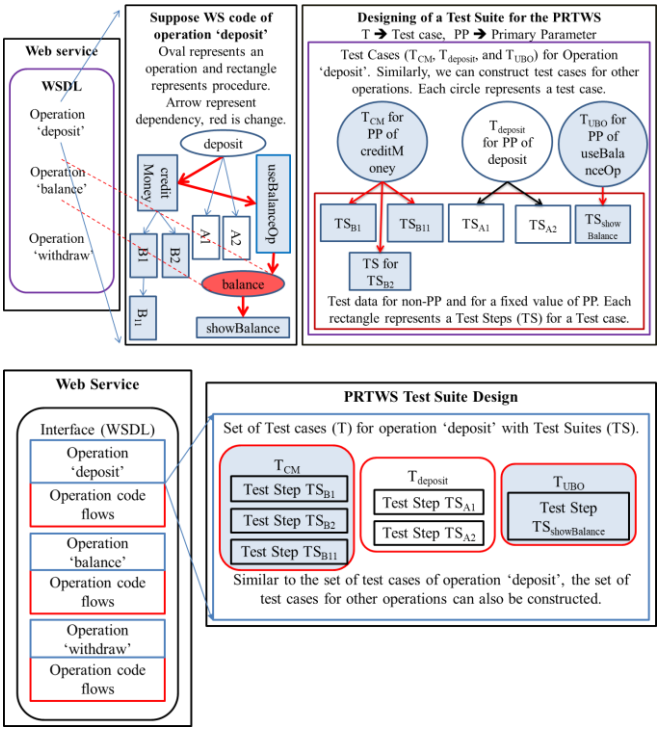

Fig. 7. An example of the *associative code-test mapping* used to design a PRTWS test suite, where each test case is designed for a fixed value of the *Primary Parameter* (PP). Each test step then maps to a specific code flow. Test cases that must be re-run are show in light blue.

Figure 7 assumes the following three code flows. First, the procedure 'creditMoney' calls the procedure 'useBalanceOp', which then calls the operation 'balance' that further calls the procedure 'showBalance'. Second, the operation 'deposit' executes procedures A_1 and A_2. Third, the procedure 'creditMoney' executes procedure B_1, which calls procedure B_{11} and B_2. The three code flows lead to a test suite design where 'creditMoney', 'deposit', and 'useBalanceOp' are tested by test cases T_{CM}, $T_{deposit}$ and T_{UBO}, respectively, which contain test steps (TS_{B1}, TS_{B11} and TS_{B2}), (TS_{A1}, TS_{A2} and TS_{A12}) and $TS_{showBalance}$, respectively. The calls highlighted in red depict changed code flows and thus identify the *affected operations* from which we construct the PWSDL (for the operation 'deposit') and the Parameterised Regression Test Suite (composed of T_{CM} and T_{UBO}). Thus, PRTWS constructs a new test suite from the change information and the old test suite.

4.3 AWSCM

In this subsection, we describe our tool AWSCM [13][14] that has two recursive sub-tasks, both illustrated in Figure 8. This section explains these two steps, the tool's internals, and the current automation-level of AWSCM.

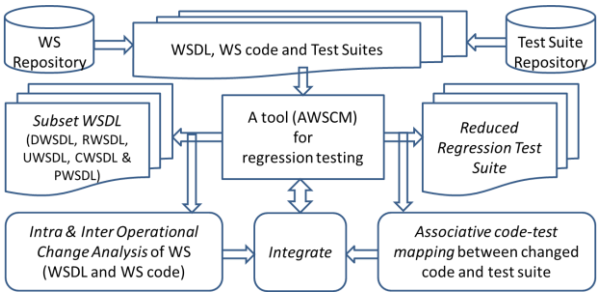

Fig. 8. AWSCM: a tool for change analysis based regression testing of web service.

In the first step, AWSCM extracts change information by capturing four sets of changed operations from a web service to create the four *Subset WSDLs*. To extract changes made to a web service, AWSCM uses *semantic differencing* applied to an old and a new version of the WSDL and the WS code. AWSCM does this using two different *semantic differencing* tools, *JDiff* [51] and *Predic8* [52]. AWSCM uses *JDiff* to extract difference between the WS code of the old and new versions. It uses *Predic8* to identify WSDL differences. For example, suppose a web service changes from version WS_1 to version WS_2. Using Intra-Operational Change Analysis, we can capture the changed operations that underwent changes at the code-level between WS_1 and WS_2. AWSCM identifies these changed control and data flows between WS_1 and WS_2.

To compliment this semantic differencing, AWSCM makes use of PRTWS's two change identification methods: *automated change analysis* and *manual query*. It first uses the results of change analysis to automatically select inputs that test the changed operations. The second method is a semi-automatic variant in which a tester manually selects test steps that execute a desired code flow of an operation.

In the second step, AWSCM exploits the associative code-test mapping between the source code and the test cases to identify test cases that should be re-run. To do so, AWSCM automates the process of *reducing the cost of web-service regression testing*. AWSCM selects test cases from the test suite of WS_1 to construct the *Regression Test Suite* for WS_2. Using a subset of the test cases from an old test suite, AWSCM constructs *Operationalized Regression Test Suites* and *Parameterized Regression Test Suites*.

Internally, AWSCM makes use of *text mining* to extract tokens from the input WSDL and source code. *Text mining* includes *pattern recognition* and *text manipulation*. Pattern recognition supports the discovery of meaningful parts of an input text. Text manipulation supports the removal, replacement, and insertion of text. Regular expressions can be used to support both. AWSCM uses Java regex [53] to extract the relevant parts of the WSDL, the WS code, and test suites.

The relationship between inter-procedural dependencies [9] can be represented as a *call graph* (or a data dependence graph); this leads to call graph analysis [10] such as call graph change detection. AWSCM uses a web archive (WAR) of a web service to extract the *call graph* information to extract inter-operational dependencies and inter-procedural dependencies. It then extracts the call graph using Gousios [54] (using the library java.util.jar).

AWSCM gives a visual report containing all changed lines of code. Based on this report, a tester can inspect the code and identify subsets of the test steps. From these test steps, AWSCM can then construct the reduced regression test suite, such that it executes the subset of the service that underwent change.

Now, we will describe the extent of automation provided by AWSCM. While AWSCM can take advantage of web-service code, it can also work without any knowledge of the web-service code. This flexibility facilitates standard WSDL based web-service testing to perform efficient web-service regression testing. The current AWSCM prototype implements the two proposed change-analysis techniques. While integration is part of our plans, AWSCM is not presently 100% automated, it still needs to be integrated with a standard testing tool such as SoapUI or JMeter. In support of the case studies presented in Section 5, we manually export test suites from SoapUI and JMeter.

In summary, AWSCM uses intra-operational and inter-operational analysis to identify changes at both the WSDL and WS-code levels. To aid in establishing the associative code-test mapping, we designed the test suites formally to supports ORTWS and PRTWS. Based on this mapping, AWSCM selects test cases associated with the changes made to a web service. Hence, AWSCM uses the identified set of affected operations to construct a reduced regression test suite for regression testing.

5 EXPERIMENTS: CASE STUDIES ON WEB SERVICES

This section first explores the performance of AWSCM as it constructs the *Subset WSDL*s and then the *Reduced Regression Test Suites*. It does so using two experiments involving eight web services. It then considers observations and findings based on the experiments. The main objective behind the experiments is to demonstrate the value of intra-operational and inter-operational change analysis in selecting subsets of a web service's test cases. The first experimental evaluation concerns Intra-Operational Change Analysis with ORTWS. The second considers Inter-Operational Change Analysis with PRTWS and includes both (a) black box and (b) white box analysis.

The two experiments make use of the following web services: *Eucalyptus*, *SaaS*, *BookService*, *Amazon Web Service* (*AWS*), *Currency conversion*, *Global weather*, *Bible*, and *Sunset Sunrise*. The reason for choosing these web services is that they are easily understandable even in the absence of having practical experience using them. Like many evolving software systems [15], *Eucalyptus* is also available on GitHub. We used the two web services *SaaS* and *BookService* to verify and illustrate our technique. We developed both *SaaS* and *BookService* in Java using the glassfish server. The last five services, *AWS*, *Currency conversion*, *Global weather*, *Bible*, and *Sunset Sunrise* are services available on the Internet.

The experimental design involves two parts: *white box analysis* (where the internals of the WS code are known) and *black box analysis* (where they are unknown). The *white box analysis* investigates the reduction achieved using ORTWS on *SaaS* and *Eucalyptus,* and then PRTWS on *BookService*. The *black box analysis* investigates the reduction achieved using ORTWS on *AWS* and then using PRTWS on the four web services *Sunset Sunrise*, *Bible, Currency conversion,* and *Global weather*. Table 2 summarizes the eight case studies where 'Y' denotes 'Yes the case study is performed' otherwise, we give the reason for not performing the case study.

TABLE 2 WHITE & BLACK BOX ANALYSIS OF EIGHT STUDIES

Web service project	White Box Analysis	Black Box Analysis	Change source
Eucalyptus	Y	Y	GitHub
SaaS	Y	Y	Self-made
BookService	Y	Y	Self-made
AWS	Code is inaccessible	Y	Change analysis not performed
Sunset Sunrise		Y	
Bible		Y	
Currency conversion		Y	
Global weather		Y	

To illustrate AWSCM, the experiments use two differently designed test suites. Each test suite is a file (e.g., an xml or xmi file) that arranges test cases in a hierarchy (or nesting) of test steps. We designed the two kinds of test suites to provide two different types of associative code-test mappings. These two mappings contain test steps that each execute a specific operation or procedure.

5.1 Evaluation of Intra-Operational Change Analysis with ORTWS

The first experimental evaluation considers how *Intra-Operational Change Analysis* generates a WSDL slice. For each web service, the ORTWS part of AWSCM takes the following inputs: two consecutive versions of the WSDL and the WS code. It uses SoapUI and JMeter to generate a *reduced Regression Test Suites* from each *Subset* (*Difference*, *Unit*, and *Reduce*) *WSDL* separately. Table 3 describes the three case studies considered as well as the *Subset WSDL*s and the resulting test case reduction.

TABLE 3 CASE STUDIES FOR INTRA-OPERATIONAL CHANGE ANALYSIS BASED SUBSET WSDL AND ORTWS

Web service project	Subset WSDL	Case study detail	Subset WSDL contains	Test cases (TCs) reduction
SaaS	*D WSDL*	5 operations of the new WSDL and 3 operations of the old WSDL. Here, 2 operations are added. Where 2 of 4 TCs were extracted.	2 operations	50%
	R WSDL	1 of 5 operations was selected. 1 of 4 TCs was extracted.	1 operations	75%
	U WSDL	1 of 5 operations underwent change at the WS code level. 1 operation that underwent change at code-level. 1 of 4 TCs was extracted.	1 operations	75%
	C WSDL	3 of 4 (2+1+1) operation are unique in DWSDL, RWSDL and UWSDL. 3 of 4 TCs were extracted.	3 operations	25%
AWS	*R WSDL*	3 of 23 operations were selected. 3 of 23 TCs templates were extracted.	3 operations	86.95%
EucalyptusCC	*D WSDL*	26 operations of the new WSDL and 24 operations of the old WSDL. Here, 2 operations are added at WSDL-level. 2 of 26 TC templates were extracted.	2 operations	92.31%
	R WSDL	2 of 26 operations were selected. 2 of 26 TC templates were extracted.	2 operations	92.31%
	U WSDL	3 of 26 operations underwent changes at the WS code level. 3 of 26 TCs templates were extracted.	3 operations	88.5%
	C WSDL	6 of 7 (2+2+3) operations in the DWSDL, RWSDL and UWSDL were unique. 6 of 26 TCs templates were extracted.	6 operations	76.92%

To begin with, the first case study considers *SaaS*. AWSCM analyses *SaaS* to construct the DWSDL, RWSDL, UWSDL, and finally the CWSDL. Assume that *SaaS_X* and *SaaS_Y* are the two versions of *SaaS*, where *SaaS_X* provides three operations: *indexing*, *reading,* and *searching*. In this case, AWSCM found the two operations ('*edit*' and '*editFile*') were added to make *SaaS_Y*, AWSCM captures this in the DWSDL. It then selects the test cases for '*edit*' and '*editFile*' and executes the two parts of the *interoperable slice*. In this instance, AWSCM reduces the number of tests by 50%.

Now, suppose the user opts to form the RWSDL from the '*edit*' operation. AWSCM selects a test case for '*edit*' and uses it to execute the interoperable slice with respect to the '*edit*' operation. In this instance, the reduction in the number of test cases executed is 75%.

Next, we consider the UWSDL for *SaaS*. AWSCM found changes in the '*searching*' operation at the code level in the new version *SaaS_Y*, which it captures in the UWSDL. To reduce costs, AWSCM selects tests for the 'searching' operation to execute against the interoperable slice. The resulting reduced regression test suite correctly contains only the test cases for the search operation. Comparing the two versions of *SaaS*, in this instance, AWSCM reduces the number of test cases by 75%.

Thereafter, AWSCM takes and combines the unique operations of the DWSDL, RWSDL, and UWSDL to construct the CWSDL. This resulted in a reduction of 25% of the test cases using the *Combined Regression Test Suite*. The resulting Regression Test Suite contains subsets of the test cases that execute the interoperable slice of service *SaaS*.

The next two case studies as shown in Table 3 use *Eucalyptus* and *AWS*; we assume one test case per operations. We describe *Eucalyptus* and *AWS* together to exemplify an example of a *Reduce WSDL*. AWSCM prompts a user to select an operation of interest for the construction of the RWSDL. Suppose the input WSDL includes *N* operations then the user can select any number of the *N* operations, which are then used to construct the *RWSDL*. Additionally, AWSCM prompts the user to select the test cases and test steps to construct the reduced *Regression test suite*. For *AWS*, assume that the user selects 3 of 23 operations, this leads to 86.95% reduction in the number of test cases considered. For *EucalyptusCC*, assume that the user selects 2 of the 26 operations, this leads to 92.31% reduction in the number of test cases considered.

Finally, we illustrate AWSCM's use in constructing the DWSDL and UWSDL from two versions of the *Eucalyptus' code* as well as its WSDL. Using the DWSDL, AWSCM identifies only 7.69% of the operations as *changed* between the two versions of the WSDL for *Eucalyptus*. This leads to a reduction of 92.31% in the number of test cases that must be considered. While using the UWSDL, AWSCM identified 11.5% of the operations as changed between the two versions of the code of *Eucalyptus*. This leads to a reduction by 88.5% of test cases that must be considered.

Threat to validity: our focus is on the detection of differences related to *intra-operational changes*, thus some *inter-operational changes* might be missed. We consider the impact of inter-operational changes in the second experimental evaluation.

5.2 Evaluation of Inter-Operational Change Analysis with PRTWS

A) Black box inter-operational analysis with PRTWS

Turning to PRTWS, we first consider *black box analysis* using the services *Currency conversion*, *Global weather*, *Bible*, and *Sunset Sunrise* as found in Table 4.

TABLE 4 BLACK BOX INTER-OPERATIONAL CHANGE ANALYSIS BASED PRTWS CASE STUDIES

Web service project	Primary parameter	Non Primary parameter	Case study detail	Test Case (TC) & Test Step (TS) reduction
Currency Conversion	FromCurrency	ToCurrency	3 out of 9 TSs are selected in 2 out of 4 TCs	66 % reduction in TSs and 50% reduction in TCs
Global weather	CountryName	CityName	1 out of 2 TSs are selected in 1 out of 4 TCs.	50% reduction in TSs and 75% reduction in TCs
Bible	Book Title	Chapter Name	2 out of 4 TSs are selected in 1 out of 4 TCs	50% reduction in TSs and 75% reduction in TCs
Sunset Sunrise	Latitude & Longitude	SunSetTime, SunRiseTime, TimeZone, Day, Month, & Year	4 out of 8 TSs are selected from 1 (single) TC	50% reduction in TSs

To begin with, *Currency conversion* includes the operation *CurrencyConvertor* in which *FromCurrency* is the *primary parameter* making *ToCurrency* the (only) *non-primary parameter*. Here, AWSCM extracts test cases for a particular fixed currency, for example the *Indian Rupee,* as the *primary parameter*. Each test case contains some other currency, such as the *Japanese Yen* or *Australian Dollar* as the *non-primary parameter*. For the second case study, *Global weather,* we construct a *Regression Test Suite* as shown in Figure 9(a). Here, *CountryName* is the *primary parameter* and *CityName* is the (only) *non-primary parameter*. Then for the *Bible* web service, we chose the parameter *BookTitle* as the *primary parameter*, thus making *ChapterName* and *Verse* as the *non-primary parameters*. Finally, for the *Sunset Sunrise* web service, we chose *Latitude* and *Longitude* as the *primary parameter*, as a special case where two parameters are used. These two determine the two *non-primary parameters*: *SunSetTime* and *SunRiseTime*.

We conducted the four case studies based on user selected test steps that form each test case. As shown in Table 4 for *currency conversion*, AWSCM selected 3 of 9 test steps (TSs) from the 2 of 4 test cases (TCs). From each web services in Table 4, we retrieve the subset of the test cases that execute the subset of the service. AWSCM combines the resulting selected test cases to form the reduced test suites.

Threat to validity: this experimental evaluation focuses on test case optimization by creating the various subsets where both Intra-Operational Change Analysis with ORTWS and Inter-Operational Change Analysis with PRTWS individually perform well. However, the combined effect of the two needs to be considered.

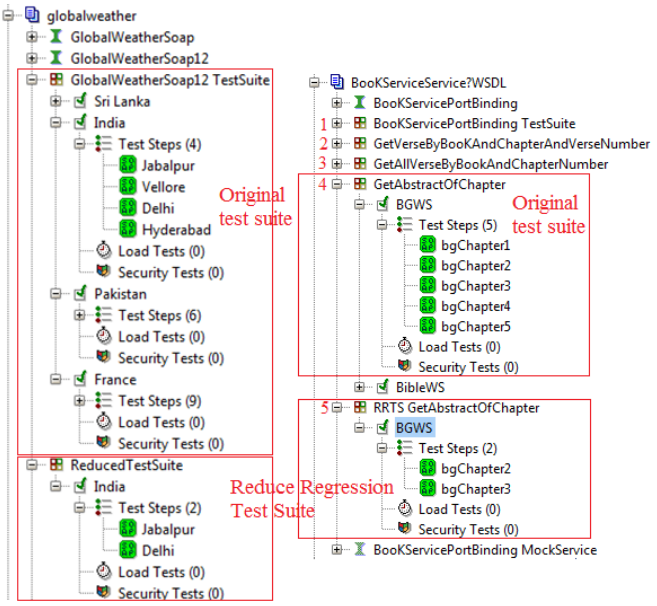

Fig. 9. Snapshots of SoapUI Test suite and Regression test suite for PRTWS of (a) *Global weather* web service and (b) *BookService*.

B) White box inter-operational analysis with PRTWS

In the *white box analysis*, AWSCM exploits the combination of Intra-Operational Change Analysis and Inter-Operational Change Analysis to reduce regression-testing cost. In the combination, Intra-Operational Change Analysis discovers the specific operations that have undergone change while Inter-Operational Change Analysis helps in the analysis of changes in complex calls (inter-operational dependencies) between web service operations. We conducted the white box analysis of *SaaS* and *BookService* by finding the subset of the test steps that make up the test cases.

Next, we consider the white box analysis of *BookService*, which exemplifies a module of a library management system. We designed the service to get information from two books the *Bhagavad Gita* (BG) and the *Bible*. The *BookService* uses two other web service *BGWS* and *BibleWS*. *BGWS* provides the three operations *bgOpAbst*, *bgAllVerse,* and *bgOp*. Similarly, *BibleWS* provides the three operations *bibleOpAbst*, *bibleAllVerse*, and *bibleOp*. *BookService* provides the four operations described in Table 5 and detailed in Figure 10.

The first operation, *findBookNumber*, takes no input and returns a name with the reference number for all the books in the library. To keep things simple, we retain only two books *Bhagavad Gita* (BG) with reference number 1 and the *Bible* with reference number 2.

The second operation, *getVerseByBookAndChapterAndVerseNumber* (number 43 in Figure 10) takes three inputs, *bookNumber*, *chapterNumber,* and *verseNumber*. Depending on *bookNumber* (1 or 2), this operation calls the operations *bgOp* (number 1) or *bibleOp* (number 22) as illustrated in Figure 11. Then depending on *chapterNumber* a call to either *bgChapterN* in class *BGVerse,* or *bibileChapterN* in class *BibileVerse* is made; here N is the chapter number. The called function returns the verse for the given *verseNumber*.

TABLE 5 PARAMETER OF OPERATIONS IN *BOOKSERVICE*

Operations	ID	Primary parameter	Non–Primary Parameter
findBookNumber	--	---	---
getVerseByBookAnd ChapterAndVerseNumber	43	bookNumber	verseNumber & chapterNumber
getAllVerseByBookAnd ChapterNumber	44	bookNumber	chapterNumber
getAbstractOfChapter	45	bookNumber	chapterNumber

BookService.BGWS.bgOp	**1**
BookService.BGVerse.bgChapter1	2
BookService.BGVerse.bgChapter2	3
BookService.BGVerse.bgChapter3	4
BookService.BGVerse.bgChapter4	5
BookService.BGVerse.bgChapter5	6
BookService.BGVerse.bgChapter6	7
BookService.BGVerse.bgChapter7	8
BookService.BGVerse.bgChapter8	9
BookService.BGVerse.bgChapter9	10
BookService.BGVerse.bgChapter10	11
BookService.BGVerse.bgChapter11	12
BookService.BGVerse.bgChapter12	13
BookService.BGVerse.bgChapter13	14
BookService.BGVerse.bgChapter14	15
BookService.BGVerse.bgChapter15	16
BookService.BGVerse.bgChapter16	17
BookService.BGVerse.bgChapter17	18
BookService.BGVerse.bgChapter18	19
BookService.BGWS.bgOpAbst	**20**
BookService.BGWS.bgAllVerse	**21**
BookService.BibleWS.bibleOp	**22**
BookService.BibleVerse.bibleChapter1	23
BookService.BibleVerse.bibleChapter2	24
BookService.BibleVerse.bibleChapter3	25
BookService.BibleVerse.bibleChapter4	26
BookService.BibleVerse.bibleChapter5	27
BookService.BibleVerse.bibleChapter6	28
BookService.BibleVerse.bibleChapter7	29
BookService.BibleVerse.bibleChapter8	30
BookService.BibleVerse.bibleChapter9	31
BookService.BibleVerse.bibleChapter10	32
BookService.BibleVerse.bibleChapter11	33
BookService.BibleVerse.bibleChapter12	34
BookService.BibleVerse.bibleChapter13	35
BookService.BibleVerse.bibleChapter14	36
BookService.BibleVerse.bibleChapter15	37
BookService.BibleVerse.bibleChapter16	38
BookService.BibleVerse.bibleChapter17	39
BookService.BibleVerse.bibleChapter18	40
BookService.BibleWS.bibleOpAbst	**41**
BookService.BibleWS.bibleAllVerse	**42**
BookService.BookService.getVerseByBookAndChapterAndVerseNumber	**43**
BookService.BookService.getAllVerseByBookAndChapterNumber	**44**
BookService.BookService.getAbstractOfChapter	**45**

Fig. 10 Operations (highlighted in bold) and procedures (in classes of *BGVerse* or *BibleVerse*) of *BookService* with its corresponding ID numbers.

The third operation, *getAllVerseByBookAndChapterNumber* (number 44 in Figure 10), takes two inputs, *bookNumber,* and *chapterNumber*. Depending on the value of *bookNumber* (1 or 2), this operation calls the operations *bgAllVerse* (number 21) or *bibileAllVerse* (number 42) as shown in Figure 11. The result of the call is all the verses from the chapter, which are obtained by sequentially calling either *bgChapterN* in class *BGVerse* or *bibileChapterN* in class *BibileVerse*; here N ranges from the first to the last chapter.

The fourth operation, *getAbstractOfChapter (*number 45 in Figure 10), takes two inputs, *bookNumber* and *chapterNumber*. Depending on the value for *bookNumber* (1 or 2), this operation calls either *bgOpAbst* (number 20) or *bibleOpAbst* (number 41) as shown in Figure 11. Its output is the abstract for the selected chapter.

To evaluate Intra-Operational and Inter-Operational Change Analysis, we introduced an intentional change in two procedures: *bgChapter2 (*number 3 in Figure 10) and *bgChapter3 (*number 4 in Figure 10). The first subgraph of Figure 12 shows that these two changes affect the two operations *bgOp (*number 1 in Figure 10 and Figure 11) and *bgAllVerse (*number 21 in Figure 10 and Figure 11). Thus, avoiding analysis of other procedures in *BGWS*, which significantly reduces the regression testing effort for *BookService*.

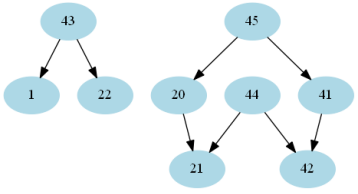

Fig. 11. The calls of operations '43', '44', and '45' in *BookService*. In the first subgraph, '1' is *bgOp* and '22' is *bibleOp*. In the second subgraph, '21' is *bgAllVerse*, '42' is *bibleAllVerse*, '20' is *bgOpAbst* and '41' is *bibleOpAbst*. All the operations are affected due to inter-operational dependencies changes and highlighted in blue.

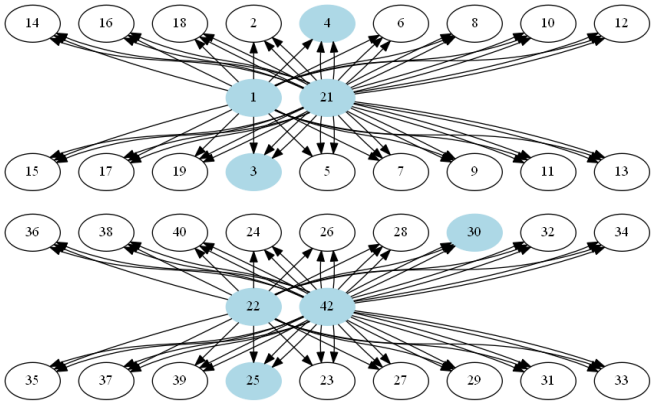

Fig. 12. The top figure shows the eighteen calls of operation *bgOp* ('1') and the thirty-six calls of operation *bgAllVerse* ('21') in BGWS. While the bottom figure shows the eighteen calls of operation *bibleOp* ('22') and thirty-six calls of operation *bibleAllVerse* ('42') in BibleWS. Affects due to the changes are highlighted in blue.

The first goal of AWSCM's Intra-Operational Change Analysis is to identify these two affected operations due to changes in the two procedures. Using these two operations AWSCM constructs the *Subset WSDLs*: *DWSDL*, *UWSDL, RWSDL,* and their combination CWSDL. The second goal of AWSCM's Inter-Operational Change Analysis is to construct the PWSDL as shown in Table 6. AWSCM uses the two operations to extract the regression test suite that can execute the interoperable slice, which is the first subgraph shown in Figure 12.

TABLE 6 WHITE BOX INTER-OPERATIONAL CHANGE ANALYSIS BASED PRTWS CASE STUDY FOR *BOOKSERVICE*

Book Service detail	Case study detail	Inter-operational Dependence analysis	PWSDL contains	Test case reduction
Original WSDL has 4 operations. Figure 11 and 12 shows 116 calls.	'*bgChapter2*' & '*bgChapter3*' had gone through changes at code level	3 out of 4 operations in *BGWS* are dependent on the two changes	3 operations	2 out of 5 test steps are extracted 60% reduction
	'*bibleChapter3*' & '*bibleChapter8*' had gone through the changes	3 out of 4 operations in *BibleWS* is dependent on the two changes	3 operations	2 out of 5 test cases are extracted 60% reduction

For the BookService case study, we used the five SoapUI test suites shown in Figure 9(b). In the figure, the first test suite (1) is for ORTWS based testing. It contains test cases for all the operations of *BookService*. We designed the next three test suites (2, 3, and 4) for PRTWS such that each test cases executes a single operation. In the figure 9(b), the second test suite tests the operation *'getVerseByBookAndChapterAndVerNumber'*, the third tests the operation *'getAllVerseByBookAndChapterNumber'*, and the fourth tests the operation *'getAbstractOfChapter'*. The fifth shows the resulting *Regression Test Suite* constructed by AWSCM. Here, AWSCM correctly extracted the two test cases ('*bgChapter2*' and '*bgChapter3*') and the fourth test suite of the operation '*getAbstractOfChapter*'. The execution of the two test cases, which exercises the affected operations '*bgChapter2*' and '*bgChapter3*'.

Similarly, we made two additional intentional changes in '*bibleChapter3'* (number 25) and '*bibleChapter8'* (number 30) as shown in Figure 10 and highlighted in the second subgraph of Figure 12. For this experiment, we followed the same pattern as the first example for '*bgChapter2*' and '*bgChapter3*'. Further, the last row of the Table 6 provides details regarding the analysis of *BibleWS*, which is self-explanatory. Avoiding the analysis of the unchanged procedures in *BibleWS* reduces the effort needed when regression testing *BookService*.

Threat to validity: we conducted this case study for our own *BookService*. However, application of the proposed techniques should be transferable to industrial projects.

5.3 Observations and findings

Finally, we discuss our experience with AWSCM while conducting the experiments.

1. *We can choose to perform change analysis on the WSDL and the WS code separately and may skip either analysis if it is not required.* Specially, skipping the WS-code analysis is necessary when we do not have access to the WS code. In this way, we can do change analysis for both the web service owner and the web server client.

2. *The Intra-Operational and Inter-Operational Change Analysis of web services identifies subsets of the test cases that lead to reduced regression-testing cost.* It is obvious that automated change analysis information is useful in regression test-case selection. Hence, we can test the changes to a service, which helps in service maintenance and evolution.

3. *We observe capturing changes are useful to make Subset WSDLs, which are helpful to identifying relevant test cases*. Our approach takes advantage of the loose coupling of operations as well as web service composition to create a *Subset WSDL*. This provides a way to perform subset service testing by executing subsets of the test cases, which takes less time and provides clarity while testing.

4. *The execution and analysis of an interoperable slice using an interface slice can reduce the number of test cases required for regression testing.* The Subset WSDLs (as WSDL slices) are helpful in building the reduced *Regression Test Suite*, which is used to execute the interoperable slice.

5. *Systematic design of the test suites for a web service according to the associative code-test mapping makes it easier to conduct regression testing.* This paper demonstrates two systematic ways to design test suites by exploiting an associative code-test mapping for ORTWS and PRTWS. The resulting test suites make test-case structure more formal, systematic, and analytic by associating test steps with the flow of the code.

6. *AWSCM proved helpful in testing the changes that occurred in a service.* We conducted the experiments using two standard web-service testing tools, SoapUI and JMeter. We executed the selected subset of test cases to test the changes. Here, change analysis of web service reduces the number of test cases.

7. *AWSCM successfully generated accurate output (Subset WSDLs* and *Reduced Regression Test Suites).* It did so in only few seconds (all of the algorithms in AWSCM are linear). Furthermore, software maintenance is a challenging task for a project manager. The case studies illustrate how change analysis of web services can help a manager improve regression testing.

6 RELATED WORK AND DISCUSSION

This section first considers work related to web service slicing, including that of Mao [16], who proposed an approach for static slicing BPEL programs in web service compositions by using an extended control flow graph (ECFG). In contrast, we consider a dynamic slicing approach for web service interfaces (i.e., the WSDL) that slices-out a given web service, such that a web service slice remains inter-operable. Zhang et al. [17] proposed a composition method based on program slicing. Earlier, as with our work, Fu et al. [18] performed slicing of WSDL instead of slicing the web service code as in the work of Mao and Zhang et al. [16][17].

Work related to *regression testing* includes that of Binkley [19][20], who describes a slicing-based approach to reducing regression-testing cost by identified changed components (e.g., statements, modules, classes, or procedures). We recast this approach in the context of web services and implement the result as part of AWSCM. With similar goals, Rothermel and Harrold [21] used control flow graphs to select tests that execute changed code.

Similarly, AWSCM internally uses the web-service call graph to gather changes. Anjaneyulu et al. [22] proposed a tool named InARTS for regression test-case selection based on sub-graph changes captured using syntactically and semantically altered procedure sets. InARTS used these sets to select tests from the old test suite.

Work related to *change analysis of web service* includes that of Andrikopoulos et al. [23], which describes evolution of services by discriminating between shallow and deep changes. Their shallow and deep changes are analogous to our intra-operational and inter-operational changes. Romano and Martin [24] presented a fine-grained change detection technique to analyse various evolving WSDLs. Most recently, Zhang et al. [25] presented FUNNEL, a tool that performs impact assessment of software changes in large web-based services.

Work related to *web service testing* includes that of Tsai et al. [26], which describes four ways of extending the WSDL to facilitate web service testing. In addition, Bai et al. [27] describe an automatic WSDL based test-case generation technique for web service testing. Most existing web service testing tools trace their roots back to these two papers. Sneed [28] presented a semi-automated approach to web-service test maintenance. He used WSDL tags and attributes to automatically generate a test script into which a tester can manually enter test data. Likewise, we perform WSDL based test-case template generation for newly inserted operations.

Work related to the *regression testing of web services* includes that of Ruth et al. [29][30], which describes a mechanism to transform web service code into a local program to which they apply a regression test-suite selection approach. Tarhini et al. [31] presented an approach that provides a safe regression test suite for web services modelled as a Timed Labeled Transition System (TLTS). However, in these works we did not find detailed information regarding tool implementations.

Massimiliano et al. [32] described a tool-supported approach to run and re-run test suites repeatedly for functional and non-functional expectations. Tamim and Reiko [33] describe model-based regression testing of web services. In contrast to the work of Di Penta and Tamim, we used standard web service testing tools such as SoapUI [55] and JMeter [56].

Chaturvedi [49] and Masood et al. [34] presented a similar approach for WSDL specification based regression testing of web-services, which uses comparison between two versions of WSDL to identify changes. Chaturvedi [11] and Sahoo et al. [35] proposed similar approach to optimize (or reduce) web-service analysis (especially in regression testing) based on slicing of web service for different evolving service conditions using change detection. Extensively, we apply a change analysis approach to web service code as well as to the WSDL. Further, we also created WSDL slicing as subset WSDL specification.

Next, we discuss four possible applications of our work. The first application motivates importance of *subset service* access. Here, a web service slice can help to perform *liveness analysis* [36] and *dead-code elimination* [37].

The second application involves *Intra-Operational and Inter-Operational Change Analysis based regression testing,* which can be helpful in maintaining *Quality of Service* (QoS) and *Service Level Agreements* (SLAs), which describe the scope, quality, and responsibilities of a service provider regarding the service consumers. To maintain QoS, service providers must monitor their web service continuously by performing change analysis based regression testing [38]. In this paper, we present an approach to do change analysis that enables *regression testing of web services* to manage the QoS, which, in turn, helps to enforce SLA requirements. Reduction in the effort of regression testing can lead to reduction in the effort of monitoring and maintaining QoS. Thus, it lowers the effort required to guarantee the SLA. In an earlier work, Chattopadhyay and Banerjee [39] demonstrated construction of composite services by keeping satisfactory QoS in large-scale web services.

The third application of our work is to improve SOAP performance. Earlier, Nayef et al. [40] presented differential deserialization (DDS) that exploits the similarities between incoming messages to reduce deserialization time. Tekli et al. [41] presented Differential SOAP Multicasting (DSM), which identifies common patterns and differences between SOAP messages as trees using Tree Edit Distance. Thereafter, Tekli et al. [42] presented a survey of similarity-based SOAP performance enhancement, which was used to improve web service performance based on techniques that identify common parts of SOAP messages. Other work on SOAP performance improvement includes work on the optimization of web service security [43] and SOAP compression [44]. Looking forward, we hope that an approach based on *subset services*, *web service slicing*, *inter-operational analysis*, and *WSDL slices* will further improve SOAP performance.

The fourth application of our work involves network slicing [45], which attempts to divide the network virtually to create sharing. For example, slicing home networks [46], network slicing as service [47], and network slicing in 5G mobile systems [48]. Anyone may incorporate web service slicing as part of network slicing, thus exploiting operation subsets that are interoperable over network.

The differences between our previous work and this paper are as follows. ORTWS (as described in this paper) is a significant extension of the technique presented in our earlier work [14]. Furthermore, PRTWS is a new novel contribution. Preliminary aspects of these ideas were presented as part of the lead author's master thesis [50]. In comparison, this paper defines and describes our contributions through the introduction of two techniques: *intra-operational analysis* and *inter-operational analysis*. Moreover, for *large-scale web services* this paper introduced *subset services, web service slicing, interoperable slices,* and *interface slices* (*WSDL slices*). These are new concepts except for the earlier presented concept of a Subset WSDL [11]. We present an experimental study of several web services to identify changes and to evaluate the effectiveness of *AWSCM* [13]. The empirical experiments also represent significant extensions to the case studies found in our previous work [11][13][14][49][50].

In summary, based on web service slicing, this work is an attempt to integrate the two well-known concepts of ubiquitous systems [1] and program slicing [2]. This paper also improves the state of the art for web-service regression testing. We consider *intra* and *inter operational changes* to a web service in terms of WSDL, WS code, inter-operational dependencies, and parameters. We used the identified intra-operational and inter-operational changes to extract various parts of the WSDL, and then combine these parts to construct the *Subset WSDL*. The change analysis information also helps uncover test data for regression testing. Finally, we uplifted the importance of integrating inter-operational change analysis based regression testing of web services.

7 SUMMARY

This paper introduces *web-service slicing*, which exploits the combination of an *interoperable slice* and an *interface slice*. An *interface slice* gives access to the subset of the operations found in a *web service slice*. To the best of our knowledge, we are the first to propose *web service slicing* for maintaining interoperability of subset services over a network. We consider the following subsets of the WSDL: the *Difference WSDL*, the *Unit WSDL*, the *Reduce WSDL*, the *Combined WSDL,* and the *Parameter WSDL*. We used these *Subset WSDLs* (or *WSDL slices*) to support regression testing of an evolving web service.

The paper also presents the usefulness and requirements for a *subset service* (e.g., a *web service slice*). This study involves *intra-operational* and *inter-operational analysis*, which gives rise to two the new *regression-testing techniques for web services*, ORTWS and PRTWS. Both techniques select subsets of existing test cases to construct reduced *Regression Test Suites*. To support regression testing, we aimed at the properties of intra-operational and inter-operational changes that enable the execution of a *subset service* using a subset of the test cases that impacts the affected operations. The result reduces testing effort during the maintenance and evolution of SOA for cloud-based systems.

In addition, this paper describes our tool AWSCM, which implements the proposed techniques. We performed eight successful case studies using eight web service projects. For each case study, we achieved a reduction in regression-testing effort. The details for the AWSCM and additional case studies are available in the first author's thesis [50] and at website link
https://sites.google.com/site/animeshchaturvedi07/research/awscm

In the future, we hope these techniques will become integral components of standard web service development and testing tools such as SoapUI, JMeter, NetBeans, and Eclipse. Finally, we plan to apply interface slicing, intra-operational and inter-operational analysis to other APIs.

ACKNOWLEDGMENT

The authors wish to thank Dr. Atul Gupta, Dr. Anjaneyulu Pasala, Dr. Srinivas Padmanabhuni, and Prof. Rushikesh K. Joshi for their advice and comments on the work.

REFERENCES

[1] Weiser Mark. "The computer for the 21st century." *Mobile Computing and Communications Review* 3.3 (1999): 3-11.

[2] Weiser Mark. "Program slicing." *Proceedings of the 5th International Conference on Software Engineering*. IEEE Press, 1981.

[3] Binkley David W., and Keith Brian Gallagher. "Program slicing." *Advances in Computers* 43 (1996): 1-50.

[4] Binkley David. "The application of program slicing to regression testing." *Information and Software Technology* 40.11 (1998): 583-594.

[5] Beck Jon, and David Eichmann. "Program and interface slicing for reverse engineering." *Proceedings of the 15th International Conference on Software Engineering*. IEEE Computer Society Press, 1993.

[6] Northrop Linda, et al. "Ultra-large-scale systems: The software challenge of the future." *Carnegi-Mellon Univ. Pittsburgh PA Software Engineering Institute*, 2006.

[7] Agrawal Hiralal, et al. "Incremental regression testing." *Software Maintenance, 1993. CSM-93, Proceedings, Conference on*. IEEE, 1993.

[8] Elbaum Sebastian, Gregg Rothermel, and John Penix. "Techniques for improving regression testing in continuous integration development environments." *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2014.

[9] Horwitz Susan, Thomas Reps, and David Binkley. "Interprocedural slicing using dependence graphs." *ACM Transactions on Programming Languages and Systems (TOPLAS)* 12.1 (1990): 26-60.

[10] Hall Mary W., and Ken Kennedy. "Efficient call graph analysis." *ACM Letters on Programming Languages and Systems (LOPLAS)* 1.3 (1992): 227-242.

[11] Chaturvedi Animesh. "Subset WSDL to Access Subset Service for Analysis." *Cloud Computing Technology and Science (CloudCom), 2014 IEEE 6th International Conference on*. IEEE, 2014.

[12] Edgar F. Codd, "A relational model of data for large shared data banks", *Pioneers and Their Contributions to Software Engineering*. Springer Berlin Heidelberg, pp. 61-98, 2001.

[13] Chaturvedi Animesh. "Automated Web Service Change Management AWSCM-A Tool." *Cloud Computing Technology and Science (CloudCom), 2014 IEEE 6th International Conference on*. IEEE, 2014

[14] Chaturvedi Animesh, and Atul Gupta. "A tool supported approach to perform efficient regression testing of web services." *Maintenance and Evolution of Service-Oriented and Cloud-Based Systems (MESOCA), 2013 IEEE 7th International Symposium on the*. IEEE, 2013.

[15] Tom Mens, Alexander Serebrenik, and Anthony Cleve. "Evolving Software Systems." *Springer Publishing Company, Incorporated*. 2014.

[16] Mao Chengying. "Slicing web service-based software." *Service-Oriented Computing and Applications (SOCA), 2009 IEEE International Conference on*. IEEE, 2009.

[17] Zhang Yingzhou, et al. "Web service publishing and composition based on monadic methods and program slicing." *Knowledge-Based Systems* 37 (2013): 296-304.

[18] Fu Wei, et al. "Program slicing based web-service generation and composition." *Service Oriented System Engineering (SOSE), 2010 Fifth IEEE International Symposium on*. IEEE, 2010.

[19] Binkley David. "Reducing the cost of regression testing by semantics guided test case selection." *Software Maintenance, 1995. Proceedings, International Conference on*. IEEE, 1995.

[20] Binkley David. "Semantics guided regression test cost reduction." *IEEE Transactions on Software Engineering* 23.8 (1997): 498-516.

[21] Gregg Rothermel and Mary Jean Harrold, "A Safe, Efficient Regression Test Selection Technique," *ACM Transactions on Software Engineering and Methodology* vol. 6, no. 2, pp. 173-210, April 1997.

[22] Anjaneyulu Pasala et al., "Selection of Regression Test Suite to Validate

Software Applications upon Deployment of Upgrades", *19th Australian Conference on Software Engineering* on IEEE, *2008*.

[23] Andrikopoulos Vasilios, Salima Benbernou, and Michael P. Papazoglou. "On the evolution of services." *IEEE Transactions on Software Engineering* 38.3 (2012): 609-628.

[24] Romano Daniele, and Martin Pinzger. "Analyzing the evolution of web services using fine-grained changes." *Web Services (ICWS), 2012 IEEE 19th International Conference on.* IEEE, 2012.

[25] Zhang Shenglin, et al. "FUNNEL: Assessing Software Changes in Web-based Services." *IEEE Transactions on Services Computing* (2016).

[26] Tsai Wei-Tek, et al. "Extending WSDL to facilitate web services testing". *High Assurance Systems Engineering, 2002. Proceedings. 7th IEEE International Symposium on.* IEEE, 2002.

[27] Bai Xiaoying, et al. "WSDL-based automatic test case generation for web services testing". *IEEE International Workshop on Service-Oriented System Engineering (SOSE'05).* IEEE, 2005.

[28] Sneed Harry M. "Web Service Test Evolution". *International Conference on Software Quality*. Springer International Publishing, 2016.

[29] Ruth Michael, Feng Lin and Shengru Tu, "Applying Safe Regression Test Selection Techniques to Java web services", *International Journal of Web Services Practices*. vol.2, No.1-2, 2006, pp. 1-10.

[30] Ruth Michael, and Shengru Tu. "A safe regression test selection technique for web services". *Internet and Web Applications and Services, 2007. ICIW'07. Second International Conference on*. IEEE, 2007.

[31] Tarhini Abbas, Hacène Fouchal, and Nashat Mansour. "Regression Testing Web Services-based Applications." *AICCSA 6* (2006): 163-170.

[32] Massimiliano Di Penta et al. "Web services regression testing." *Test and Analysis of web Services*. Springer, Berlin, Heidelberg, 2007. 205-234.

[33] Tamim Ahmed Khan and Reiko Heckel, "A Methodology for Model-Based Regression Testing of web service", *IEEE Testing: Academic and Industrial Conference - Practice and Research Techniques*. IEEE Computer Society. pp. 123-124, 2009.

[34] Masood Tehreem, Aamer Nadeem, and Shaukat Ali. "An automated approach to regression testing of web services based on WSDL operation changes." *Emerging Technologies (ICET), 2013 IEEE 9th International Conference on*. IEEE, 2013.

[35] Sahoo Sobhana, and Abhishek Ray. "A framework for optimization of regression testing of web services using slicing." *Advances in Computing, Communications and Informatics (ICACCI), 2017 International Conference on*. IEEE, 2017.

[36] Araujo Miguel, and Ahmed Bougacha. "Interprocedural Variable Liveness Analysis." (2014).

[37] Bodik Rastislav, and Rajiv Gupta. "Partial dead code elimination using slicing transformations." *ACM SIGPLAN Notices*. Vol. 32. No. 5. ACM, 1997.

[38] Canfora G., Di Penta M., "Testing services and service-centric systems: challenges and opportunities" *IT Professional* vol. 8, may 2006.

[39] Chattopadhyay Soumi, and Ansuman Banerjee. "QoS constrained Large Scale Web Service Composition using Abstraction Refinement." *IEEE Transactions on Services Computing* (2017).

[40] Abu-Ghazaleh Nayef, and Michael J. Lewis. "Differential deserialization for optimized soap performance." *Supercomputing, 2005. Proceedings of the ACM/IEEE SC 2005 Conference*. IEEE, 2005.

[41] Tekli Joe, Ernesto Damiani, and Richard Chbeir. "Differential SOAP multicasting." *Web Services (ICWS), 2011 IEEE International Conference on*. IEEE, 2011.

[42] Tekli Joe M., et al. "SOAP processing performance and enhancement". *IEEE Transactions on Services Computing* 5.3 (2012): 387-403.

[43] Teraguchi Masayoshi, et al. "Optimized web services security performance with differential parsing". *Service-Oriented Computing–ICSOC 2006* (2006): 277-288.

[44] C. Werner, C. Buschmann, and S. Fischer, "WSDL-Driven SOAP Compression," *Int'l J. Web Services Research*, vol. 2, no. 1, pp. 18-35, 2005.

[45] Vassilaras Spyridon, et al. "The algorithmic aspects of network slicing." *IEEE Communications Magazine* 55.8 (2017): 112-119.

[46] Yiakoumis Yiannis, et al. "Slicing home networks." *Proceedings of the 2nd ACM SIGCOMM workshop on Home networks*. ACM, 2011.

[47] Zhou Xuan, et al. "Network slicing as a service: enabling enterprises' own software-defined cellular networks." *IEEE Communications Magazine* 54.7 (2016): 146-153.

[48] Foukas Xenofon, et al. "Network Slicing in 5G: Survey and Challenges." *IEEE Communications Magazine* 55.5 (2017): 94-100.

[49] Chaturvedi Animesh. "Reducing cost in regression testing of web service." *Software Engineering (CONSEG), 2012 CSI Sixth International Conference on*. IEEE, 2012.

[50] Chaturvedi Animesh. "Change Impact Analysis Based Regression Testing of Web Services." *arXiv preprint arXiv:1408.1600* (2014).

[51] Apiwattanapong T, Orso A, and Harrold M J, "JDiff: A Differencing Technique and Tool for Object-Oriented Programs," *Journal of Automated Soft. Engineering*, vol. 14, no. 1, pp 3-36, March 2007.

[52] "Membrane SOA Model," http://membrane-soa.org/soa-model/, March 12, 2017.

[53] "Java Regular Expressions" https://docs.oracle.com/javase/tutorial/essential/regex/ March 12, 2017.

[54] Georgios Gousios, "Programs for producing static and dynamic (runtime) call graphs for Java programs" https://github.com/gousiosg/java-callgraph, March 12, 2017.

[55] "SoapUI," http://www.soapui.org/, March 12, 2017.

[56] "JMeter," http://jmeter.apache.org/, March 12, 2017.

Animesh Chaturvedi is pursuing his PhD degree from Indian Institute of Technology Indore. He received his Master of Technology (M.Tech) degree from Indian Institute of Information Technology, Design and Manufacturing Jabalpur. He received his Bachelors of Engineering degree from IET - Devi Ahilya University. To do research, he declined non-research based job offers of few MNC's at early 20's of his life. He did research work in Motorola Mobility, Arris, and IIT-Kanpur. His research interest is in SOA, Cloud computing, Big Data Analytics, Data Mining, Machine Learning, Evolving systems, Software Maintenance and Evolution.

Dave Binkley is a Professor of Computer Science at Loyola University Maryland where he has worked since earning his doctorate from the University of Wisconsin in 1991. He has been a visiting faculty researcher at the National Institute of Standards and Technology (NIST), worked with Grammatech Inc. on CodeSurfer development, and was a member of the Crest Centre at Kings' College London. Dr. Binkley's current NSF funded research focuses on software product families and the application of information retrieval techniques in software engineering. He recently co-chaired the program for 2014 Software Evolution Week combining WCRE and CSMR.