

Applying Safe Regression Test Selection Techniques to Java Web Services

Michael Ruth, Feng Lin and Shengru Tu

Computer Science Department, University of New Orleans
2000 Lakeshore Drive, New Orleans, Louisiana, USA
[mruth, flin, shengru]@cs.uno.edu

Abstract—As web services grow in maturity and use, so do the methods being used to test and maintain them. Although regression testing is a major component of most major testing systems, it has only begun to be applied to Web services. The majority of these tools and techniques focus on test case generation, ignoring the potential benefits of regression test selection. Regression test selection (RTS) is a mechanism for reducing the number of tests that must be rerun to ensure some level of confidence. Safe RTS techniques add an additional constraint, namely no modification revealing tests are left unselected. Since Safe RTS techniques involve white-box testing, they cannot be directly applied to Web services. We will propose a mechanism for applying a safe RTS technique to Java-based Web services in an end-to-end manner using a code transformation-based approach. Additionally, we will provide implementation details as well as an example.

Keywords: Web Services, Regression Test Selection, Safe Regression Test Selection, End-to-End RTS.

I. INTRODUCTION

WEB services have become the de-facto standard for modeling inter- and intra-enterprise business processes. Since the business processes often change very rapidly, the Web services must also undergo frequent and rapid changes. These modifications have to be supported by frequent verification to ensure some level of service quality. Testing the actual code directly is normally performed using mature test techniques and test case generation tools [1]. Often the integration and system testing that must be performed to ensure the most recent modification did not cause any unexpected errors somewhere else in the system are much more costly tasks. The bottom line is that the modified systems still function at least as well as they did prior to modification. This is a very important step to establish a level of confidence in the Web services that model the ever-changing business processes in a real-world enterprise.

The typical process of verifying that the modifications in the system do not make the system worse is to run the test cases previously used to test the original system. This “retesting” is called Regression Test (RT). One of the key ideas of RT is reducing the size of what must be retested, or regression test selection (RTS). A safe regression test selection technique has an additional quality; it guarantees that

nothing that could produce an error in the original set of tests will be left untested in the reduced set of tests under certain well-defined conditions. This is particularly important in a Web service environment in which innumerable services interact with one another because retesting all previously run test cases would be extremely costly.

To date, several safe RTS techniques have been developed and proven to reduce the size of a regression test suite effectively. One of these is specifically for Java software and was developed by Harrold [2]. This technique uses a combination of static and dynamic (using a profiler) analysis to produce a model called a Java Interclass Graph or JIG. Based on this model, one can determine the minimum set of test cases that need to be rerun in order to ensure the same level of confidence as not removing any tests. Since this RTS technique requires dynamic analysis, which in turn requires the system to be run in a single Java Virtual Machine (JVM), this mechanism cannot be applied to the inherently distributed Web Service environment.

In this article, we will report a safe RTS approach for Java-based Web services in an end-to-end manner. This approach is based on Harrold’s method [2]. The dynamic analysis for Java Web services is carried out using a simulation tool that transforms the Java Web services code into local Java programs which can be connected together to form a global JIG. This global JIG is passed to a selection mechanism, which will safely determine which tests need to be rerun. The tests that need to be rerun will be rerun on the actual deployed services.

The key contribution of this article is that this is the first approach to safe regression test selection for end-to-end Web-service applications in Java. First, our approach reduces regression testing costs compared to the retest-all approach. Second, the transformation takes into account changes and effects in both the back-end components and the choices of service deployment modes such as session versus stateless. Third, the selection of test cases preserves the safety provided by the safe RTS techniques themselves. In other words, it is guaranteed that every test case that is affected by the modification will be selected to retest. This safety is maintained by strict adherence to a set of guidelines provided by the authors of RTS technique we have selected.

In our experiments, the simulation was developed using

Apache Axis (Java version) open source toolkit because many significant WS development tools are based on Apache Axis [3]. The tool which allows us to perform dynamic analysis automatically combines the Java application code at both the consumer and provider side into a local java program based on the involved WSDL documents. The software tester can then carry out test case selection without manually changing any application code.

Regression test and RTS techniques are generally employed for changes involving enhancements, corrections, and other maintenance related reasons rather than major system overhauls. In the realm of Web Services, WSDL specifies service interfaces, which should remain fairly stable through maintenance cycles. WSDL changes, since they require large changes to underlying code and are generally thought of as major changes in the realm of Web services, are outside of the scope of this regression test selection technique.

The remaining of this paper consists of the following sections. Section II provides background information about RTS techniques and the Apache Axis toolkit. In Section III, we discuss related work. Section IV discusses the overall approach. Section V describes how a local Java program is transformed from the given WSDL and the Web service application. Section VI describes the implementation details. Section VII gives an example of the use of our simulation tool. Section VIII concludes.

II. BACKGROUND

In this section we will describe the mechanisms leading up to the safe RTS for Java software, as well as a brief discussion of the most pertinent parts of the Apache Axis toolkit.

A. RT, RTS, and RTS for Java

Software testing in general is the process of executing a given program P in the attempt to reveal failures in the program. A test case is a set of inputs for P along with the corresponding expected output of program P . A test suite is a set of test cases. A test run is the execution of P with respect to a test suite T . The adequacy of each test suite is often determined by the percentage of P that the test suite covers.

Regression testing is the process of validating modified software to provide confidence that the changed parts of the software do not adversely affect the unchanged parts. Suppose we have program P , the modified version of P which we call P' , and a test suite T . Suppose also that after testing program P has a total of X faults. (Normally, X should be zero. However, it is possible that some required correction has not been made.) After modifying, we want to run those same tests in T on P' to determine what effect the modifications had on P' . Specifically, we are ensuring that after modification not a single *new* fault happens.

As mentioned earlier, RTS is a key component of most regression testing systems due to the fact that it helps to limit the cost of the testing. The most straightforward method of

regression testing is to retest all original test cases. That is to run the entire old test suite T on P' . However, this can often be very expensive especially considering that there can be a very large number of service interactions within any given enterprise. RTS techniques attempt to reduce the cost of RT by selecting T' , a subset of T , and using T' to test P' . If the cost to run the original testing suite is more expensive than the cost of running the reduced testing suite plus carrying out the deployed RTS computation, then a measurable cost reduction is achieved by performing the RTS technique.

Many of the RTS techniques use information about the program's source code to select T' . For instance, one such mechanism involves generating control-flow graphs (CFGs) from code. A CFG is a graph in which each node represents a code entity, and each edge represents the relationship in the flow of control from one entity to another. These entities can be statements, methods, or classes. Additionally, this control flow graph is supplemented with the coverage information regarding what entities that each test t covers. The RTS techniques compare their representations of P with P' to determine the set of dangerous edges. Dangerous edges are program entities that may behave differently under a single test case due to differences between P and P' . If a test case t does not cover any dangerous entity, then running t on P' will be the same as on P . It will behave in the same way in both P and P' . Thus it is safe to omit those test cases that covers no dangerous edge. This is precisely what is meant by a safe RTS. No modification revealing test case is left unselected in the reduction phase. Safe RTS techniques minimize the number of test cases while maintaining the same level of confidence provided by the retest-all regression testing.

The safe RTS mechanism we have selected is control-flow based. Typical control-flow based safe RTS performs three main steps.

- Step 1** It constructs a graph to represent the control flow, the type information of the classes under consideration.
- Step 2** It traverses the graph to identify dangerous edges. This traversal is performed by comparing the CFG for P and the CFG for P' . This comparison is performed using a dual-traversal.
- Step 3** Based on the coverage information derived from the Step 1, and the set of dangerous edges derived from the Step 2, it selects from the test suite those tests that need to be rerun.

This safe RTS mechanism can handle the features of the Java language such as: polymorphism, dynamic binding, and exception handling [2]. Additionally, their technique avoids analyzing and instrumenting code components, such as libraries, which are used but not modified. They consider every program as consisting of two parts: internal code and external code. Only internal code is analyzed. Their RTS system is safe because it follows a set of assumptions as it applies its extensions for the Java language to control-flow graph based RTS. These assumptions are: 1) Java reflection is not applied to any internal classes; 2) All external classes must be independent; 3) All test runs must be deterministic.

In order to extend CFGs to handle features of the Java language the following extensions were created: 1) variable and object type information, 2) internal or external methods, 3) interprocedural interactions through calls to internal or external methods from internal methods, 4) interprocedural interactions through calls to internal or external methods from external methods, and 5) exception handling. The graphs produced are called Java Interclass Graphs or JIGs.

B. Apache Axis toolkit

Since this work uses the Apache Axis [3] framework quite extensively, some of its more salient features will need to be discussed. One of the reasons Apache Axis was chosen is that it is well known and respected in the Web services community. Many significant software companies such as Apple, BEA, Borland, IBM, JBoss, and Oracle use Axis in their products which support Web services.

Axis shields the service developer from many of the details required when developing Web services, such as SOAP, WSDL, etc and allows them to concentrate on implementing the logic for their service. Developers deploy their services by providing a custom configuration file in Apache's Web service deployment descriptor (WSDD) format, and let Axis take care of the rest. Additionally, Axis provides a tool which generates client-side binding code from a WSDL file, which makes calling a Web service as easy as calling a local method.

Many Web services make use of the service deployment features provided by Axis to achieve a chosen service lifecycle, which Axis refers to as its scope. A Web service deployed in Axis can have one of three scopes: *request*, *session*, and *application*. The *request* scope informs Axis to instantiate a new service instance for each request. The *application* scope informs Axis to use a singleton service instance to handle all requests. The *session* scope informs Axis to use a singleton service to handle all requests by a single client. The request scope is the default and is totally stateless, while the other two maintain state across multiple requests. Axis can support maintaining session state using two mechanisms: either HTTP-cookie based, or the SOAP-header based.

Axis comes with a tool, WSDL2Java, which generates wrapper classes for the Web service according to the WSDL description of the service. This tool generates both client-side and server-side bindings, along with all the Java classes necessary to form a proxy between Java code and Web service standards, such as SOAP, HTTP, and WSDL. For each portType specified in the WSDL file, a Java Service Definition Interface (SDI) is defined, which is the interface the client programs use to access the operations of the service. A stub class implements the SDI, which contains the code which turns the method invocations into SOAP requests. Client programs can then call this stub as if it were a local invocation. The client program does not instantiate a stub directly. It instead instantiates a service locator and obtains a stub object from the locator. This locator is derived from the

service clause in the WSDL. WSDL2Java generates a locator component from each service clause. The service *locator interface* defines a factory method for each port listed in the service element of the WSDL. The *locator class* is the implementation of the locator interface, and implements the factory methods. It serves as a locator for obtaining stub instances.

On the server side, WSDL2Java generates skeleton classes and deployment descriptors (deploy.wsdd) in addition to the files normally generated for the client-side. Skeleton classes are Java objects that sit between the Axis engine and the service logic (or implementation). These skeleton classes are how Axis transforms WSDL-invoked calls to the service's implementation and back.

III. RELATED WORK

Among the recent regression testing techniques and tools, the majority focus on test case generation and actually performing the regression testing on services, such as the approach proposed in [4]. To date, very few techniques or tools are for applying RTS to Web service applications.

Our most closely related work was presented by Tsai et al [5-10]. They reported a framework which requires the use of enhanced WSDL specifications [9] which includes additional information such as dependency information, function descriptions, invocation, and concurrent sequence specification. In [7] they focus on what they call scenario-based testing with distributed agents. Scenarios are execution traces through a system [5]. They use the dependency information provided in [9], an enhanced UDDI server [10], with distributed agents to remotely execute test cases. The UDDI server is used to determine when the services are modified, thus need to be tested. In [6], they discuss scenario-based regression testing. This framework is described as rapid and adaptive end-to-end regression testing. The RTS technique used in that framework is scenario-slicing which is not safe [11]. Additionally, the mechanism with which they connect services and applications together is performed manually through the creation and use of scenario-slices [7]. Our approach, in contrast, is safe, works with Java Web services without requiring any additional information other than what is provided by Axis and the source code, and is provided programmatically.

The code-based RTS for component-based software by Harrold's team [12] is also a significantly related work. Their approach revolves around what the authors refer to as meta-content. This meta-content is additional data expected to be provided by vendors in lieu of source code in such cases where that content may be restricted by patent or copyright law. Their required meta-content is threefold: edge coverage achieved by the test suite with respect to the component, component version, and a way to query the component for the edges affected by the changes between the two given versions. This code-based approach is a white-box testing using code

where applicable and using meta-content where necessary. This is a safe RTS. However, it cannot be directly applied to Web services due to Web services being composable in nature. In a Web services world, an application could call a service, which in turn calls other services. In such an environment, an application can be affected not only by the changes in the services it directly calls on but also by the changes in the services that are used indirectly. Thus, just querying about changes in every called component (service) is not sufficient.

IV. THE OVERALL APPROACH

In this section, we will discuss the overall approach we have developed to perform RTS on Java-based Web services. We follow the three main steps of the typical control-flow based safe RTS outline in the background, Section II.A.

First, we construct a control-flow graph. In our case, since we are working solely with Java-based Web services, we are using the Java-based control flow graph (JIG) defined by Rothermel and Harrold [2] as described in Section III. JIG are created by performing a static and dynamic analysis of the involved code. This is performed by first analyzing the Java code to determine what service is going to be called, then using a lookup tool to determine where the corresponding code is for that service. The lookup tool simply uses information about URIs in the system and their corresponding Java code locations. Since constructing JIG needs to determine the specific runtime choice of the objects at every polymorphism point in each test case, dynamic analysis of the code has to be carried out. Therefore, we simulate the end-to-end Web service application with a automatically generated local Java program that replaces every remote invocation between a calling client and a called operation of a service with local a method call. This simulation tool is described with more detail in the Section V. Once we have the local simulation Java program, we can generate the JIG, along with the table that matches tests with the edges they cover using the algorithm given by Rothermel and Harrold [2]. The generated JIG represents the end-to-end Web service application.

The second and third steps are performed in the exact same manner as the manner described by the developers of the technique for RTS for Java [2]. In step 2, we compare the newly created JIG with the one created the last time it was generated, which we call the old JIG. We perform a dual-traversal of both trees to determine what parts of the tree are different. Upon performing this step, we recognize a set of dangerous edges and pass it on to the third step.

In the third step, the table of edges covered by the tests is compared with the set of dangerous edges to determine which tests need to be performed.

V. SIMULATING AXIS WEB SERVICES IN A SINGLE JVM

In this section, we will discuss the simulation tool in detail.

In short, the simulation tool provides a simple proxy class enabling the client to bypass the complications involved in message passing and interact with the service locally (in the same JVM).

A. Localize Web service applications

As described in the second part of the background, the WSDL2Java tool generates all the required classes for a Web service, which can be then connected in an end-to-end application as in Figure V.1.

The stub, skeleton, and actual service classes all implement the same Service Definition Interface, or SDI. By actual service classes, we are referring to the actual code on the service side that implements the interface described by the SDI. The stub class resides in the same JVM as the client application and is used as a proxy to the actual Web service. On the other service side, the Axis server engine takes care of the XML to Java object mapping in accordance with the information in the deployment specification (the `deploy.wsdd` document generated by WSDL2Java), loads the skeleton, and the actual service classes into its JVM, and invokes the methods specified in the SDI.

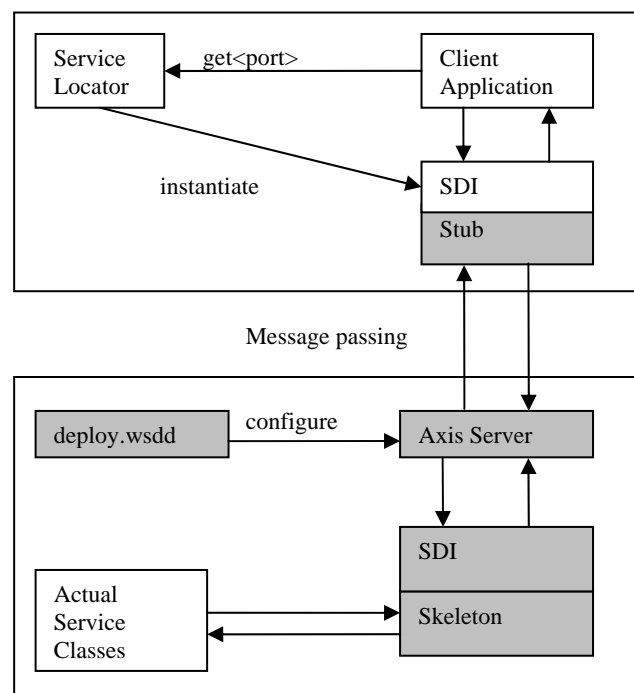


Fig. V.1. Server-Client interactions through components WSDL2Java Generated

In order to simulate the Web service with local objects in Java, we need to replace the SOAP messaging with local method invocation. Consequently, many components, such as the stub on the client side, the Axis server, and the skeleton on the server side, can all be removed (the shaded elements in Figure V.1). The stub and skeleton objects are not actually removed, but rather combined into a single local proxy object,

as shown in Figure V.2. The consumer and provider of the Web service are then running in a single JVM. Since the actual SDI remains the same, the necessary modifications to the system are transparent to both the consumer and provider. Since this is transparent, almost no direct code modification must be performed.

We intentionally preserve the “service locator” for two reasons. First, by preserving the service locator, changes in the client application can be completely avoided. Second, it is useful in simulation service deployment scope. We preserve the service locator, but we must modify so that it returns the reference of the local proxy object (instead of the stub object) to the client application.

The most central piece of the simulation tool is the local proxy object, which must comply with the service definition interface, or SDI. The SDI is generated according to the service specification set forth in the WSDL document. Similarly, the local proxy object will be generated by the simulation tool according to both the given WSDL document and the deployment descriptor.

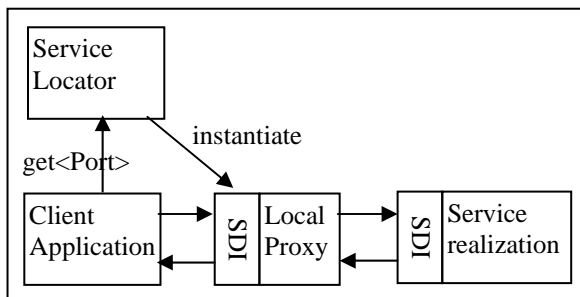


Fig. V.2. Localizing a Web service application

B. Simulating Web server features

The behavior of a Web service is determined by not only the code that implements the service but also the deployment option (scope), as described in the second part of the background. This deployment option provides the opportunity to predetermine what type of state behavior this Web service will support. Being state-aware prevents us from simply delegating each client call to the service provider object directly. This complexity must be carefully considered. In the simulation tool, we defined a special class (SessionEnabledService) that implements this state behavior if necessary. This special class will be described in more detail in section VI.2.

C. Invariance of Control-flows

As mentioned earlier, the point of simulating the behavior of the entire program is so that existing RTS techniques can be directly applied to the entire program. This needs to be done, because in order to generate the Java Interclass Graph (JIG) required by the RTS algorithm, the simulation program must be executed in order to perform dynamic analysis [2]. In our case, this execution will be performed using the simulation program and not the original system. This is

perfectly acceptable since both the simulation program and the original system will have identical JIGs. In performing the transformation, the code that performs the actual work on both the server and client side will not have changed. The only code being modified in the simulation program is the part of the code that handles the SOAP messaging (packing and unpacking Java calls in and out of SOAP messages). Clearly, if the WSDL does not change, then this code will never need to be retested.

Additionally, the mechanisms with which the client and server communicate are no longer RPC calls, but local Java method calls. This is purely a communication mechanism change. Although a client and its service synchronize, the service provider’s only role is to act on behalf of the caller. Gregory Andrews pointed out that conceptually it is as if the calling process itself was executing the call, and the synchronization between the caller and service is implicit [13]. This observation clearly justifies our replacement of the service calls with Java method invocations.

Lastly, this process of performing RTS is purely to determine which tests need to be rerun. The tests that need to be rerun will be executed on the deployed system. This is done to ensure that nothing is left out during a test which will skew the test results.

VI. IMPLEMENTATION

Our simulation tool has been implemented by modifying and extending the Apache Axis open source toolkit.

A. Code generation classes for simulation

Table VI.1 lists the Java classes in our toolkit that are used to generate the files.

B. Implementation for simulating the session behavior

To simulate the session behavior, the simulation tool defines an abstract class `SessionEnabled-Service`, which extends the `Service` class defined by Axis. The important new fields and methods of class `SessionEnabledService` are listed in Figure VI.1.

The service locator generated by our simulation tool extends the `SessionEnabled-Service` class. A session is identified by the target service URL and the `Service` object. The `SessionEnabledService` achieves this by using the instances’ scope `Hashtable` object – `sessions` – to store the sessions indexed by target service URL.

The `prepareContext` method is the core of the `SessionEnabledService` class. In local simulation environment, whenever the client application invokes a method of the SDI through the local proxy, the local proxy calls this method first to prepare the simulation context.

The `prepareContext` method does the following things:

- 1) *creating a new MessageContext;*

A `MessageContext` is a structure which contains three important parts: (1) a "request" message, (2) a "response" message, and (3) a bag of properties.

Table VI.1. Toolkit Classes for Code Generation

Class	Description
JavaConfigWriter	generates a config.properties file as deploy.wsdd and undeploy.wsdd. It contains the chosen scope option.
JavaLocalProxyWriter	generates a local proxy class from a WSDL binding. The generated file name is the binding name with a suffix "StubLocal".
ModifiedJavaServiceImplWriter	A subclass of JavaService-ImplWriter that generates a service locator that returns different types of stubs according to the configuration in the config.properties file.
ModifiedJavaServiceWriter	A subclass of JavaService-Writer that uses ModifiedJavaServiceImplWriter instead of JavaServiceImplWriter
ModifiedJavaGeneratorFactory	A subclass of JavaGenerator-Factory that uses ModifiedJavaServiceWriter instead of JavaServiceWriter, adds JavaConfigWriter to the definition writers, and adds JavaLocalProxyWriter to the binding writers.
WSDL2Java	A customized command line tool, derive from org.apache.axis.wsdl.WSDL2Java and use ModifiedJavaGeneratorFactory instead of JavaGeneratorFactory.

- 2) *saving a "fake server" engine in the MessageContext;* The FakeEngine class extends Axis server engine AxisServer. It simulates the real Axis server engine, provides the service realization component a chance to access the application-scope session. When the FakeEngine is instantiated the first time, it will create a default server-config.wsdd file in the run-time folder. The simulation tool users can customize the engine by editing the file as necessary.
- 3) *setting other necessary properties in the MessageContext;*
- 4) *checking the configuration, gets a Session if necessary. Otherwise, instantiating a SimpleSession object (for non-session scope service);*

The service might try to get the Session object, even if the client application does not maintain the session. In that case, the service can still use the Session object, but the objects that are saved in the Session will not be available for any other requests.

- 5) *getting the service object according to the chosen scope, by calling getNewServiceObject, retrieving from the session, or calling getApplicationScopedObject;*
- 6) *saving the service object in the Session object obtained in step 4);*
- 7) *saving the Session object in the MessageContext;*
- 8) *saving the MessageContext for the current thread.*

```

protected int _callStyle = UNSET;

protected int _scope = UNSET;

protected void config(int callStyle, int
scope);

protected void config(String resourceName);

private Hashtable sessions = new Hashtable();

private synchronized Session getSession(
String endPointURL);

protected abstract Object
getNewServiceObject(
Class targetServiceInterface);

protected abstract Object
getApplicationScopedObject(
Class targetServiceInterface);

public void prepareContext(Stub stub);
private static AxisEngine serverEngine =
new FakeEngine();

protected AxisEngine getServerEngine();

protected void setupMessageContext(
MessageContext mc);

```

Fig. VI.1. Important new fields and methods in class SessionEnabledService

To make use of the SessionEnabledService, the service locator class generated by the toolkit extends the SessionEnabledService class. It implements the getNewServiceObject method by instantiating a new service realization object with matched SDI.

Essentially, the local proxy simulates the dispatching functions of the Axis server. Although we did not have the formal specification of the Axis server, there have been numerous documents explaining the functionality of the Axis server. We validated our simulation tool by code inspection and numerous test cases.

VII. EXAMPLE

In this section, we apply our approach to a group of Web

services along with two applications. This is a dramatically simplified purchase order system. (When we applied our tool to a simple working purchase order system, the resulting JIG exceeds a hundred nodes and edges which is much too large to present here.) We believe the simplified system is able to shed enough light on our approach.

The simplified purchase order system consists of three Web services and two applications. The three services are: 1) a hardware provider that accepts an order of a hardware product; 2) a software provider that accepts an order of a software product; and 3) an ordering service that accepts an order and forwards the incoming order to the correct service depending on the type of the order.

In addition, there are two applications: 1) Application 1. It orders items and prints out the results. 2) Application 2. It orders items and determines if the call is successful and prints out helpful user messages. In a sense, a simple proxy to the order service would be similar to Application 1. An application interacting with a human operator will be similar to Application 2.

In the following discussions we will use pseudo-code rather than the actual Java code to simplify the discussion. The names, the JIG diagrams, and the pseudo-code of these five programs are listed in the following table.

```

S1 (Software service)
1 Order(item) {
2   if (item exists) {
3     if (item is in stock) {
4       order item;
5       return successful;
6     } else {
7       return error("ERROR:104");
8     }
9   } else {
10    return error("ERROR: 109");
11  } }
    
```

Table VII.1 Service and applications JIG and Abbrev.

Program	Name	JIG
Software provider service	S1	Fig. VII.1
Hardware provider service	S2	Fig. VII.2
Ordering service	S3	Fig. VII.3
Application 1	A1	Fig. VII.4
Application 2	A2	Fig. VII.5

```

S2 (Hardware service)
1 Order(item) {
2   if (item exists) {
3     if (item is in stock) {
4       order item;
5       return successful;
6     } else {
7       return error("ERROR:106");
8     }
9   } else {
10    return error("ERROR: 109");
11  } }
    
```

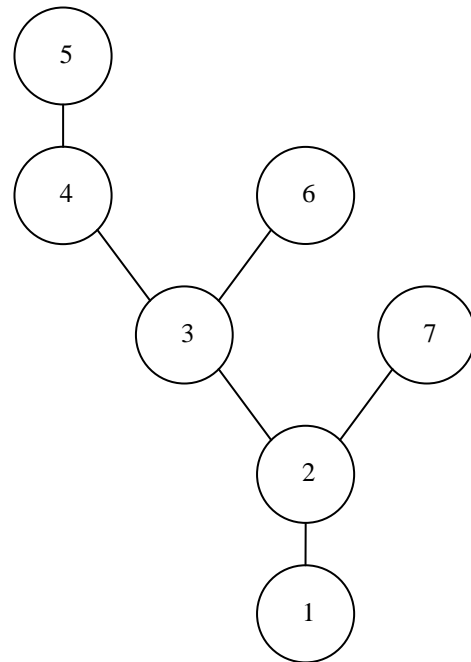


Fig. VII.1. The JIG of S1 and S2

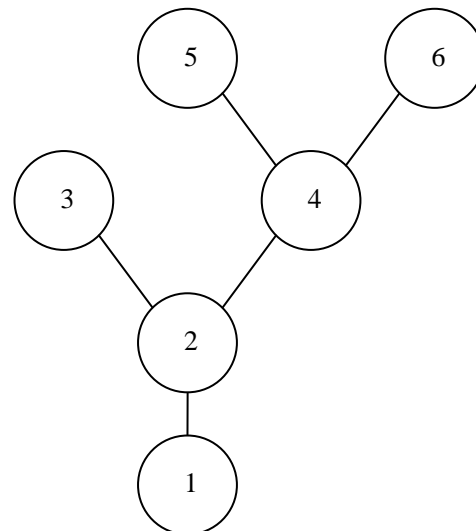


Fig. VII.3 The JIG of S3

```

S3 (Ordering service)
1 Order(item) {
2   if (item is software) {
3     return call S1;
4   } else if (item is hardware) {
5     return call S2;
6   } else {
7     return error("ERROR: 34");
8   } }
    
```

```

A1 (application 1)
1 Order(item) {
2   if (parse(item) is successful) {
3     print call S3;
4   } else {
5     print error("item is not acceptable");
6   } }
    
```

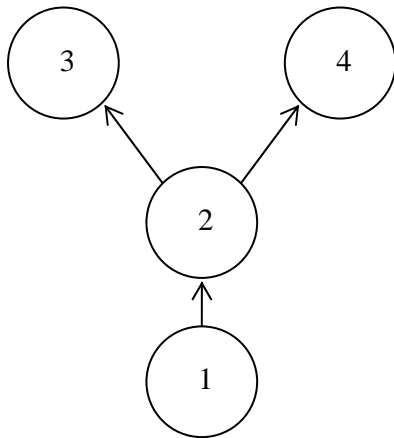


Fig. VII.3. The JIG of A1

```

A2 (application 2)
1 Order(item) {
2   if (parse(item) is successful) {
3     result = call S3;
4     if (result is an error) {
5       parse(error);
6       print error message based
7 on result;
8   } else {
9     print "Success";
10  }
11 } else {
12   print error("item is not
13   acceptable");
14 } }
    
```

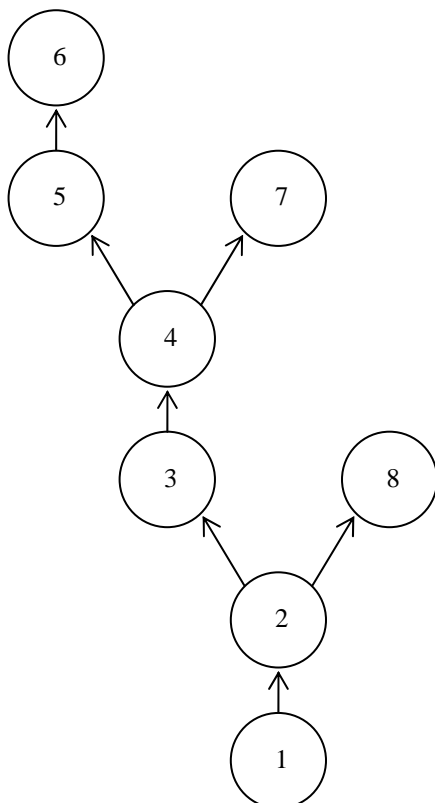


Fig. VII.4. The JIG of A2

Note the line numbers are not part of the pseudo code but the correspondence between the JIG nodes and the pseudo code statement. Note also that all the services return a message indicating success or failure. This is indicated in the graphs by an undirected arc between the two nodes. The control flow percolates up and then the control flow returns back to the start node. However, in applications A1 and A2, the control flow moves only in one direction, which is denoted by directed arrows.

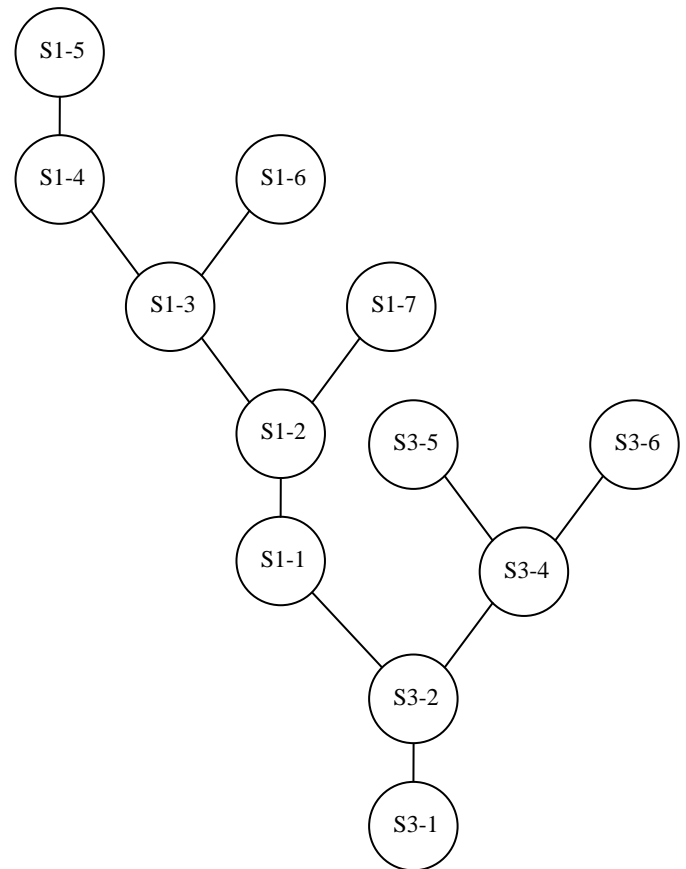


Fig. VII.5. Composite JIG for S3 and S1

Having all of the individual JIGs, we can compose the global JIGs for the end-to-end applications. In the applications and composite service (service S3), the node corresponding to the service call statements are “call nodes”. To assemble the global JIGs, we replaced the call node with the JIG corresponding to the called service. In the composite JIGs, we attached the program name as the prefix of the label of every node belonging to program. For instance, label S1-1 denotes node 1 in S1.

Inserting S1 into S3, then inserting S2 into S3, we obtain the end-to-end JIG of the composite service S3. This JIG is contained in the JIG shown in Figure VII.6. To see this JIG, the reader should simply ignore the three nodes at the lower part of the diagram, namely nodes A1-1, A1-2 and A1-4.

The global JIG of A1 and S3 is obtained by inserting the JIG for S3 into node 3 of the JIG for A1 as shown in Figure

VII.6. Similarly, the global JIG of A2 and S3 is obtained by inserting the JIG for S3 into node 3 of the JIG for A3.

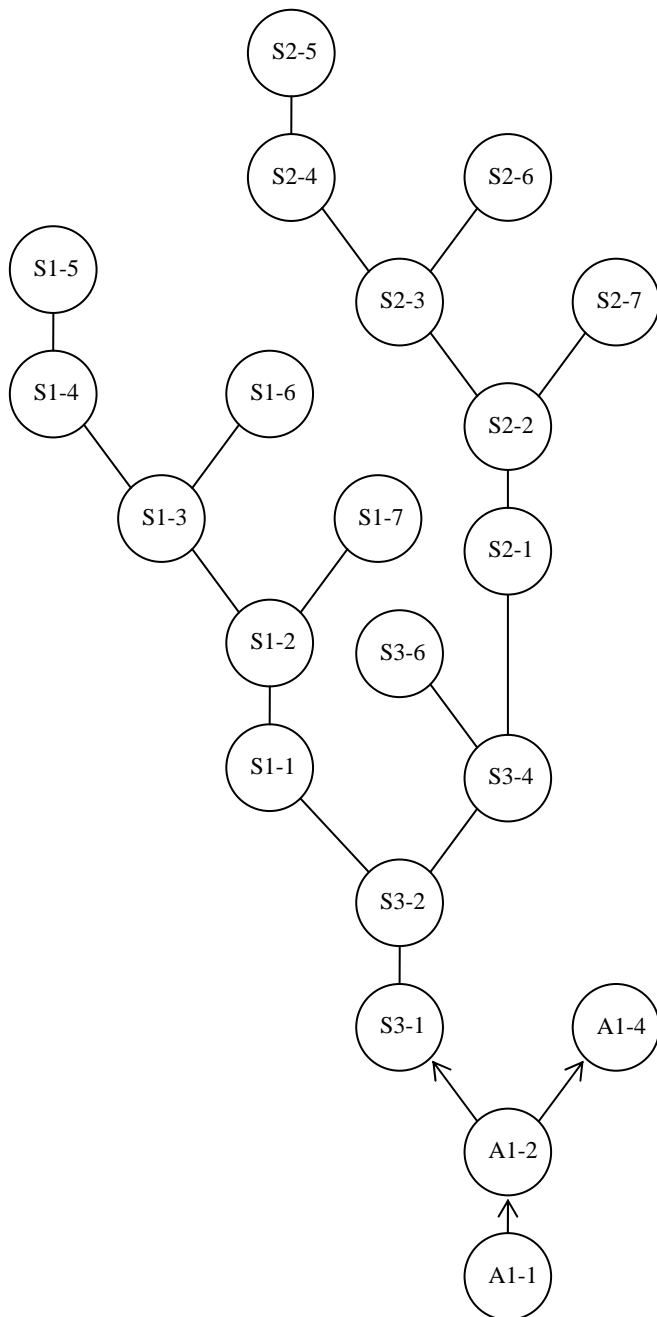


Fig. VII.6. Complete JIG of A1

Once we have the global JIGs, we can apply the RTS method for Java directly to determine which parts of the code have been modified, and therefore which tests must be rerun [2].

Suppose we have tested this system many times and have performed the maintenance required to have a completely working system that produces no error under testing by the testing suite. Suppose also we have a separate test case targeting each line of pseudo code. This assumption is a simplified basis for estimation of the cost saving. Under such

an assumption, in the overall testing suite, there are 7 tests for each of S1 and S2, 18 tests for S3, 4 tests for A1, and 8 tests for A2.

Suppose we need to modify the error code at line 6 of S1 for some reason. This change in S1 triggered the production of a new version of JIG, S1'. The pseudo code of S1' is shown below.

S1' (Software service after modification)

```

1 Order(item) {
2   if (item exists) {
3     if (item is in stock) {
4       order item;
5       return successful;
6     } else {
7       return error("106--ERROR");
8     }
9   } else {
10    return error("ERROR: 109");
11  }
12 }

```

For S1', we need to test the path that leads to 6, which in this case is the path 1-2-3-6 in the JIG shown in Figure VII.7, where The change is highlighted at node 6. We can fully test S1 by running only 4 tests rather than 7. For S1, the savings is only 43%. However, for the services that use S1, more savings can be achieved.

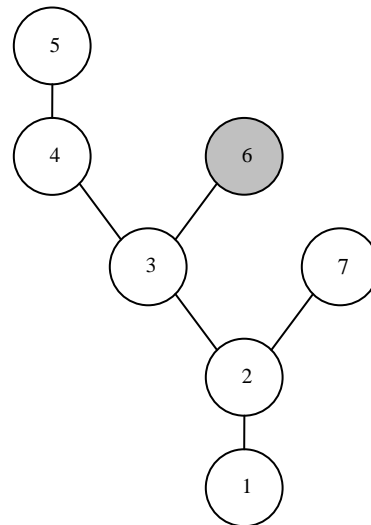


Fig. VII.7. S1 JIG after change

For S3 (see S3's JIG in Figure VII.6 by ignoring nodes A1-1, A1-2 and A1-4 as explained earlier), only the path leading up to the node in question (node 6) must be tested. In this case, it is (S3-1) – (S3-2) – (S1-1) – (S1-2) – (S1-3) – (S1-6). This means a total savings of 12/18, or approximately 67% was achieved. For the two applications, A1 and A2, the savings that were achieved were 17/25, or approximately 68%, and 13/21, or approximately 62% respectively.

Lastly, we like to emphasize the importance of end-to-end testing. In the example we discussed, we described a very specific change in one of the services, namely S1. This

change involves altering the response code. Obviously, this change will have no effect in the testing of S1, S3, and A1. However, when A2 parses the resulting codes, an error was produced at Line 5 because this part of the code parses the response code against a predefined format. This situation highlights the importance of performing end-to-end testing. A change somewhere in the system can produce errors anywhere along the path of execution.

As more and more business processes utilize Web services to deliver and compose functionality, ensuring that they actually deliver the functionality that was promised is always critical. Regression testing has shown itself to be a key ingredient to ensuring that changes do not produce undesired effects any where in the system. Consequently, safe RTS techniques become even more important because they reduce the amount of testing time for each modification while still guaranteeing that the reduction of tests does not harm the quality of the tests. In this paper, we have presented an approach to apply a safe regression test selection technique to Java Web services.

VIII. CONCLUSION

The approach transforms the service code into a local Java program to perform dynamic analysis of the code in an end-to-end manner. The result of the static and dynamic analysis is an end-to-end Java interclass graph, which can then be fed into the safe RTS technique for Java [2]. The overall accomplishment is a safe mechanism to reduce the number of tests which need to be rerun to the minimum number of tests that will provide the same level of confidence as not removing tests at all.

In this implementation, our focus was solely on Web service applications written in Java and deployed in Axis. Web services are platform and language neutral, and in our future work we will generalize this very approach to be capable of supporting platform and language neutrality.

ACKNOWLEDGMENT

The prologue of this work was supported in part by a grant by Lockheed Martin Corp in 2003.

REFERENCES

- [1] Myers, G. J. 1979 *Art of Software Testing*. John Wiley & Sons, Inc.
- [2] M. J. Harrold, et al, "Regression test selection for Java software", *Proceedings of OOPSLA'01*, Tampa Bay, FL, Oct. 2001, pp 312-326.
- [3] Apache Axis project, <http://ws.apache.org/axis/>
- [4] M. Bruno, et al, "Using Test Cases as Contract to Ensure Service Compliance across Releases", *Proceedings of 3rd International Conference of Service Oriented Computing*, vol 3826, pp. 87-100, 2005.
- [5] W. T. Tsai, et al, "Scenario-based modeling and its applications", *Proceedings of the Seventh International Workshop on Object-Oriented Real-Time Dependable Systems, (WORDS 2002)*, pp.253-260, 2002.
- [6] R. Paul, et al, "Scenario-Based Functional Regression Testing", *Proceedings of 25th COMPSAC*, pp 496-501, 2001.
- [7] W. T. Tsai, et al, "Scenario-Based WS testing with distributed agents", *IEICE Transactions on Information and Systems*, 2003.
- [8] W. T. Tsai, et al, "End-to-end integration testing design", *Proceedings of 25th COMPSAC*, pp.166-171, 2001.
- [9] W. T. Tsai, et al, "Extending WSDL to facilitate Web services testing", *Proceedings of 7th IEEE International Symposium on High Assurance Systems Engineering*, pp. 171- 172, 2002
- [10] Tsai, W.T.; Paul, R.; Cao, Z.; Yu, L.; Saimi, A., "Verification of Web services using an enhanced UDDI server", *Proceedings of the Eighth International Workshop on Object-Oriented Real-Time Dependable Systems,(WORDS 2003)*, pp. 131- 138, Jan. 2003
- [11] G. Rothermel and M. J. Harrold, "Analyzing regression test selection techniques", *IEEE Transactions on Software Engineering*, vol.22, no.8, pp.529-551, Aug 1996
- [12] A. Orso, et al, "Using component metacontent to support the regression testing of component-based software", *Proceedings of IEEE International Conference on Software Maintenance*, pp 716-725, 2001.
- [13] G. R. Andrews, "*Concurrent Programming*", Chapter 9, Addison-Wesley Pub, Reading, MA, 1991.

Author Biographies

Michael E. Ruth received his Bachelor of Science and Master of Science in Computer Science at the University of New Orleans in 2002 and 2005 respectively. In 2003, he was awarded the Crescent City Doctoral Scholarship. He is currently a Research Assistant and PhD candidate in Engineering and Applied Science at the University of New Orleans. His research interests include Web services, Distributed Systems, software testing and Software Engineering.

Feng Lin was born in Xiamen, in the Fujian province of China. He completed his Bachelor of Science degree in Electronic Engineering at Tsinghua University in 2003. He received his Master's degree in Computer Science from the University of New Orleans in 2006. His current research interests include Web services, Distributed Systems, and Software Engineering.

Shengru Tu is a professor of Computer Science at University of New Orleans. He received his Ph.D. in Electrical Engineering and Computer Science at the University of Illinois at Chicago in 1991. His research interests include service-oriented computing, enterprise software integration, geographic information systems, static analysis of concurrent programs, and Petri net theory. He recently co-edited the special issue of *IEEE Internet Computing* on Web services for GIS.