8-8-2007

Automating Regression Test Selection for Web Services

Michael Edward Ruth
University of New Orleans

Automating Regression Test Selection for Web Services

A Dissertation

Submitted to the Graduate Faculty of the
University of New Orleans
in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy
In
Engineering and Applied Science

By

Michael Ruth

B.S. University of New Orleans, 2002

August, 2007

This dissertation is dedicated to
Rebecca, my family, and friends.

Acknowledgement

I would like to thank Dr. Shengru Tu, my advisor, for providing me with the guidance needed to see this research project through to its completion.

I would also like to thank Dr. Vassil Roussev, Dr. Nauman Chaudhry, Dr. Huimin Chen, and Dr. Mahdi Abdelguerfi for being a part of my dissertation committee.

I would also like to show my appreciation for the assistance of Feng Lin, Sehun Oh, Adam Loup, Olin Gallet, Brian Horton, and Marcel Mata.

I would also like to thank Rebecca, my wife, without whom I would still be wearing white socks with dress pants.

Also, my friends have my gratitude for helping me through both good times and bad, and allowing me to disappear for long periods of time to work on my dissertation.

Lastly, and most importantly, I would like to thank each and every member of my family for their support and understanding throughout.

Table of Contents

List of Figures

List of Tables

Abstract

As Web services grow in maturity and use, so do the methods which are being used to test and maintain them. Regression Testing is a major component of most major testing systems but has only begun to be applied to Web services. The majority of the tools and techniques applying regression test to Web services are focused on test-case generation, thus ignoring the potential savings of regression test selection. Regression test selection optimizes the regression testing process by selecting a subset of all tests, while still maintaining some level of confidence about the system performing no worse than the unmodified system. A safe regression test selection technique implies that after selection, the level of confidence is as high as it would be if no tests were removed. Since safe regression test selection techniques generally involve code-based (white-box) testing, they cannot be directly applied to Web services due to their loosely-coupled, standards-based, and distributed nature. A framework which automates both the regression test selection and regression testing processes for Web services in a decentralized, end-to-end manner is proposed. As part of this approach, special consideration is given to the concurrency issues which may occur in an autonomous and decentralized system. The resulting synchronization method will be presented along with a set of algorithms which manage the regression testing and regression test selection processes throughout the system. A set of empirical results demonstrate the feasibility and benefit of the approach.

Keywords: Regression Testing, Regression Test Selection, Safe Regression Test Selection, Web services, Concurrency, Coverage Conflict, Test Inconsistency

Chapter 1: Introduction

1.1 General Introduction

Web services have enabled business workflows to be extended beyond the boundaries of companies and organizations. A single business process can be realized by utilizing possibly many different Web services either directly or indirectly in workflows which may flow internally or externally in/out of the enterprise. Since the business world often involves very rapid change to keep up with current market conditions, the business processes inevitably need frequent adjustment, along with their supporting Web services. These rapid adjustments, or modifications, must also be supported by rapid verification in order to provide a desired level of quality assurance. Every time the system is modified, we must ensure that the modification does not have an adverse affect on any unmodified areas, or regions, of code (the modification does not introduce new problems into the code). Typically, this is done by running the test cases previously used to test the system prior to modification again. This processing of "retesting" is called regression testing and its goal is to determine whether or not the system has been made worse by the modification.

One of the key ideas associated with regression testing is to reduce the number of tests which has to be rerun to ensure that the system performs no worse than it did before the modification with some level of confidence. Reducing the number of tests to be rerun is called regression test selection. The need for regression test selection has been well established for traditional software systems [1]. Furthermore, a safe regression test selection technique which ensures that the level of confidence provided by the selection mechanism is no worse than not removing any test cases, preventing any potentially modification-revealing and thus possibly fault-revealing, test cases from being left unselected in the regression test selection process.

Although there are several safe regression test selection techniques which have been proven effective for traditional software applications [2], there has been very little work on safe regression test selection for verifying Web service systems. This is a cause for concern especially considering that regression testing Web services can be much more costly than doing so for traditional software (all calls must be marshaled in and out of SOAP and sent over a network).

1.2 An Ideal Regression Test Selection Technique for Web Services

An ideal regression test selection technique for the verification of Web service systems would have the following properties: 1) safe, 2) interoperable, 3) composable, 4) decentralized, 5) end-to-end, and 6) automated. Hardly any safe regression test selection techniques are available for Web services because safe regression test selection techniques involve code-based, or white-box, testing which is unlikely for a number of reasons. There can be service interactions in which two or more of the services interacted are in different languages on different platforms or service providers who can never be counted on to share their source code due to copyright and other legal concerns.

Interoperability, or the ability of services to interact with other services which are in different languages on different platforms, was the catalyst for the sudden popularity of Web services. However, this particular benefit of Web services is a detriment to safe regression techniques which require knowledge of the implementation of a system in order to ensure the safety of the technique.

Web services may be composed to form more complex services and this particular feature also presents a set of challenges. Since the explicit goal of regression testing is to ensure that a modification somewhere does not affect the system somewhere else it is important to know how the services are related to one another. Long call chains could exist and a modification in one subsystem may require testing in several other subsystems. If an entity is modified everything that depends on that entity must also be tested. It is important that testing occur at each and every end point along the way to ensure that all the dependencies of all modifications are tested. This is end-to-end testing and it gets its name because it tests from the point of view of every end point along the way. To facilitate this type of testing through composite services the regression testing and regression test selection technique must be as composable as the underlying services.

The autonomy and distributed nature of the Web services themselves, requires that the regression test selection and regression testing processes be decentralized because some service providers may wish to retain a desired level of control over the information they share. In general, not all services in an interaction will be developed by the same group and there may be interactions in which no central authority dictating the rules of engagement. This means that some service providers will be reluctant to provide any details of their service and cannot be forced to do so. In order to alleviate their concerns thus increase the likelihood of participation of more providers, the information which must be shared must be carefully considered. Therefore, a decentralized solution which carefully considers what information must be shared and how this information should be protected is essential.

Finally, the regression testing and regression test selection processes which by definition must be repeatedly run should be automated to ensure timely feedback for all modifications. Automating the regression testing and regression test selection processes also ensures that no modification is left untested. If every modification in the system is not tested automatically, a careless operator could forget to test a modification which could lead to serious verification issues later on. However, if the processes are automated this problem is alleviated.

1.3 Main Contributions

The main contributions of my work are threefold: First, a new safe regression test selection technique was developed for the verification of Web services in an end-to-end manner. Second, a new framework was devised to monitor and synchronize the distributed modifications in order to automate the regression test selection and regressing testing processes. Third, the concurrency challenges which present themselves in a decentralized, automated framework for performing regression test selection and regression testing were recognized and a set of solutions in the form of algorithms for agents to follow to handle the synchronization issues. In addition, I have established a basis for experimental studies by providing a collection of Web services systems, the lack of which has hindered healthy comparison studies of regression testing and regression test selection research.

In this dissertation, an approach to perform safe regression test selection in an end-to-end manner for the verification of both intra- and inter-enterprise Web services is proposed. The approach is based on a safe regression test selection technique developed by Rothermel and Harrold for monolithic applications using control flow graphs [1]. The proposed approach successfully ensures that safety is maintained by carefully ensuring the transitions from the model do not affect the safety of the proposed approach.

A control-flow graph-based approach is ideal for performing safe regression test selection for Web services because control-flow graphs are an ideal medium for addressing the interoperability and composability concerns. Control-flow graphs can be generated from any program written in any modern programming language, especially those used to develop Web services and since control-flow graphs are a special case of finite state machines, they can be composed. Control-flow graphs are interoperable because if two control-flow graphs were generated for two systems each built on different platforms and in different languages the two control-flow graphs could be composed.

The proposed approach particularly observes the autonomous, decentralized nature of Web service systems. The approach only requires three elements from each participant: Control-flow graphs of each the operations the service provides, test cases for those control-flow graphs, and coverage information which maps the provided control-flow graphs to the provided test cases. Each of the participants will maintain control over the granularity of the control-flow graphs they provide which can vary from very detailed (statement level) to very abstract, depending on the need for security (someone who does not wish to share information will only share a operation level control-flow graph, whereas within an enterprise everyone can share statement level control-flow graphs). In order to ensure higher levels of participation, the source code is shielded from the tester using hashes of the code. This allows the tester to determine if the underlying code was modified without being able to determine where or how the code was modified. These tactics to attract more independent participants is a novel approach in regression test selection.

This approach is designed to be automated by using a set of distributed agents, one each for every service and application in the system, which interact together to automatically perform both the regression testing and regression test selection processes over the entire enterprise. These agents perform their work by monitoring, exchanging, and updating the control-flow graphs of the participants in a publish-subscribe method. This allows this automated system to perform end-to-end testing because if any of the participating services are modified, each and every service or application which calls this service either directly or indirectly will be notified.

Automating the proposed approach presents an entirely new set of issues. More specifically, issues related to concurrent modifications become increasingly important. These issues are: 1) coverage conflict, 2) test inconsistency, and 3) communication issues. Coverage conflict issues arise from the manner in which the regression test selection technique is performed. More specifically, it is possible for one modification to conflict with another because of its location. Test consistency is related to the test cases having a consistent view of the system under test which implies that the system under test seen at the beginning of all the tests is the same system seen at the end of all the tests. These issues were carefully considered, recognized, and solved by synchronizing the modifications.

Since there was no preexisting benchmark for comparing regression test selection techniques for the verification of Web services, a group of systems was developed to perform an empirical study of the proposed approach. The benchmark was used to compare the proposed safe regression test selection technique with the select-all regression test selection technique. The select-all regression test selection technique is the technique in which the selection step is skipped and all tests are run every time. In order to be effective, a regression test selection technique must perform better than the select-all technique which means that the cost of performing the technique and executing the selected test cases must be less than executing all test cases without performing the selection step. The empirical study demonstrates that the proposed technique is feasible and can be effective. Also, observing the lack of a common benchmark for comparison studies of regression test selection techniques, this collection of systems can be a promising seed of future regression test selection benchmark models.

In summary, the proposed approach is unique in that it is the first safe regression test selection technique for the verification of Web service-based frameworks which is automated, decentralized, and end-to-end which manages the interoperability and composability of Web service frameworks. This approach is also novel because it is the first approach to use techniques to increase participation through information hiding. Also, the issues related to concurrent modifications were carefully considered and recognized for the first time. Lastly, a novel benchmark for comparing regression test selection techniques was developed and the first empirical study was performed on a regression test selection technique for the verification of Web services. The results of this empirical study were used to demonstrate that the proposed approach is feasible and can be effective in reducing the cost of performing regression testing.

1.4 Organization

The rest of the dissertation will be organized as follows: In chapter 2, the background information regarding Web services, regression testing, regression test selection, and control-flow based regression test selection techniques will be discussed in detail. Chapter 3 will provide a survey of related literature including topics such as: test case generation frameworks, automated test execution frameworks, mediums used to bridge disparate services, and a variety of regression test selection frameworks for Web services. In chapter 4, the approach to perform regression test selection for the verification of Web service-based frameworks will be discussed in terms of construction of the three elements (control-flow graph, test cases, and coverage information) for the entire system and in terms of how the framework operates once constructed. The material was published at the International Conference on Internet Applications and Web Services (ICIW) in 2007 under the title "A Safe Regression Test Selection Technique for Web Services". Chapter 5 will discuss the automation of the approach, the concurrency issues, and their solutions. The material is based on a recent paper, "Towards Automating Regression Test Selection for Web Services" which was accepted for publication at the Testing Emerging Software Technology (TEST) Workshop which was held in conjunction with IEEE Computer and Software Applications Conference (COMPSAC) 2007. The empirical study will be presented in chapter 6 including descriptions of the experiments, descriptions of the five systems developed for this purpose, and the results of the empirical study. Finally, chapter 7 will conclude and provide avenues for future work on this subject.

Chapter 2: Background

In this chapter, an introduction to Web services, along with background information on regression testing techniques including safe regression test selection techniques, will be provided in detail.

2.1 Web Services

Broadly, Web services refer to self-contained web applications that are loosely coupled, distributed, capable of performing business activities, and possessing the ability to engage other web applications in order to complete higher-order business transactions, all programmatically accessible through standard internet protocols, such as HTTP (Hypertext Transport Protocol), JMS (Java Messaging Service), SMTP (Simple Mail Transfer Protocol), etc [3]. More specifically, Web services are Web applications built using a stack of emerging standards that form a service-oriented application architecture (SOA), an architectural style whose goal is to achieve loose coupling among interacting software components through the use of simple, well defined interfaces. In [3], a stack of emerging standards on which Web services are built were described. Figure 2.1 shows a conceptual overview of the Web Services stack.

Extensible Markup Language (XML) provides the basis for most of the standards that Web services are based on. XML is a standard that has been developed by the World Wide Web Consortium (W3C) [4]. XML is a text-based meta-language for describing data which is extensible and therefore used to define additional markup languages. The mechanism with which a markup language is defined in XML is termed a schema definition. A schema definition is a set of rules that define the structure and content of an XML document. Since XML is text-based and extensible, it provides the standard on which other standards are built in the realm of Web services.

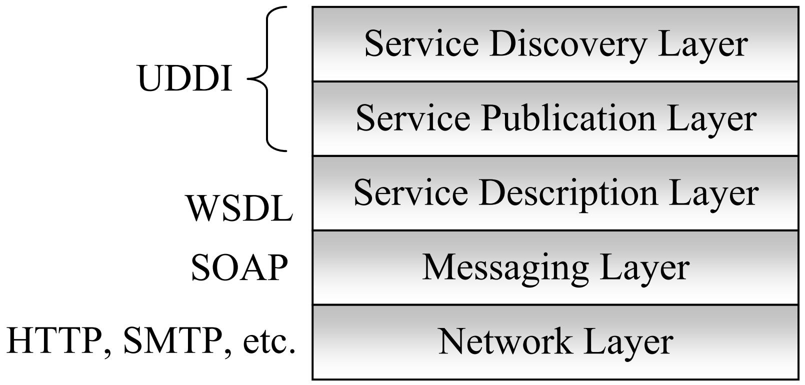


Figure 2.1: Conceptual Web services stack

The lowest layer of the Web services stack, the network layer, is defined since a Web service has to be network accessible in order to be invoked by its clients. Although Web services are typically thought of as operating over HTTP, they are also capable of operating over many different types of transport layers, such as HTTPS (Secure HTTP), JMS, and even SMTP, providing a great deal of flexibility to application developers. Although, just about any internet traversable transport layer can be used underneath Web services, HTTP is by far the most commonly used Web service transport.

The next logical layer in the stack is the messaging layer, and its related standard is SOAP [5]. SOAP defines a common message format for all Web services. SOAP is designed to be a lightweight protocol for information interchange among disparate systems in a distributed environment. The actual format consists of an envelope which define the contents of the messages and how to process those contents. In the envelope there are a number of standard headers, and a body. SOAP is entirely encoded in XML.

The minimum requirements of a service provider or consumer of Web services are to be able to build, process, and send (over the network layer) these SOAP messages.

The layer above the messaging layer is the description layer. Its specification is defined by the Web Service Definition Language (WSDL) [6]. It provides a mechanism for describing Web services in a standard way. The description provides an interface for using the Web services, in terms of available operations, their names, parameters and return types. The description binds a service, termed abstract endpoints in the specification, to concrete endpoints, which is a description of the service defined abstractly then bound to a concrete network protocol and message format. This description is represented using XML as well. This layer is the key element that gives Web services their loose coupling and allows for a new level of interoperability, platform and language neutrality [3].

The highest layer of the protocol stack is the discovery layer. It is modeled by the Universal Description, Discovery and Integration (UDDI) [7] specification. UDDI provides a means to locate and use Web Services programmatically. Service providers publish high level descriptions of their Web services into a UDDI repository, with which their services can be looked up and used. When an application wants to use a service published in the repository it downloads what the application needs to connect to and consume the Web services it found in the repository. These standards have addressed the connectivity, messaging, description, and discovery issues for Web services, providing the simple, well-defined interfaces required for the loosely coupled, interoperable building blocks known as Web services.

2.2 Software Testing, Types, and Levels

Software testing in general is the process of executing a given program P in an attempt to reveal possible failures in the program [8]. A test case is a set of inputs for P along with the expected output of program P when given said inputs. A test suite is a set of test cases, and a test run is the execution of P with respect to some test suite T. The adequacy of each test suite is normally determined by the level of coverage (normally a percentage) that the test suite covers the possible executions of P.

There are two basic types of software testing: Black-box testing and White-box testing [9]. The two types depend on the point of view of the tester when developing test cases. Black-box testing takes an external perspective of the test object to derive test cases, while white-box testing uses an internal perspective of the system. In black-box testing, the test designer selects valid and invalid input and determines the correct output using no knowledge of the test object's internal structure. While this method can uncover unimplemented parts of the specification, one cannot be sure that all existent paths are tested. White-box testing requires programming skills to identify all paths through the software. The tester would then choose test case inputs to exercise paths through the code and determine the appropriate outputs. Since the tests are based on the actual implementation, if the implementation is modified the tests will also need to be modified. Though this method can uncover an overwhelming number of test cases, it might not detect unimplemented parts of the specification or missing requirements. However, using white-box testing one can be sure that every path is exercised. The focus of this work is on using white-box testing largely because when the system is being regression tested, the system has usually been completely implemented.

There are several levels of testing, which refer to the level, or view, of the system being tested [8]. The level of the system under test refers to how the parts of the system are grouped together for testing, such as individually, or the entire system as a whole. The levels which will be discussed are as follows: unit, integration, system, scenario testing (also known as use-case testing), and end-to-end testing. Unit testing is a procedure used to validate that individual units of source code are working properly. A unit is the smallest testable part of an application. The goal of unit testing is to isolate each part of the program and show that the individual parts are correct. A unit test provides a strict, written contract that the piece of code must satisfy. As a result, it affords several benefits including: facilitating changes (making it easier to test modifications) and simplifying integration (helps to eliminate uncertainty in the units themselves, making bottom-up approaches simpler). However, unit testing will not catch every error in the program, since by definition it only tests the functionality of the units themselves. Thus, unit testing is only effective if it is used in conjunction with the other testing levels. In Web services testing, the smallest testable part of the system is testing each service independently of other services. If the service is a composite service, stub implementations would normally be used in the place of the composed services.

The next level of testing is integration testing in which individual software modules are combined and tested as a group. The purpose of integration testing is to verify functional, performance and reliability requirements placed on major design items. The general idea is a "building block" approach, in which verified assemblages are added to a verified base which is then used to support the integration testing of further assemblages. There are several different types of integration testing, but only the most common two will be discussed, which are: big-bang and bottom-up. The big-bang approach is when all or most of the developed modules are coupled together to form a complete software system or major part of the system and then used for integration testing. This approach has one major benefit which is that it takes less time to design and test if everything goes well. However, if everything does not go well it may be harder to pinpoint those problems. This approach is most often used when the system was developed before the need for integration testing. In the bottom-up approach, all the bottom or low level modules, procedures or functions are integrated and then tested. After the integration testing of lower level integrated modules, the next level of modules will be formed and can be used for integration testing. This is the more common of the two approaches and is normally applied when the system was developed with testing in mind. However, this approach requires all or most of the software to be completed before testing can begin. The major issue with integration testing is that any conditions not stated in specified integration tests, outside of the confirmation of the execution of design items, will generally not be tested and integration tests can not include system-wide modification testing. In Web services testing, this implies that the composed systems are tested in terms of their interactions with other services. If the service is a composite service, the composed services would be called rather than using stubs.

The next level of testing, system testing occurs when the system is integrated together to form the entire system. One can think of this as the final stage of integration testing with the exception that the system is not just being tested to determine if the parts work together correctly, but also whether or not the system works as a whole. The goal of system testing is not only to determine whether or not the components play well together, but also that the components work together to complete some goal. This implies that system testing is determining whether or not the functional requirements are met. It also determines whether or not the system meets its non-functional requirements as well, such as response times, etc. Testing the functional requirements of the system involves testing the system as it would actually be used, normally using either use-cases (black-box) or white-box approaches. This implies end-to-end testing, which simply means that the system end-points are being tested, using the actual components and control flows through the application from one end (the user) to the other (which completes the work). Scenario testing [10], a variation of system testing, uses the black-box approach which entails using design elements, such as use cases, to system test the application. In the realm of Web services, system testing refers to the testing of the services as they would be used by the various applications and services. This is precisely the testing level and approach which will be the focus of this work.

2.3 Regression Testing and Regression Test Selection

Regression testing is the process of validating modified software to provide confidence that the changed parts of the software behave as intended and that the unchanged parts have not been adversely affected by the modification [1]. Suppose there is program P, its modified version P’, and a test suite T. Suppose also that after testing program P has a total of X faults. After modifying P, resulting in P’, those same tests in T must be rerun on P’ to determine what effect the modifications had on P. In other words, the goal is to determine whether or not the system is made worse in terms of having not more than X faults. It is provably impossible to determine whether or not one could make the system better even if the system determines that fewer than X faults were the result of the modification.

There are five basic problems associated with regression testing: 1) Test case revalidation problem; 2) Regression test selection problem; 3) Test suite execution problem; 4) Coverage identification problem; 5) Test suite maintenance problem. This section will briefly cover all of the problems in some detail, but this work is focused on solving the regression test selection problem in the realm of Web services, and therefore will only discuss the regression test selection problem in full detail.

The test case revalidation problem is focused on maintaining test suite T. Any framework attempting to solve this problem would be trying to identify and remove obsolete test cases from T when the specification of P is modified. Test cases become obsolete when the test case no longer corresponds to a part of the system. Suppose that there is a system S, which has feature F, and at some point later in time the developers of S remove feature F. A framework which solved this problem would be able to identify that the tests which cover the old, but removed, feature F are no longer. Although technically, regression test selection and execution are next on the list, they will be skipped for the time being so the other problems can be discussed first.

The coverage identification problem is also focused on maintaining the test suite T, but it is the opposite of the revalidation problem. Any framework attempting to solve it would be trying to identify when new test cases are necessary, and when they are it would create them. Supposing there is a system S, and we create feature F for it. A framework which solved this problem would be able to determine that there is a new feature F and create test cases for the new feature.

The next problem focuses on maintaining the test suite, but at the overall test suite level rather than adding or removing groups of test cases. This particular problem is more of a combination of problems rather than its own problem on its own. Specifically, the test suite maintenance problem takes the result of the revalidation problem solution (a set of test cases to remove), the result of the coverage identification problem solution (a set of test cases to add), and the old test suite and amalgamate them together to form a new test case suite for P'.

The test suite execution problem is focused on actually performing the tests in the test suite. There are solutions to this in the Web services world, and they will be discussed in the related works section. The goal of anyone solving this problem is to test P' with T' in order to establish the correctness of P' with respect to T'. A basic solution to this problem must be solved in order to attempt to solve the regression test selection and show how effective regression test selection can be, so a solution to this will be provided at least basically in the sections detailing implementations.

Finally, the focus of this work will be discussed, which is the regression test selection problem. As mentioned earlier, regression test selection is a key component of most regression testing systems because it helps to reduce the cost of the testing effectively. The most straightforward method of regression test selection is simply to select all of the original test cases. However, this can be excessively costly for any thorough test case suite which must test large-scale systems. Regression test selection techniques attempt to reduce the cost of regression testing by selecting T', a subset of T, and using T' to test P'. If the original testing suite is more expensive to run than the reduced testing suite, along with the cost of performing the regression test selection technique deployed, then measurable cost reduction has been achieved by performing the regression test selection technique. This equation is vital to determining whether or not performing the actual regression test selection mechanism was beneficial and will be calculated often in the empirical study sections.

A safe regression test selection technique is a regression test selection technique which guarantees that no modification-revealing, and thus potentially fault-revealing, test case is left unselected when the technique is finished. This establishes the same level of confidence that removing no test cases would. It does not guarantee that if there is a fault the tests will catch it; it simply guarantees that if a modification would have been caught by running all the test cases, it will be caught by running the test cases selected during a safe technique. Lastly, a regression test selection technique can only be safe if applied using controlled regression testing. Controlled regression testing implies that when one tests a modified version of a program, all factors which could influence the output of the modified version, except for the code, are kept constant.

2.4 Survey of Regression Test Selection Approaches

There are many regression test selection techniques available which were developed for traditional monolithic software and in this section some of them will be discussed. This particular work is focused only on safe regression test selection, and therefore this section will only cover those regression test selection mechanisms which are safe. In [2], the authors present a set of categories with which to compare regression test selection. These categories are: inclusiveness, precision, efficiency, and generality. Inclusiveness measures the extent to which a technique chooses tests that will cause the modified program to produce different output that the original output, and therefore could expose faults caused by modifications. This property for the majority of the techniques will be safe, which implies that no modification revealing test case will be left unselected. Precision measures the ability of the technique to avoid choosing tests which will not cause the modified program to produce output different than that of the original. Efficiency measures the computational cost, and thus the practicality of the approach. The following techniques will be discussed: linear equation, firewall, cluster identification, and modified entity.

Using the linear equation regression test selection approach, the mechanism first analyzes the program and generates a set of matrices one each which represent the reachability, connectivity, test case dependency, and variable set/use of the software artifacts [11]. The connectivity matrix models the control flow of the program. The reachability matrix reflects the indirect and direct interconnections between segments (similar to data flow) which can be computed directly from the connectivity matrix. The test-case dependency matrix models the test coverage of the test cases. Finally, the variable set/use matrix reflects the use of variables throughout the segments. Note that the connectivity matrix is nothing more than a control-flow graph and the coverage matrix holds the coverage information. The selected approach is very similar since it uses the same two elements to perform its work. However, the algorithm they use to compute the set of test cases which need to be rerun is much different. The linear equation algorithm is based on a zero-one integer programming model with an objective function, which represents the minimum number of test cases that must be rerun after modifications. Zero-one integer programming is a special case of integer programming, in which all variables are integers, where the variables are required to be either 0 or 1 (binary). This objective function only chooses a minimum number of test cases that must be rerun after modification and may miss a modification-revealing test if other test cases cover all of the paths the test case covers. However, if one were to remove the minimization requirement, the algorithm would be safe [2]. Even though this algorithm can be applied in a safe manner, this technique requires the use of algorithms to solve zero-one integer programming problems which may carry exponential in the worst-case time, and in fact, the underlying problem is NP-hard. Even though there have been some optimizations bringing the cost down to a reasonable time frame [12] (around an hour for some large systems) the cost is still far too high to apply to a system repeatedly.

Although the firewall technique is not completely safe, the technique and the conditions under which it is safe will be discussed. The firewall technique developed by Leung and White [13] determines where to place a firewall around modified code modules. Their technique selects unit tests for modified modules which lie inside the firewall and integration tests for groups of modules which interact with modules which lie inside the firewall. Their technique is safe if and only if the unit and integration tests used to test the system are reliable. Reliable test cases implies that correctness of modules exercised by those tests for the tested inputs implies correctness of those modules for all inputs, which was shown by Leung and White to never be the case in practice.

The next technique to be discussed is cluster identification which was developed by Laski and Szermer [14]. Their technique identifies single-entry, single-exit subgraphs of a control flow graph, called clusters, which have been modified from one version of a program to the next. A control-flow graph is a graph in which each node represents a code entity and each edge represents the flow of control from one node to another. In the next section, control-flow graphs will be discussed in more detail. This approach computes control dependence information for a procedure and its changed version, and then computes the control scope of each decision statement in the procedure by taking the transitive closure of the control dependence relation. This technique uses this information to identify clusters and establish a correspondence between the control-flow graphs (old and new) and selects the tests which cover modified, new, and deleted clusters in the process. The downfall of this approach lies in its low efficiency. The running of time of this algorithm is bounded by the time required to compute the control scope of decision statements, which is $O(n^3)$ for a program with n statements. Additionally, the algorithm for establishing a correspondence between clusters is quadratic in the size of the larger of P and P'. Therefore, even though this algorithm is safe, it is excessively expensive compared to other approaches [2], especially considering that every time the system is modified, these algorithms must be performed.

Chen, Rosenblum, and Vo presented the modified entry technique which is a regression test selection technique which selects modified code entities [15]. Code entities are defined as executable portions of code such as functions, or as non-executable components, such as storage locations. Their technique selects all tests associated with modified entries. The authors implemented the technique as a software tool, called TestTube, which performs regression test selection for programs written in the C programming language. Program entities are kept in a database which facilitates the comparison of those entities to determine where modifications have occurred. This approach is safe, and certainly efficient which at worse case is equivalent to the size of the program (must scan it once) multiplied by the number of tests (must scan the list once). However, since the approach uses a database to perform its work, which allows for some performance enhancements, the efficiency of this approach is the best approach in terms of performance. Even though distributing this approach would be possible, it is unpractical for companies not wishing to share database access with outsiders, especially outsiders whom it may never actually meet, which would be the case with some services.

2.5 Selected Regression Test Selection Approach

Most safe RTS techniques rely on information about the program's source code. The technique which has been adopted by the approach presented in this work involves generating control-flow graphs from the involved code [1]. As mentioned earlier, they are graphs in which each node represents a code entity and each edge represents the flow of control from one node to another. An additional structure needed by this particular algorithm is a mapping of the test cases to the control-flow graphs. Since each test case covers a path through the system it also covers a path through the control-flow graph and the algorithm uses this information to determine which test cases to select and which ones to pass by.

The techniques involving control-flow graphs follow three basic steps, which will be covered in more detail: 1) It constructs a control-flow graph for P'; 2) Identifies dangerous edges by comparing the control-flow graph of P with the control-flow graph of P'; 3) Based on coverage information and the set of dangerous edges it selects from the test suite those tests that need to be rerun. The three steps require an initialization phase as well, since before they can begin, a control-flow graph must be generated for P, along with the supplemental coverage information providing the mapping between paths through the control-flow graph and the test cases which cover them.

As mentioned earlier, a control-flow graph is a graph in which each node represents a code entity and each edge represents the flow of control from one node to another. They are constructed directly from code artifacts. These artifacts can either be source code, byte code, or machine instructions. The entities themselves can be statements, blocks of statements, methods, and operations. The choice of entity determines the granularity of the control-flow graph. The first phase of the regression test selection technique is constructing a control-flow graph for P'. This process is identical to the process of creating the original control-flow graphs. The process of creating control-flow graphs is a well established part of compiler theory since control-flow graphs are used in the optimization process of many compilers.

For example, suppose there is a method with psuedocode presented in figure 2.2, the control-flow graph for this method would look like the one in figure 2.3. Additionally, note the brick and checkerboard nodes which represent the starting node (which denotes the beginning of execution) and end nodes (which represent the end of execution).

```
1 order(item) {
2   if (item exists) {
3     if (item is in stock) {
4        order item;
5        return successful;
      } else {
6        return error("ERROR: 104: item not in stock");
      }
    } else {
7        return error("ERROR: 109: item does not exist");
    } }
```

Figure 2.2: Psuedocode for an ordering service

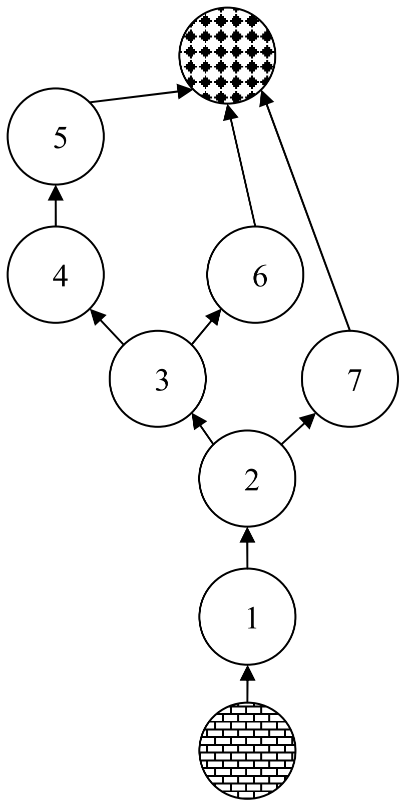


Figure 2.3: Control-flow graph for the psuedocode in Figure 2.2

This particular control-flow graph is at statement level since each of the nodes corresponds to statements in the psuedocode. Although the control-flow graph presented in figure 2.3 does not have any cycles, control-flow graphs are allowed to have them. Cycles are generally caused by either looping or recursion constructs in the programming language. Since control-flow graphs are allowed to have cycles, all of the control-flow graphs in this work can be considered as directed graphs.

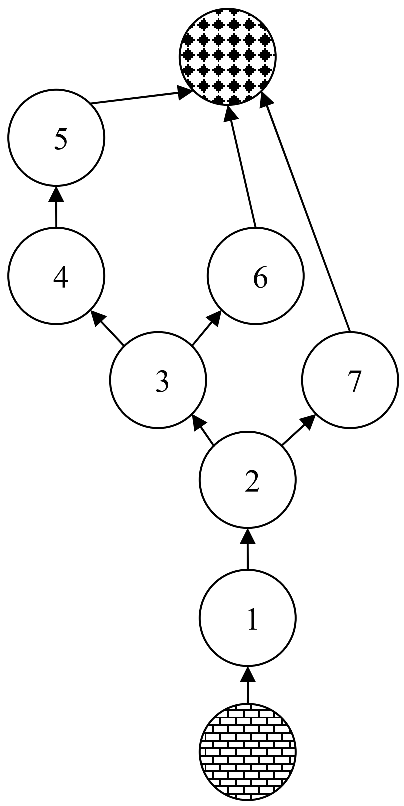
The process of identifying dangerous edges by comparing the control-flow graphs of P and P' is one of the important parts of the process which will be discussed in detail. Dangerous edges correspond to program entities that may behave differently under a single test case due to differences between P and P'. The regression test selection algorithm compares the two control-flow graphs by traversing the two control-flow graphs simultaneously looking for differences between them. If the two nodes are different in terms of their children or their values, the algorithm adds the node to the dangerous edge list. Since it compares the node and its children the algorithm is capable of finding both structural (new branches, i.e. new case clauses) and textual differences (altered lines of code, i.e. changing x < 3 to x < 5). The algorithm is recursive and stops either when it finds a difference in the control-flow graph, when it reaches a node it has already compared, or when it reaches an exit node without finding a difference. The recursive algorithm ensures that any modification in the graph will be found (since the entire graph is traversed) and that if one is found, it is added to the set of dangerous edges.

For example, suppose that service A is represented by the psuedocode and the control-flow graph presented above which was presented in figure 4.2 and 4.3 respectively. Suppose after some time, the developers of service A modify the psuedocode to what is shown in figure 2.4. The differences are shown in italics.

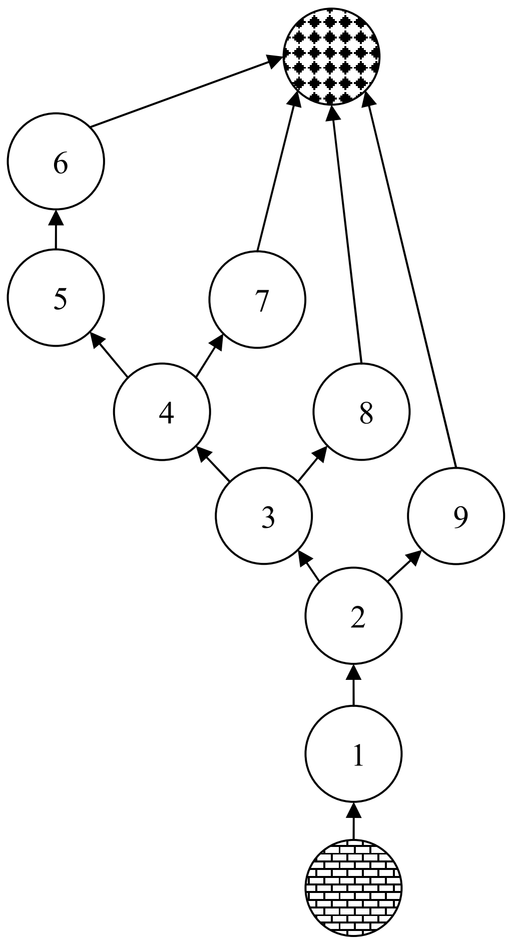
```
1 order(item) {

2   if (item exists) {
3      if (item is in stock) {
4         if (customer has money) {
5            order item;
6            return successful;
        } else {
7            return ERROR: 103: customer lacks funds
        }
     } else {
8         return error("ERROR: 104: item not in stock");
     }
  } else {
9      return error("ERROR: 110: item does not exist");
  } }
```

Figure 2.4: Altered psuedocode for ordering service from Figure 2.2

The regression test selection approach must build the control-flow graph for this new version of the order service and that is shown with the original one in figure 2.5.



(a)



(b)

Figure 2.5: CFGs for the psuedocode in Figure 2.2 (a) and Figure 2.4 (b)

The algorithm which determines the set of dangerous edges compares the two control-flow graphs by performing a dual-traversal as described. The result of the dual-traversal marks the following edges dangerous: 1-2, 2-3, 2-7, and, 3-4. It selects these edges because the node corresponding to four is structurally different than the original and because the node corresponding to seven is textually different.

The last part of the process is very important, since it selects the test cases that will be actually run and completes the process of regression test selection. The algorithm uses the coverage information provided during the initialization step, which maps test cases to the original control-flow graph, along with the set of dangerous edges produced by the second step to select the tests from the test suite which must be rerun. The process of actually performing this step is very straightforward. The coverage information can easily be thought of as a table and the process is simply a table lookup using that coverage information. The technique guarantees that any test case which does not cover a dangerous edge, or entity, will behave exactly the same in both P and P’, and thus can never expose a new fault in P’. Since it is guaranteed to only remove those tests which can never expose new faults in P’, this technique is safe because it minimizes the number of test cases while maintaining the same level of confidence provided by the selecting all test cases.

For example, suppose that the original service A was augmented with test cases and coverage information which are both shown in figure 2.6. Note that since the code shown is psuedocode, the test cases will follow suit.

```
Test Cases

   Inputs corresponding to three test cases
                   1. Order item which does not exist
   2. Order item which does exist but is not in stock
   3. Order item which does exist and is in stock

   Expected outputs corresponding to the three test cases
   1. return error
   2. return error
   3. return successful

Coverage Information

   1. 1-2-7
   2. 1-2-3-6
   3. 1-2-3-4-5
```

Figure 2.6: Three test cases and their coverage information for service A

Suppose that the modification described in figures 2.4 and 2.5 occur and the following edges are marked dangerous: 1-2, 2-3, 2-7, and, 3-4. The coverage table is used as lookup table and tests numbered one and three are selected for retesting. These tests are selected because the dangerous edge list prefixes these two tests completely.

Another method is an extension of the selected approach by the one of the same authors, Mary Jean Harrold, and her research group which must be discussed in the context of this work [16]. This technique was developed to apply regression test selection to Java software. They develop a graph structure, an extension to control-flow graphs, which handles the features of the Java programming language, such as exception handling, non-complete programs (code which uses libraries), and polymorphism. The most important point of this work is that in their analysis they ignore libraries which do not change. Programmers often make use of these libraries, without access to the underlying code. However, since these libraries never change from release to release, they can be safely omitted from analysis.

Chapter 3: Survey of Related Literature

In this chapter, a survey of related works will be presented with special emphasis on how these works are related and how this work is differs from theirs. In terms of service-oriented architecture, most of the existing approaches focus either on test case generation or test execution. They ignore the potential cost reduction of regression test selection. There is also some research which delves into the use of other representative models to perform regression test selection, as well as automated frameworks for regression testing. Lastly, there have been some researchers working to perform regression test selection on Web based systems. Each of these items will be presented, along with a discussion of the techniques which have been proposed for regression test selection for Web services.

3.1 Test Case Generation and Automated Test Execution

Test case generation is related to this work because in order to provide some level of confidence that the system is fault-free the system must be tested using test cases. Offutt and Wu [17] designed an approach to generate test cases for Web services using data perturbation. Their proposal works at the messaging layer of the Web services protocol stack, SOAP. Existing XML messages are modified based on a set of rules defined on a set of message grammars, and then used as test cases. Their proposed approach uses both data value perturbation, which works as described, and interaction perturbation, which classifies communication messages into two broad categories: RPC communication and data communication. The data communication category is used for those messages which would normally carry a payload of some type and RPC would normally simply be any other type of communication, such as requests and other control messages. Their approach can be considered a "black box" approach because it only perturbs the SOAP messages based on their modifications and make no use of internal implementation artifacts.

Another approach aimed at test case generation for Web services was developed by Siblini and Mansour [18]. Their approach uses mutation analysis, which is a fault-based testing method which measures the adequacy of a set of externally created test cases [19]. It works by inducing faults into software by creating many versions, called mutants, of the software each containing a fault. Mutants are limited to simple modifications to the original program on the basis of a coupling effect, which simply implies that there is a relationship between complex faults and simple faults in that a test data set that detects all simple faults will detect most complex faults. After creating the mutants, they are executed, and killed if necessary. A mutant is killed when the result is different than that of the original program. After executing all mutants, the test is left with two pieces of information: the number of dead mutants and the number of still living mutants. The still living mutants are compared to pre-existing test cases to determine equivalence based on input to produce a number of equivalent mutants. Along with the count of all mutants, a final mutant score is computed. This mutant score is the kill ratio, or number of dead mutants divided by the number of still living mutants, ignoring the equivalent mutants. However, mutation analysis can not be directly applied to Web services since the tester may not have all the requisite information, such as code, specification, etc. Their approach uses mutation analysis which is based on applying a set of mutation operators to a given WSDL document in order to generate mutated Web service interfaces that will be used to test the given Web service. Their set of mutant operators is specific to WSDL documents. Finally, they presented a set of empirical analysis to show the usefulness of their approach. The two approaches to generating test cases are very interesting but are only relevant in the sense that test cases are required to perform regression test selection, and thus are only marginally related.

Researchers at North Carolina State University have developed a framework which automatically generates test cases and executes them [20]. Their approach relies on using a given WSDL document to generate a client for the given service, then leverage existing automated test case generation tools, and finally execute those test cases. Their approach is focused on unit testing, specifically the JUnit framework. The test cases are executed using the JUnit framework, which is a regression testing framework. However, JUnit itself provides no facility for determining when or how the system was modified. It simply executes the test cases when the developer tells it to execute them. The automation they have achieved is through the one time execution of the tests post-generation. This approach is related in that it focuses on automating the generation and execution of test cases for Web services, however, the approach is focused on unit testing and not at the system level, much less end-to-end testing. Additionally, the approach has no mechanism in place for the automation of determining when the tester needs to run.

Another approach, developed by Fu et al, is focused on testing the error recovery code of Web services using white-box def-use testing [21]. Some error recovery code may handle situations which occur with a very small frequency due to interactions with the computing environment and these situations cannot be tested simply by manipulating program inputs. Their analysis techniques identify program points which are vulnerable to certain faults and the corresponding error recovery code for these specific system faults. Their techniques allow compiler-inserted instrumentation to inject appropriate faults as necessary and to gather recovery code coverage information, which enables a tester to systematically exercise the error recovery code. In their approach it is important to be able to precisely locate where an exception was thrown in response to an experienced fault, which is termed a def, and where that exception was handled, which is termed a use. A key concern of their use of def-use testing is to minimize the number of spurious def-uses reported in the analysis. They use exception-catch link analysis, which is performed at compile-time, to minimize them. This analysis is essentially an interprocedural def-use dataflow analysis calculation with two refinements: 1) it inlines constructer code, and uses the absence of data reachability through object references to confirm the infeasibility of links. Lastly, their approach automates the program instrumentation directed by the analysis. However, the services they discuss are simply Web applications which follow a client-server model, and they are focused solely on the server, ignoring the interaction between the two. Additionally, their approach also is not language neutral since it only works for servers built in Java.

3.2 Other Mediums to Support the Interoperability of Web Services

The use of a medium to handle the issues that arise as the result of the interoperability of Web services is a common theme. The mediums which have been discussed are interoperable containers for sharing test cases. However, there are approaches which use other means to determine what has changed in a system using other models, such as UML (the Unified Modeling Language). While not many of these approaches have been applied to Web services, they merit mentioning because of the application of the medium to handle the issues related to interoperability.

Bernard Stepien at the University of Ottawa developed an interesting approach to automating the testing of Web services [22]. Their approach focuses on using the Testing and Test Control Notation (TTCN-3) [23] and mapping the XML data descriptions in SOAP and WSDL to TTCN-3. TTCN-3 is the third version of a programming language developed specifically for testing which is used to define test procedures to be used for black-box testing of distributed systems. TTCN-3 is also used as a test specification language since the system is capable of specifying test input as well as how the test should be executed. Stepien provided a means to translate SOAP messages into a format that TTCN-3 can understand and utilize to generate and execute test cases. A later work builds on this by noting that TTCN-3 forms an abstract test suite (ATS) which is language and platform neutral [24]. More importantly in the second work, the authors describe mechanisms for distributing the testing. On the server side, the testers would generate the ATS and publish it. On the client side, the testers would use the generated ATS from the server side to test the service. There is no mechanism for handling the composition aspect of Web services and no mechanism for composing the test suites so that test can be end-to-end. However, this approach is very interesting because it uses a specification language of TTCN-3 as a medium to handle the interoperability aspects of Web services. Additionally, their approach publishes test related meta-data for use in testing. Some of their colleagues worked on a very similar approach using an arbiter to manage the testing [25]. This works identifies an architecture which is almost identical the one just presented with the additional of the arbiter to manage the ATS. This particular work is interesting only because it attempts to map the testing concepts in UML 2.0 to the TTCN-3 language in a standard way.

Another approach aiming to automate testing Web services which uses a medium for handling the interoperability concerns of Web services was developed by Dustdar and Haslinger [26]. Their approach uses a meta-language in XML to define test cases for services. Their framework, which is named Service Integration Test Tool (SITT), was developed to handle test case execution and monitoring to ensure correct behavior. The test execution for their approach is distributed, with one test agent per service running on the same machine as the service. These agents run the actual tests and log incoming and outgoing messages of every service in the system. They send the important items in the log to a master which parses the log and performs the testing. This master determines if the logged inputs match to correct outputs for each service endpoint in an interaction. This particular study is very interesting and related in that it handles composition using a medium for sharing test cases and that it uses a distributed scheme for testing even if the scheme uses a centralized controller to determine the results.

Yet another approach aimed at the execution of Web services testing is the work by the researchers at RCOST (Research Centre on Software Technology) at the University of Sannio [27]. This particular work aims to use test cases as a contract to ensure service compliance across releases. Specifically, they wish to annotate each service with test cases in a standard way, using an XML based medium, and provide this to the users of the service so they can test the service. The standardized test cases are called testing facets and are a mixture of JUnit tests and tests generated from static and dynamic analysis of the source code. The testing facets are published along with the service so that users of the service can perform the testing of the service. Their work also presents a set of perspectives of who might wish to use such a framework and those are: providers, users, certifiers. This work is highly related to this because they share test cases using XML, but there is no attempt to perform regression test selection which is the central focus of this work.

Another of these approaches was developed by Heckel and Lohmann at the University of Paderborn [28]. The testing in this paper is focused on developing a means to test Web services by using Design of contract to add behavioral information to the specification of a Web service. They use graph transformation rules to describe the contracts at the level of UML models which allow for the simulation of the required service by accomplishing the contracts during the execution of a test case. It translates UML class diagrams into contract graphs which are represented by UML object diagrams. These contracts graphs are visual representations of the details required to correctly identify service assertions. For instance if an ordering service took only credit cards, the ordering service would be only be connected to that data type. Once the contracts are generated, a test case generation tool, specifically JUnit, is run to create the test cases and tester. This approach, as mentioned before, is important and related because the medium it uses to solve the issues related to interoperability, namely UML, is being used not to share, but as a major part of the approach.

Another approach related to the use of UML was developed by researchers at the Software Quality Engineering Laboratory at Carleton University [29]. The approach focuses on using the class and sequence diagrams in UML to perform impact analysis.

Impact analysis is highly related to regression testing and regression test selection in that they both focus on the how the modification changes the system. The difference between the two is that while regression testing requires a modified system to determine the difference between the old and new, impact analysis does not. Impact analysis is focused on determining the impact before the modification occurs. However, in this particular work, the two terms are identical. The authors first use impact analysis to determine what and how the system has changed and then they use this information to determine (select) which test cases need to be rerun. The test cases are associated with a sequence of triplets consisting of: Method signature, Source class, and Target Class. This triplet is the path through the diagrams that the test would take. In a later work [30], they describe a set of rules to check the consistency of the UML models post modifications to ensure preconditions of impact analysis. Additionally, they prioritize the results of the impact analysis using a distance measure. These studies are related to this work in that a representative model of the system under test is being used directly to support test selection. However, their approach is only capable of handling modifications at the design level since it is a black-box based approach. Finally, these frameworks use program slicing to determine which test cases to run. Program slicing tries to eliminate all parts from the program that are not currently of interest to the programmer. More specifically, dependency graphs are built using program entities as nodes and the edges follow a dependent relationship. The program entities used to develop dependency graphs in their frameworks are specification based entities such as UML design elements. Once a dependency graph is built, the slicing technique slices the graph to only include those nodes which are dependent on the modified node [31]. This approach to regression test selection is unsafe [2] since it misses test cases in situations in which code was inserted into the program, since there were no dependency relations on the code in the previous iteration.

Another related work was performed by researchers at Siemens Corporate Research which uses UML statecharts as a means to generate and execute test cases [32]. Their work entails the use of statecharts which are generated by the developers of the system under test. These statecharts specify the interactions amongst the components interacting in the system and must be annotated with test requirements. Their approach uses these annotated statecharts to generate test cases using a tool designed to perform this. The resulting test cases are then executed using another tool designed for this purpose. The tools were designed separately so that a user could manually create additionally test cases and execute them using the second tool, or a user could generate the test cases and execute the test cases using a different tool. Their approach hinges on the ability of their statecharts, which are very much like control-flow graphs, to be composable. They argue, just like this work does, that since statecharts are nothing more than special cases of finite state machines they can be composed just as finite state machines can. This work is especially related since it uses a composable medium to perform its work, which is test case generation and test case execution and not test case selection. Additionally, this approach is predicated on using user defined statecharts which may not actually represent the system at hand, which would call into question the reliability of this approach. Even though these techniques and tools handle the interoperability and composability of Web services using interchangeable mediums, none of them could ever be safe since they are all specification-based.

3.3 Regression Test Selection Techniques for Web-based Systems

At the time of writing, there are very few techniques available for performing regression test selection on Web services. The largest and most related body of work on performing regression test selection was developed by researchers at the Arizona State University, Wei-Tek Tsai, and the Department of Defense, Raymond Paul. They reported a framework which requires the use of enhanced WSDL specifications [33]. In another work, they describe specifically what enhancements are required [34]. They describe testing Web services as being equivalent to black box testing since only interfaces are known, and the specifications for the interfaces are written in WSDL. They argue that in order to support black box testing more information that what is provided in WSDL is necessary. This additional information is fourfold: input/output dependency, invocation sequences, hierarchal functional description, and concurrent sequence specifications. Their framework uses the enhanced WSDL documents to develop what the authors call scenarios [10]. A scenario describes a function from an end-users point of view [35] and can be thought of as thin-threads that trace a path through the system under test starting with the user, much like a use-case would. Scenarios in their view are directly related to use-cases, but with detailed design information incorporated. In their work, scenarios can be generated in a variety of ways, namely user-generated, generated from enhanced WSDL documents, or generated from design elements such as UML diagrams. The information which accompanies a scenario in their systems is: 1) an ID, 2) a name, 3) a context, which relates the circumstances under which the scenario is used, 4) reference, link to artifact the scenario was developed from (could be code, the service, WSDL, etc),

5) Inputs, the input required to trigger the scenario, 6) outputs, the data or product of the execution of the scenario, 7) precedents, scenarios which should be executed before this one, 8) successors, scenarios which should be executed after this one, and 9) a description of the scenario. It is important to differentiate between a scenario and a control-flow graph. A scenario is an artifact which is generated to support black box testing which does not require insight into the development of the implementation of the system components. The control-flow graphs that will be used throughout this work do require information regarding the inner workings of the system it is testing, and thus is a white-box approach. However, their work is related in that their approach as well as this work is interesting in verifying the system under test from the point of view of the end user. These scenarios can also be used to generate test cases by using the input/output fields of the scenario [35].

The framework they propose in [36] specifically looks at using the scenarios to run the test cases using two different approaches including: a centralized tester and a test master with distributed testers. They argue that the distributed testers must collaborate while sharing information during the process. They mention that several synchronization schemes are available to coordinate the testers, but the architecture requires a master for logging and other shared facilities so the schemes must all use the master coordinate the testers. This is related to the approach presented in this work as well since the testers in this work will also be distributed and missing a master, thus use decentralized algorithms to handle concurrency.

In another work from their group, they present an approach to support functional regression testing, and selection, based on scenarios [37]. Their approach uses what the authors call scenario-slicing which is based on program slicing. Their scenario-slicing technique uses the input and output, along with the dependencies which are in the scenarios to perform a kind of program-slicing. Specifically, if any field of the scenario contains a given attribute which matches one described by the modification, that scenario is selected for testing. This is not a safe regression test selection technique, just as program slicing is unsafe [2], for the same reasons. Suppose one of the services in the system under test was modified to include a new artifact, the algorithm would not select these because there is no dependency information on that artifact which did not exist during the creation of the scenarios.

They build on this framework in [38] providing an object-oriented framework to perform end-to-end testing on systems of systems, which performs scenario-based regression test selection, scenario-based test case generation, and automates the testing using a centrally controlled distributed group of testers. The framework presented in [39] adds to this monitoring capabilities and change management. Their test monitors monitor the messages being exchanged throughout the system and keep track of system state using the information. This state is sent to the test master and is used to determine if the interaction, or behavior, of the services was correct. Change management is performed using an enhanced UDDI server [40]. Their approach enhances a UDDI by adding check-in and check-out capabilities. The UDDI server tests services upon check-in and clients are provided with test cases upon check-out. The clients can then test the services before using them if they need to. The client is only directed to the test cases when the service has been checked in and thus modified. This is how the UDDI server handles change management for the system. Important to note, that the user of the service is not directed to retest the service until the user attempts to look up the server again.

Another stem of their work involves test case generation based on scenarios which include both positive and negative test cases [41]. This work involves all of the other works presented thus far by their group and specifically focuses on adding the negative testing capability to their frameworks. Another work aims to improve the completeness of the specification of their scenarios using what they term consistency and completeness criteria [42]. They try to identify coverage gaps in their scenarios using min-terms of Boolean expressions that combine multiple conditions into a single checkable item. The more important artifact of their approach is that post generation they use OWL-S to share their test cases. OWL-S supplies Web service providers with a core set of markup language constructs for describing the properties and capabilities of their Web services in unambiguous, computer-interpretable form. OWL-S markup of Web services will facilitate the automation of Web service tasks including automated Web service discovery, execution, interoperation, composition and execution monitoring [43]. Yet another work by the researchers at Arizona State provides an in-depth discussion of collaborative verification and validation (CV&V) contrasted with independent verification and validation (IV&V). This collaborative view of testing allows for voting algorithms and group testing which aims to rank services in order to select the best service on functional criteria. Lastly, the researchers at Arizona State present a framework which takes an OWL-S specification of a Web service interaction, converts it into C, uses the BLAST toolkit to then transform the C code into a control-flow graph, and finally generates test cases for the Web services using them [44]. This is related only in that it uses control-flow graphs to generate test cases. The OWL-S specification supports only Web services as the level of granularity and thus is a black-box approach, even though it uses a white-box method to get there. Although there are a number of similarities between their approaches and the approach outlined in this work, there are also a number of fundamental differences: 1) the proposed approach is safe, 2) requires no extensions to Web service standards, and 3) requires no modifications to the Web services themselves.

Another related work developed by the researchers at North Carolina State University and ABB Inc. in which the researchers report a regression test selection technique which works even when source code is unavailable [45]. Their work aims to perform regression test selection on commercial off the shelf (COTS) component software for which the tester has access to glue code, the binaries of the software, and the test suite for the glue code. The first step of their process is to decompose the binary files of the components into code sections and exported functions using a set of binary parsing tools. The next step is to compare the code sections between the two versions using standard differencing tools. The ultimate goal of this is to feed the change information into code-based regression test selection mechanisms. The regression test selection algorithm they use is the firewall method developed by Leung and White, which as mentioned in the background is unsafe. More importantly, the approach presented in this work could never be applied to Web services for which neither source nor binary code are available.

Another approach to apply regression test selection to Web services was presented by Tarhini, Fouchal, and Mansour [46]. They present a way to generate test cases based on three elements: 1) WSDL files, 2) the specification of the component services, and 3) the specification of the system as a whole. The first set of test cases is generated using the WSDL files using boundary analysis. The second set is generated using a notion of the TLTS specifications. TLTS, or time labeled transition system, is a graph where each node is an abstract representation of a single component that models the behavior of a component. The edges represent flow of actions. Additionally, these edges are annotated with timing constraints. Essentially, their model can be though of as a UML statechart annotated with timing constraints, which is very similar to the control-flow-based approach presented by this work. They even present a technique to compose the TLTS automatically. The third set of test cases is generated using the global TLTS. Once generated all test cases are stored in a central log file which stores the URLs of all services and their test cases. The test history is updated as well. Their regression test selection algorithm given a new TLTS, creates a global TLTS based on the TLT'S given. It then generates a new set of test cases given this new global TLTS, T’. The newly created set of test cases is compared to the original set of test cases, T, and all those test cases found in T’ but not in T are executed.. All test cases found in T but not in T’ are deleted from the log, and are thus removed. Finally, all test cases found in both T and T’ are kept but not executed. Effectively, they are performing the path analysis regression test selection technique, which performs precisely as they describe [47]. This technique misses test cases for which the path has been removed or the path has been newly added. However, their use of the log solves these two problems because the test cases would not be generated for the missing path and would be generated for the newly added path. However, the path analysis problem is part of their solution and is computationally expensive. The technique, without generating a single test case, carries an exponential worst-case running time. This approach is excessively costly considering that there may be thousands of test cases generated every time the services are modified in addition to the cost of performing the path analysis.

Another related work by Harrold et al, is a mechanism to perform regression test selection on component-based software [48]. Their approach, which is very similar to the approach presented here, revolves around publishing and using metadata. Their version of metadata is provided by vendors in lieu of source code in such cases where that content may be restricted by patent or copyright law. Their metadata is threefold: edge coverage achieved by the test suite with respect to the component, component version, and a mechanism to query each component for the edges affected by the modifications between any two versions. The coverage information is obtained using the following procedure: 1) The component is first told to turn on instrumentation facilities by the application, 2) for each test case you want coverage information on run it and gather coverage information for the component, and 3) turn off instrumentation. A very important point to note here is that there is no way to determine that the component is being fully tested. That is to say, the component only responds to the test cases it is presented with and therefore, the tester has no way of knowing how well they are covering the component (may miss modification revealing tests if they fail to provide adequate coverage). However, this does not affect the safety of the technique since the only thing safety requires is that all tests which could be modification revealing are selected. In the approach presented here, the Web service needs only to publish the coverage information for a set of test cases developed for their purposes and never need to provide coverage information for arbitrary test cases. Additionally, the component version in their work is used to identify if a modification has occurred since the last test. However, there is no facility presented which informs interested parties automatically upon modification. Additionally, the authors do not propose a mechanism to support a component which calls another component which in turn calls yet another component.

A second facet of their approach is based on specification-based regression test selection. In this approach, the developers of the component provide a statechart as metadata, with no test cases or coverage information. The testers would need to generate this information using the provided statechart which is composed into the statechart of the application using the component. For a provider to only provide statecharts as metadata for a service, the statecharts must have decision information which can be used to determine what makes one execution go down one path and another go down another. The approach presented by this work prevents this kind of information sharing which would not be tolerated by a vendor which would not wish to share how his service works.

Table 3.1 will further clarify the differences between the proposed technique and the related techniques by describing the features of the proposed approaches in comparison to the listed approaches. Note that the approach from the NCSU is not listed since it cannot be used for systems such as Web services in which the binary code itself is not made available.

Regression test selection approaches	Decentralized	End-To-End	Safe	Automated
Scenario-based Approach	N	Y	N	N
TLTS Approach	N	Y	Y	N
Georgia Tech Approach	Y	N	Y	N
Proposed Approach	**Y**	**Y**	**Y**	**Y**

Table 3.1: Regression test selection approaches comparison

Note that none of the works presented involve distributed regression test selection and testing. More specifically, none of the automated testing approaches are distributed approaches without a centralized controller, and none of the test selection approaches are automated. An automated test selection technique would determine automatically that a modification has occurred, and inform all the necessary parties to perform regression test selection and regression testing if necessary.

Chapter 4: Safe RTS Approach for Web Services

In this chapter, the safe regression test selection technique for Web services will be discussed in detail.

4.1 Introduction

The foundation of the approach is the control-flow based approach proposed by Rothermel and Harrold for traditional monolithic applications [7]. Their approach was chosen since control-flow graphs are ideal for use in Web service environments for a number of reasons. First, control-flow graphs can be generated from programs written in any language, or extracted from designs at any granularity. Thus, they can be used as a common representation mechanism among Web services which could be written in any language on any platform. Second, since control-flow graphs are special cases of finite-state machines, they can be composed into global finite state machines [18]. These two characteristics of control-flow graphs are essential for supporting both the interoperability and composition of Web services.

In a Web services environment, each service is autonomous and can be thought of as its own development island because each service is developed independently of other services. However, any service may interact with any number of other services at any time to perform more complex business functions. To carry out safe regression test selection for composite services there needs to be some information sharing. This information is called meta-data since it is information describing the system in operation and can be shared in a standard way among all services. Specifically, each service will share its terminal (complete) control-flow graph, a set of test cases, and a mapping of the test cases to the control-flow graph. All of these items can be encoded in XML and shared with each other using standard Web interfaces, such as using simple standardized Web services or using the WS-MetadataExchange framework [19]. The WS-MetadataExchange framework allows users of a Web service to query the Web service for meta-data the developers of the service wish to publish. These standard Web interfaces are called meta-methods since they are used to handle meta-data and there will be one for each of the three items of interest: a control-flow graph meta-method, a test case meta-method, and a test coverage meta-method.

The approach involves two phases of operation which must be discussed: the initialization and critical phases. In the initialization phase, all the required artifacts are initially generated and stored in their appropriate places for use later by the approach. This is a complicated process which cannot always be performed automatically, and will be discussed before the critical phase in terms of the three requisite artifacts. The critical, or operating, phase happens when a modification in the system occurs and the testing harness is awoken to test the modification. The majority of the events that take place during this phase are self-explanatory. This phase will be discussed last.

4.2 Initialization Phase

As mentioned in the previous section, the three pieces of meta-data which need to be shared are as follows: terminal control-flow graph for the site, a set of test cases for the site, and coverage information which maps the test cases to the paths they cover in the control-flow graphs. This phase does require human intervention which will be described in detail.

To simplify the construction, there are two cases which will be considered separately: a simple operation which is an operation which never calls another operation to perform its work, and a composite operation which calls other operations (either simple or composite) to perform its work. The construction of the three elements for the simple operation will be discussed first, followed by the construction of the three elements for composite operations.

4.2.1 Simple Operation Initialization

The construction of the three elements (control-flow graphs, test cases, and coverage information) for simple operations follows the same pattern as traditional monolithic applications. Control-flow graphs can be constructed directly since the responsible party has all the necessary code artifacts for the simple operation. The code artifacts can be specifications, such as communication diagrams, or even lower level artifacts such as source code, byte code, or machine code. Additionally, the control-flow graphs can be generated in a variety of granularities, such as statement level (in which the statements form the nodes), block level (in which blocks of code form the nodes), method level (in which methods form the nodes), and finally, operation level (in which the entire operation is represented by a single node). Design level is an additional level, which is used when the control-flow graph is generated from specification. There may be many different reasons for the selection of each individual level, including performance, level of confidence, and security (a developer who wishes to keep some code information private). The nodes are connected, regardless of granularity, using a relationship based on execution order.

All nodes are given a node identifier to uniquely identify each node in the control-flow graph. These identifiers are used in the test selection process when identifying dangerous edges and when selecting test cases based on identified dangerous edges. The values of each of the nodes are a string representation of the code which represents the node at the nodes level of granularity. The string representation is simply a one-way hash of the code itself. This makes the tester capable of determining whether a specific region of code has been modified without requiring that the region of code be revealed to the tester. In summary, each and every node in a control-flow graph carries two pieces of information: 1) An identifier which uniquely identifies each node in the control-flow graph and 2) a value which is a one-way hash representation of the code itself. Since the mechanism to generate control-flow graphs is well-known (control-flow graphs are used in several steps in the compiler optimization process of most compilers), it will not be discussed in further detail.

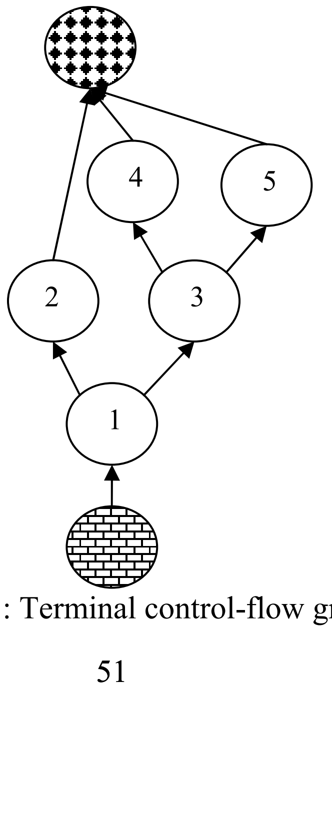


Figure 4.1: Terminal control-flow graph of B

For example, assume that we have a simple operation B which performs some work. Since all the requisite artifacts to generate a complete control-flow graph for B will be present at the site, the control-flow graph can be generated directly. The control-flow graph for this operation is presented in figure 4.1.

Test cases can be generated in a wide variety of mechanisms using a variety of artifacts. Test cases can be generated using specifications, or using code-based artifacts. Test cases are the set of inputs to the operation along with the expected result of calling the operation with the given inputs. The mechanisms used to generate test cases for simple operations are identical to the mechanisms in place for traditional monolithic applications, which are all well established mechanisms, and thus will not be discussed any further.

The coverage information which must be generated along with the test cases and the control-flow graphs is attained by instrumenting the code so that when the test cases are run over the system, the path through the system is recorded. Therefore, the coverage information is a lookup table consisting of test case identifiers along with the paths through the system the test case covers. Again, since simple operations have all the artifacts, the mechanism to handle simple operations is identical to the mechanism for traditional monolithic applications. The mechanism which provides the coverage information is well-established, and therefore will not be discussed in any more detail.

4.2.2 Composite Operation Initialization

The approaches to the generation of the three required pieces of meta-data for simple operations are identical to those for traditional applications, but none of these are directly applicable to composite operations since all of the necessary artifacts will not be available at generation time. In fact, considering that operations are developed by different groups of people and can be developed in different languages, there may be cases in which the necessary pieces may be unavailable or unusable. The construction of each of the three required elements for composite operations will be discussed in detail.

The construction of each of the elements requires two assumptions: 1) all call graphs for all composite operations are acyclic, and 2) the underlying WSDL for each operation is monitored and handled separately. Suppose we have a system S, with a call graph shown in figure 4.2. This diagram is being called a "call graph" simply because it describes how operations call one another throughout the system of systems.

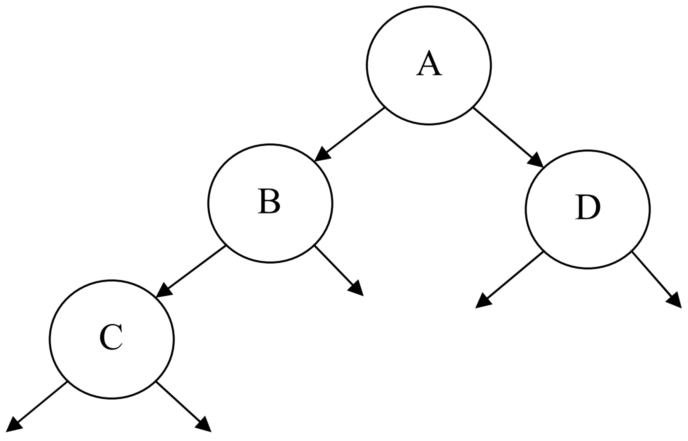


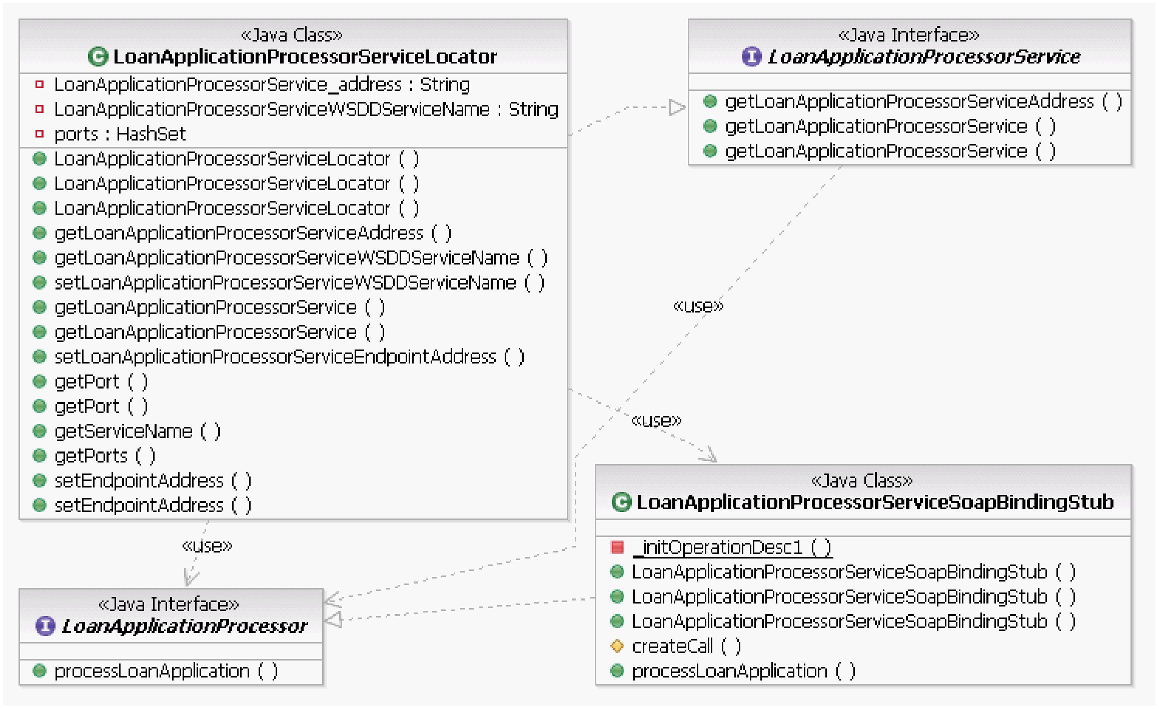
Figure 4.2: A call graph of a system

The nodes in this diagram are operations and the edges form a "Can call" relationship meaning that during the operation to perform the operation the operation may call other operations to perform its work. For instance, operation A can call either B, D, both of them, or even none of them during any given execution depending on the logic present in A, but the graph connects A to B and D because A may call them during its operation. The first assumption implies that recursive calls and looping are not allowed in any call graph, thus every call graph can only form directed acyclic graphs. An additional point is that each and every operation only is aware of the operations it calls directly so the notion of a call graph is outside of the scope of control of each operation and must be controlled via policy decisions externally. WSDL can and should be monitored and handled separately since WSDL modifications are analogous to interface modifications which are outside of the scope of any regression testing system.

An additional assumption is that the called operations (the operations the composite services call directly) must have already finished producing the three elements they are responsible for which are necessary for the composite service to produce its three elements. After the three elements of every simple operation are generated, the three elements of the composite operations that only call simple operations will be generated. As more operations complete the generation of their three elements, the more operations are ready to be generate their three elements. Eventually, every operation will have completed generating their three elements. Thus this assumption does not impose any limit to the approach but can simplify our discussions in the next subsection.

4.2.2.1 Composite Control-flow Graph Construction Initially, since composite operations do not yet possess all of the required artifacts to produce a complete control-flow graph, an intermediary graph must be used. The intermediary graph is called a non-terminal control-flow graph and the final, complete control-flow graph is termed a terminal control-flow graph. The difference between the two is that terminal control-flow graphs have no "call" nodes. Call nodes are the nodes in a non-terminal control-flow graph which correspond to the location in the code which calls another operation. Call nodes will be discussed in greater detail later in this section.

Similar to the construction of terminal control-flow graphs for simple operations, the code for the composite operations must be analyzed to determine which operations the operation is calling. In other words, a control-flow graph is generated, but everywhere the operation would call another operation "call nodes" are placed. "Call nodes" are special nodes which contain information regarding the call, specifically the URI of the service and the name of the operation being called. This information is obtained by analyzing the source code and recognizing where service operation calls take place using prior knowledge about how the Web service toolkit builds its "glue code", which handles the message packing, unpacking, transportation, security, and auditing,.


«Java Class»
LoanApplicationProcessorServiceLocator
LoanApplicationProcessorService_address : String
LoanApplicationProcessorServiceWSDDServiceName : String
ports : HashSet
LoanApplicationProcessorServiceLocator ( )
LoanApplicationProcessorServiceLocator ( )
LoanApplicationProcessorServiceLocator ( )
getLoanApplicationProcessorServiceAddress ( )
getLoanApplicationProcessorServiceWSDDServiceName ( )
setLoanApplicationProcessorServiceWSDDServiceName ( )
getLoanApplicationProcessorService ( )
getLoanApplicationProcessorService ( )
setLoanApplicationProcessorServiceEndpointAddress ( )
getPort ( )
getPort ( )
getServiceName ( )
getPorts ( )
setEndpointAddress ( )
setEndpointAddress ( )
«Java Interface»
LoanApplicationProcessorService
getLoanApplicationProcessorServiceAddress ( )
getLoanApplicationProcessorService ( )
getLoanApplicationProcessorService ( )
«use»
«use»
«use»
«Java Class»
LoanApplicationProcessorServiceSoapBindingStub
_initOperationDesc1 ( )
LoanApplicationProcessorServiceSoapBindingStub ( )
LoanApplicationProcessorServiceSoapBindingStub ( )
LoanApplicationProcessorServiceSoapBindingStub ( )
createCall ( )
processLoanApplication ( )
«Java Interface»
LoanApplicationProcessor
processLoanApplication ( )
Figure 4.3: Class diagram of "glue code" generated using Apache Axis

For example, when using the Apache Axis Web service toolkit, the "glue code" to call an operation of a service is generated using the WSDL document describing the service. The classes it produces are shown in figure 4.3, which is a UML Class Diagram of the "glue code" for a "Business Loan Processor" service. Each of these classes is produced every time the "glue code" is generated. In the client code, the developer uses the "glue code" to call the remote operation as if it were a local Java object as shown in figure 4.4. In order to call the service, the service "locator" object is instantiated and that object is queried for a "stub" object. The "stub" object then acts as a local Java object even though it is carrying out the remote Web service calls. If the framework were to use the information known about how Web services calls take place within the code and how the Apache Axis stores it, the required information to build a call node can be easily attained.

```
lap.LoanApplicationProcessorServiceLocator locator = new
   lap.LoanApplicationProcessorServiceLocator();

lap.LoanApplicationProcessor processor =
   locator.getLoanApplicationProcessorService();

processor.processLoanApplication(loanApp);
```

Figure 4.4: Client code using Axis "glue code"

The "locator" object which is instantiated in the client code always extends the class `org.apache.axis.client.Service`, and it holds the URI of the service in a variable post fixed with "_address". In the example provided above, the `LoanApplicationProcessorServiceLocator` object holds the URI of the service in its local variable: `LoanApplicationProcessorService_address`. This variable is always generated by the Apache Axis toolkit and is always a string holding the URI of the service.

However, scanning for the method name is not as straightforward. After creating a new "locator" class, a "get" method is called on that "locator" object which returns the "stub" object. This object is the object which is responsible for actually transforming a local Java call into a remote Web service call. It holds the names of all operations for the service. In the example provided above, the "stub" object is named: `LoanApplicationProcessorServiceBindingStub`. In order to call the service this object will be directly used. However, the "get" method of the "locator" object returns an interface which has the names of all the operations. In the example, this is the `LoanApplicationProcessor` interface. This interface is scanned and the signature of all of the operations is saved. In the client code, the "stub" class will be called and the name of the method can be retrieved and correlated to the signature of the operation saved earlier. This yields the name of the operation and completes necessary information to create a call node.

Although the Apache Axis toolkit is the most popular toolkit for generating "glue code", it is only one of many which do so. However, which one of the toolkits is used to build the glue code is known to the developers of the service and thus the information about how each of toolkits specifically function can be used to produce the information required for call nodes. Important also is that each of the toolkits perform similar functions and generally produce a standard set of classes and methods. At this point, the call nodes are simply nodes in a non-terminal control-flow graph.

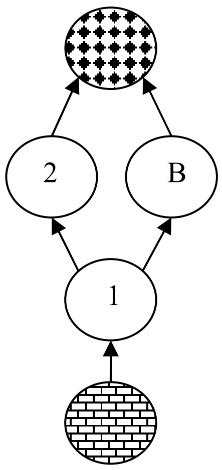


Figure 4.5 Non-terminal control-flow graph for A

As an example, assume there is a new composite operation A which calls operation B, the operation shown in figure 4.1, to perform its work. Since A calls B to perform its work, there is a call node in the non-terminal control-flow graph for A, shown in figure 4.5, which is labeled with the letter B to show it is a call node.

Once the non-terminal control-flow graph is generated, including the information necessary for all of the “call nodes”, this special control-flow graph is saved for future use, since it provides a blueprint of how to compose this operation correctly. It will be used in virtually every step involving the composition of control-flow graphs. The saved non-terminal control-flow graph maintains two other pieces of information to make the rebuilding of the control-flow graphs inexpensive when necessary. This information is acquired when replacing the “call nodes” with terminal control-flow graphs to complete the terminal control-flow graph for this operation. The first piece of additional information is a list of control-flow predecessors to the “call node” and the second is a list of all the control-flow successors to the “call node”. These two pieces of information are required to replace the control-flow graphs of the composed operations at will, while maintaining a low insertion cost. At this point, the non-terminal control-flow graph is complete with regards to the code which is internal to this operation and the operations this operation calls. However, the task is still not complete until a terminal control-flow graph has been generated for this operation.

As mentioned earlier, terminal graphs are control-flow graphs which have no “call nodes”, therefore the ultimate goal of this phase is to replace all of the “call nodes” present with the terminal control-flow graphs each of the “call nodes” represent. This is where the special meta-data operations become critical. The information in each of the “call node” is used to call the control-flow graph meta-method. The control-flow graph meta-method returns the terminal control-flow graph of the service the meta-method responsible for. Although any entity may request a control-flow graph at any time, the control-flow graph meta-method transfers only terminal control-flow graphs because this avoids unnecessary communication. Once the terminal control-flow graph is received, the “call node” is replaced with terminal control-flow graph. It performs this step by using the saved lists of predecessors and successors in the following way: each of the predecessors now has the start node of the terminal control-flow graph as its successor, and each of the end nodes of the terminal control-flow graph now has the successors of the call node as successors.

The nodes may require renumbering due to the uniqueness requirement of the identifiers in the control-flow graphs. Renumbering is performed rather than prefixing the already present identifiers to ensure that the anonymity of the involved operations is preserved. If one were to prefix the identifiers they would be presenting the tester with too much information regarding how the composed system works. Suppose that X calls A which in turn calls both B and C and prefixing is being used. The information about A calling B and C is unnecessarily revealed to the tester at X. To ensure that no unnecessary information is shared, renumbering is performed.

The framework treats the “glue code” as library code and removes it from analysis. It is safe to do so because this code is guaranteed not to be modified unless the underlying WSDL of the operation is modified. As mentioned earlier, having stable WSDL interfaces is a precondition of the regression test selection technique. Thus, the terminal control-flow graph of each operation the composite operation calls will be inserted into the composite non-terminal control-flow graph of the composite operation directly, using only those nodes which actually implement the operation. Once all “call nodes" are replaced with their respective terminal control-flow graph, the result is a terminal control-flow graph for the composite operation which is ready for use in the regression test selection process. At this point, this operation can send its terminal control-flow graph to any entity which had requested it.

As an example, suppose the non-terminal control-flow graph for A shown in figure 4.5 is to be transformed into a terminal control-flow graph for A. In this case, since A calls B, A requests from B its terminal control-flow graph (shown in figure 4.1). Once A receives the control-flow graph from B, A replaces its call node for B with the control-flow graph of B. It would repeat this process for all call nodes to produce a terminal control-flow graph. However, since this particular case only had a single call node after replacing it, the terminal control-flow graph has been produced, which is shown in figure 4.6.

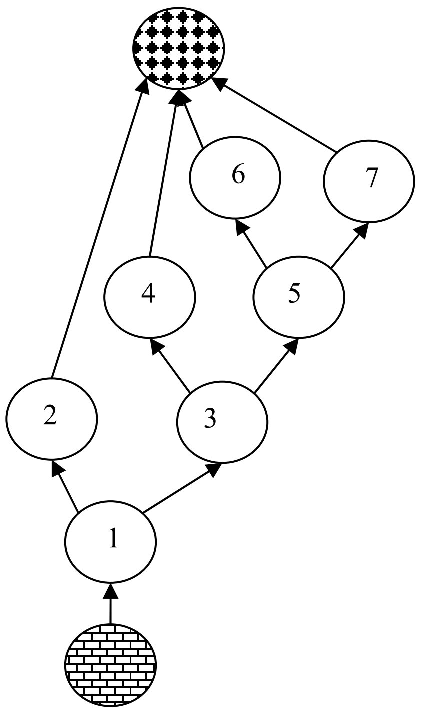


Figure 4.6: Terminal control-flow graph for A

4.2.2.2 Composite Test Case Construction The process of building the test cases requires human intervention in the construction. As mentioned earlier, test cases are inputs to an entity, such as an operation, which produce an expected output. The goal of this step in the process is to generate a set of test cases, along with coverage information for each test case, which exercise the entire composite control-flow graph including inside the composed parts of the control-flow graph (where another operation's control-flow graph will be inserted). Again, meta-methods are employed to assist in the process, and the test case meta-method, which returns the test cases for the operation. However, the test cases returned from the meta-method are for the operation which is called inside the composite operation, and must be made valid at the "call node" of the composite operation. For instance suppose that one of the test cases for the operation being called takes an input of "2". Upon composition of test cases, the tester would have to find an input to the composite operation which results in the called operation being called with a "2". The test cases are uniquely numbered for each operation, thus the test cases which need to be added, are added to the end of the list of test cases generated thus far for this operation, and then renumbered to ensure uniqueness. The input part of these test cases are valid for the entry point of the called operation, not the calling operation, and this information is relayed to the testers responsible for generating test cases for the composite operation. Since each test case represents an input, which must be valid at the point of the "call node", the tester must determine the inverse, what input must be delivered to the composite operation to deliver the input required by the test case at the point of the call node. If this is impossible, one of the two types of manual manipulation is required: test case removal and data augmentation.

Test case removal requires an assertion about the test input in order for the test case to be removed safely. If the human tester can assert that the calling operation can never call the called operation with the same input as the test case because of type then the test case can be removed without missing any test cases that are possibly fault revealing in the calling operation because the calling operation will never make such a call. For example, suppose that a book seller calls Amazon's purchase order service. Even though Amazon provides test cases for purchase books and electronics, the book seller only sells books and therefore never deals with electronics. If this is the case, the book seller would remove the test cases involving electronics and would do so safely due to the strong assertion that the book seller does not sell electronics.

Data augmentation involves informing the calling system of data it was not previously aware of. If the input of the test case is not within the calling operation's available data, but is reasonably similar, the calling operation must add the proper data to its database to be able use the given test case. For example, suppose in the previous example the book seller does not sell the book "Algebra I" and is given a test case by Amazon for selling the book "Algebra I". The database manager could add the title "Algebra I" so that the test case could be directly used.

In both scenarios, a human is directly involved in manipulating the construction of the set of test cases. Once this process is finished for all composed operations, the second required element for regression test selection has been created.

4.2.2.3 Composite Coverage Information Construction The final step in the process is generating a mapping between the test cases and the control-flow graphs. The coverage information is a lookup table consisting of test case identifiers and the paths through the control-flow graph they cover. As the test cases were being renumbered in the previous step, the test case identifiers on the lookup table were updated as well. Finally, the paths are updating by prefixing each of the paths of the composed tables with the path to the composed call node of the composite operation. Once this final step is performed, all the information required is ready for use in the safe regression test selection process.

4.3 Critical Phase

The process of actually performing regression test selection is an adaptation of the three phase process outlined in the background in section 2.5. The first step of the process is building the control-flow graph of the modified operation, the second step is to identify dangerous edges, and the final step is to select the test cases which need to be rerun based on the set of dangerous edges resulting from the second step and the coverage information provided in the initialization phase.

The composition of the control-flow graph for the modified version of the operation is identical to the process outlined earlier in the initialization phase. The operation uses its original “call node” representation of the operation to assemble the terminal control-flow graphs of the called operations to the non-terminal control-flow graph of the composite graph to create a new modified terminal control-flow graph. The control-flow graph for each of the called operations is saved separately during construction and this information is used as much as possible in this phase of the process. For instance, suppose that a local composite operation was modified, the graph would be rebuilt and provided the operation calls the same simple operations, the building of the modified control-flow graph would use the same terminal control-flow graph for those operations. Suppose also that a remote operation which this operation calls was modified. The newly generated terminal control-flow graph for that operation would be inserted into the original non-terminal control-flow graph along with the other intact terminal control-flow graphs. If a local modification occurs which leads to calling a new operation, the framework simply gets the information for that operation and informs the tester that additional "human" work must be done. In other words, this would be outside of the scope of automatic operation.

Once the modified control-flow graph is generated, the process of the rest of this approach is identical to the approach outlined in the background in section 2.5. It compares the original control-flow graph with the modified control-flow graph to identify a set of dangerous edges. Once the dangerous edge list is computed, the tests which need to be rerun can be determined from using the test coverage information and the dangerous edge list, and executed.

Chapter 5: Automation of the RTS Approach

This chapter will discuss the automation of the approach to perform regression test selection on Web services. The chapter will begin by discussing the overall high-level architecture of the involved objects, including two broad approaches to automating the regression test selection technique outlined in the previous chapter. The adopted approach involves a new set of challenges which will also be discussed. Finally, the solutions to these new challenges will be presented in the form of agents.

5.1 Approaches to Automate Regression Test Selection for Web Services

This section will focus on the types of approaches applicable to automating the regression test selection technique and will begin with the discussion of two broad architectural systems which could be applied to produce the automation, the selected approach, and why the selected approach was adopted. Lastly, this section will briefly discuss some of the new challenges the adopted approach presents.

There are two broad mechanisms by which the approach could be automated: 1) A centralized solution, which monitors and controls all parties' modifications and testing; and 2) a distributed solution, in which each service monitors its own modifications, notifies interested parties, and performs the regression test selection process and the regression testing for it. The first approach is straightforward to implement, but may not be feasible for some parties which want to maintain a certain level of control over the processes for any number of purposes such as information security and granularity control. The second approach alleviates these concerns because it allows everyone to maintain a full level of control over the process. The approach adopted for use in regression test selection will be the distributed solution, since distributed solutions will allow greater levels of control over the process for the individual parties.

In a distributed solution, testing Agents, one for each service, will be responsible for monitoring the service they are responsible for, and when necessary perform regression test selection and then the regression testing processes. However, since modifications can occur concurrently, the challenges which arise from this concurrency must be carefully analyzed, since the agents will ultimately be responsible for handling this as well.

5.2 Concurrency Challenges

This section will discuss the concurrency issues which can occur in a distributed automated regression test selection system. The section will describe the types of issues which can occur and provide some example scenarios for each. The types of issues which can arise are test consistency, coverage issues, and communications issues.

5.2.1 Coverage Conflict Coverage issues arise from the manner in which the regression test selection technique is performed. As described in the background, the regression test selection technique uses three pieces of information, which were all built as described in Chapter 4. The first piece of requisite information is the control-flow graph of the operation. After construction, each control-flow graph is a directed graph with nodes corresponding to code artifacts, such as statements or blocks, and edges representing control-flow between them. The second piece of information required is a set of test cases. After construction, the set of test cases is an ordered list of the inputs to the service along with the outputs which correspond to the given inputs. The last part is the coverage information, which after construction, is a list of test case identifiers of the test cases (from the second piece of information) and paths each of those test cases cover in the control-flow graph (from the first piece of information). Upon receiving a modification, a new control-flow graph is built and the regression test selection algorithm performs a dual-traversal over both the newly created control-flow graph and the original one. This is performed exactly as described in the background in section 2.5. The result is a set of dangerous edges, which is directly used to determine which test cases to select. The set of dangerous edges form all the paths which begin at the start node of the control-flow graph and lead to all modifications.

Definition: *A revealing path is a path in a control-flow graph which starts at the start node of the control-flow graph and leads to a modification.*

This definition will help simplify any discussion involving dangerous edge lists, because a dangerous edge list can now be described as the set of all revealing paths. Since the regression test selection algorithm uses the location of each modification to select test cases, the location of the multiple concurrent modifications can have a negative impact on the result. More specifically, if two, or more, modifications result in the selection of the same group of test cases, two basic issues arise: 1) if any of the shared test cases (selected by both the algorithm for both modifications) result in a fault, the tester will not be capable of determining which modification was responsible. This issue is known as fault locatability. 2) Redundant test cases are being run which result in a waste of resources. Any two or more modifications which suffer from issues due to the location of their modifications in relation to one another are said to be in coverage conflict.

Coverage conflict will be defined more formally. The following notation will be used throughout this section: M_1 is the first modification and M_2 is the second modification. Each of these modifications corresponds to modified control-flow graphs, which will be denoted using C_1 and C_2 for M_1 and M_2 respectively, each of which will be compared to the original control-flow graph C. A dual-traversal shall be performed on C_1 and C as well as on C_2 and C which will result in two corresponding dangerous edge lists, DE_1 and DE_2. The dangerous edge lists will be used to select corresponding test cases T_1 and T_2.

Definition: *Two modifications, M_1 and M_2 are in coverage conflict if the regression test selection algorithm selects some common test cases. In other words, M_1 coverage conflicts with M_2 if $T_1 \cap T_2 \neq \emptyset$.*

Since the number of test cases can be arbitrarily large, another way to determine when conflict occurs is required. Let us begin by looking at how two modifications in coverage conflict relate to one another.

Theorem 5.1: *Two modifications, M_1 and M_2 are in coverage conflict if one of the revealing paths from one dangerous edge list is a subgraph of one of the revealing paths of the other dangerous edge list.*

Let us focus on those situations in which two modifications, M_1 and M_2, result in coverage conflict, namely when the same set of test cases are selected by both modifications, or $T_1 \cap T_2 \neq \emptyset$. That is when two modifications result in shared test cases. Furthermore, let us focus on only $T_1 \cap T_2$, or the shared test cases, and the situation which leads to non empty intersections. The dangerous edge lists, DE_1 and DE_2, are composed of all revealing paths. The set of selected test cases, T_1 and $T_{2.,}$ are selected using those two dangerous edge lists, DE_1 and DE_2, and the coverage information, which is a lookup table with test cases and their corresponding paths through the control-flow graph. Each path in the dangerous edge lists is compared to the listed paths for each of the test cases, and if a test case covers the entire dangerous edge path the test case is selected. For any test case to be selected, it must cover an entire revealing path in the dangerous edge list. For any test case to be selected in two separate modifications, the test case must cover an entire revealing path in both dangerous edge lists. This implies that one revealing path of one dangerous edge list must lie in the path of the one of the other revealing paths in the other dangerous edge list, which is the very definition of subgraph. *QED*.

5.2.2 Test Consistency Another type of issue which must be carefully considered is that of test consistency. Test consistency involves ensuring that each test case gets a consistent view of the system under test. For instance, one begins testing A, which calls B, but before finishing the test cases, B notifies A that it was modified. Some of the test cases ran on the old B and some ran on the new B. There is no way to identify which test cases ran on which version of B. This issue is important in terms of ensuring the accurate reporting of test results. If test inconsistency is never allowed to happen, then all test results should be guaranteed to be accurate. Another form of ensuring testing consistency is ensuring that once a system reaches a stable-state the test cases which were last run were all consistent. This work refers to this stable-state consistency as eventual test consistency. Either way, test inconsistency can only occur in those cases in which coverage conflict occurs, however, the two issues can be handled separately.

5.2.3 Communication Issues Communications issues arise from faulty or slow communications channels and usually involve delayed messages, dropped messages, or messages being sent to the wrong location. This issue has an interesting impact on the described approach in terms of the control-flow graphs. Suppose that we have two modifications, M_1 and $M_{2,}$ and the two modifications occur in that order. Suppose that the notification for the first modification was delayed and the second notification arrives first. The control-flow graph for the second modification actually contains the information for the first modification (the first modification modifies the system and the second modification modifies the system which was just modified). This implies that when the second modification arrives before the first, the system will perform the entire regression test selection process for both modifications, selecting the test cases which need to be run for both modifications. The notification of the first modification can safely be dropped.

5.3 Solutions in the Form of Agents

The agents are responsible for handling all of these issues as necessary. The agents will handle the issues outlined in the previous section using a set of algorithms to ensure correct operation. The definition of correct operation depends on what the tester is specifically interested in solving. All of the testers in all cases are interested in solving issues related to communication, but have different goals in their relation to the coverage and test consistency issues discussed in the previous section.

Suppose that the testers are not interested in complete fault locatability or a running test consistency. This implies that the tester is only interested in ensuring that eventually, once all modifications arrive, the system is tested completely the last time each of the test cases were run they were consistent. This provides the utmost in terms of concurrency while still ensuring that the tests are eventually consistent.

However, suppose that the tester is interested in full test consistency. This implies that some mechanism is in place that ensures that every test case is consistent as it completes. Finally, ensuring fault locatability also ensures test consistency and is performed by ensuring that there are never any coverage issues. This implies that concurrency is allowed only in cases in which there are no coverage issues. The two of these will be left for future work, and only the algorithm which ensures eventual test consistency will be discussed in detail.

5.4 Eventual Test Consistency Agent

In this section, the agents which are responsible for maintaining eventual test consistency will be discussed. First, the data structures, objects, and methods available to the agents will be discussed. Then the algorithm will be described in terms of how it operates under internal and external modifications. Finally, a proof that the agent performs as expected will follow the discussion.

5.4.1 Data Structures and Methods Every agent, each of which corresponds to a service, has and maintains the following data structures:

- Up-to-date terminal control-flow graph from the point of view of the site, *C*.
    - This data structure is initialized as described in section 4.2

- Subscriber list, *NotifyList*.
    - This is a list of the addresses of agents which monitor those services which call this service directly while performing their duties.
    - The agents build this list by sending their address when they request the terminal control-flow graph of the services they call.
    - An empty list implies that no other service calls this service
- Identification of the current local site, *myID*.
    - the URI of the agent
- Logical clock object, *LC*, which has three operations:
    - *getLC()* – gets the current value of the logical clock
    - *incrementLC()* – increments the current value of the logical clock
    - *setLC(timestamp)* – sets the current value of the logical clock to timestamp.
- A lists of tasks, *RunningTasks*
    - A list of concurrently running tasks which have a one-to-one correspondence with *Tester* objects which will be discussed in detail.
    - *RunningTasks* = $\{task_1, task_2, \ldots, task_k\}$, where
        - $task_i = (E_i, opID_i, ts_i)$ for $i = 1, 2, \ldots, k$.
            - E_i is the dangerous edge set;
            - $opID_i$ is the operation which was modified
                - Each service may have more than one operation and each operation is identified by its name
            - ts_i is the timestamp of the modification

    - Has two operations:
        - *addTask($task_i$)* – adds a task to be completed
        - *removeTask($task_i$)* – removes a running task
- A set of timestamps, *watermarks*
    - the last timestamp seen for each operation that this operation calls directly

Each of the tasks in *RunningTasks* corresponds to *a Tester* object, which are directly responsible for the execution of a given set of test cases. Each *Tester* object executes a set of test cases and maintains two separate lists, *to_test* and *done*, which hold the test cases which still need execution and the test cases which have already been executed respectively. The *Tester.Test* operation takes as parameters a set of test cases. As it finishes the execution of each test case, the *Tester* moves the test case from the *to_test* list to the *done* list. These two lists will be used to handle conflict when it occurs. The operation of this method is shown in figure 5.1.

```
Tester.Test(TC) {
   to_test = TC;
   done = empty;
   for each test case, tc, in TC {
      test tc;
      move tc from to_test to done;
   }
   send report to test manager;
   RunningTasks.removeTask(this.TaskID)
}
```

Figure 5.1: *Tester.Test* operation

Finally, there are a number of methods available to each of the agents. The first of these methods is used to select test cases using the dangerous edge list, and its signature is: *TestCase[] selectTestCases(EdgeSet es);* This is performed as described in the background in section 2.5.

Another method available to the agents is used to determine if two dangerous edge lists conflict, and its signature is: *boolean coverageConflict(EdgeSet a, EdgeSet b)*. This method will be discussed in detail as it is central to some of the algorithms. This function takes as arguments two edge sets containing two separate dangerous edge lists and returns whether or not the two dangerous edge lists conflict. This method uses Theorem 5.1, which states that two modifications are in coverage conflict if one of the revealing paths on one dangerous edge list is a subgraph of a revealing path in the other dangerous edge list. This problem of determining if one path is a subgraph of another path has a standard solution from graph theory, and will not be discussed in further detail.

The final method available to the agent is the *mergeDangerousEdgeLists* method, which merges the two given dangerous edge lists and returns a merged dangerous edge lists. In this agent, coverage conflict is handled by merging the two conflicting modifications. However, when the two modifications are merged, later incoming modifications must be able to determine whether or not they conflict with either of the two currently conflicting modifications. Since dangerous edge lists are used to determine whether two modifications conflict, the dangerous edge lists must be merged. In order to merge the two dangerous edge lists, all non-conflicting revealing paths and the shorter of the two conflicting revealing paths form the result. The reason the shorter of the two conflicting revealing paths is chosen is that the shorter revealing path will conflict with an incoming modification which conflicts with either of the two merged conflicting modifications. This method is performed by traversing the two dangerous edge lists and adding all edges to the result which the algorithm crosses until either each of the current nodes is a leaf or one is a leaf and the other is not. The result is a merged dangerous edge list, which ensures that if an incoming modification were to conflict with either of the two conflicting modifications, it would conflict with the merged dangerous edge list.

Now that the data structures and methods available to the algorithms are in place, a functional description of the architecture will be discussed.

5.4.2 Agent Operation This agent is responsible for ensuring that all modifications will eventually be tested. This algorithm will not be capable of determining where the fault occurred (fault locatability), but will ensure that once the system reaches a steady-state the tests will be consistent (eventual consistency). Also, during the operation no redundant test cases will be executed.

In addition to the data structures and methods previously discussed, the agent has only two important operations: *receive(MSG),* which is for remote modifications, and *localModification(),* which is for local modifications.

```
localModification() {
   update the global CFG of the local site into $C_{new}$;
   LC.incrementLC();
   lc = LC.getLC();
   create a message, MSG=(myID, $C_{new}$, lc)
   send MSG to each of the subscribers in NotifyList;
}
```

Figure 5.2: Agent *localModification()* operation

The *localModification()* operation, shown in figure 5.2 creates the new control-flow graph of the local site, increments the logical clock, and sends the newly created control-flow graph to all of its subscribers so that they can start their regression test selection. An assumption here is that before developers can commit their own modifications into the system they have must have already thoroughly tested their own system. This is because unit-testing must always precede integration and system testing.

The other operation, *receive(MSG),* is specifically designed to handle the concurrency issues, regression test selection, and the management of the Tester objects. It takes a message, *MSG*, which describes a modification which occurred in one of the services that the agent calls directly as a parameter. This message, *MSG*, contains the terminal control-flow graph of the agent informing this agent of the modification, an identifier identifying the agent informing this agent of the modification, and a timestamp.

The operation begins by determining whether or not the arriving message is the most recent one received from that sender. If it is not the most recent message, it is discarded. If it is the most recent message, the watermark which holds the most recent message for each sender is updated. The logical clock for this agent is then updated. The agent then uses the information made available to it from the initialization phase (specifically how the original control-flow graph was created) to insert the modified terminal control-flow graph of the subordinate site (or calling site) into the terminal control-flow of the agent's site to create a new terminal graph for this site, C'. After generating C', the agent sends this control-flow graph to every subscriber in its subscriber lists, *NotifyList*, along with its timestamp and identifier. The agent then compares the two control-flow graphs, C and C', by traversing them simultaneously to compute the dangerous edge list. The agent then selects the test cases based on the resulting dangerous edge list (using the method *selectTestCases*). The agent then creates a new

Tester for this modification and adds the newly selected test cases to it. The *receive(MSG)* operation is shown in figure 5.3.

```
receive(MSG) {
   if (MSG.TS < watermark.get(MSG.ID)) {
      LC.setLC(max(LC.getLC()+1, ts) + 1);
      watermark.put(MSG.ID,MSG.TS);
      build C’ by embedding MSG.CFG into C;
      build MSG’ using (C’, myID, LC.getLC())
      send MSG’ to every subscriber in NotifyList
      compute the dangerous edge set, $E_0$, by comparing C and C’;
      TC = selectTestCases($E_0$)
      Create a new tester, Tester’, to test TC;
      Tester’.to_test = TC;
      foreach task = ($E_d$, $opID_d$, $ts_d$) $\in$ RunningTasks {
         if (coverageConflict($E_d$, $E_0$)) {
            pause $Tester_d$;
            Tester’.to_test = $Tester_d$.to_test $\cup$ Tester’.to_test;
            $Tester_{new}$.done = $Tester_{new}$.done $\cup$
               ($Tester_d$.done – Tester’.to_test);
            $E_0$ = joinDangerousEdgeLists($E_0$, $E_d$)
            Kill $Tester_d$;
      }  }
      start $Tester_{new}$;
      add task’=( $E_0$, opID, ts) to RunningTasks;
      LC.incrementLC();
} }
```

Figure 5.3: Agent *receive(MSG)* operation

Then the agent determines whether the newly generated dangerous edge list conflicts with any of the dangerous edge lists currently running. This is determined by using the *coverageConflict* method described earlier. If the method returns false for all running modifications, the algorithm simply starts the *Tester* created earlier, adds the task to *RunningTasks*, and increments the logical clock.

If any of the currently running *Tester*s do conflict with the incoming modification, it is paused. The algorithm deals with conflict by joining the incoming modification to the conflicting modification. The algorithm reconciles the *to_test* and *done* lists for the new *Tester* using the information from the paused *Tester*. The *to_test* list gets a union of the *to_test* list of the paused *Tester* along with the *to_test* list of the newly created *Tester.* This ensures that the tests which need to be rerun again from the new *Tester* get rerun along with any tests which were not in the new *Tester* but were in the paused *Tester*. The algorithm then sets the *done* list of the new *Tester* to the done list of the paused *Tester* minus *to_test* of the new *Tester*. This ensures that the tests which were finished by the paused *Tester* which need to be rerun are removed from the *done* list, but the test cases which do not need to be removed are not removed. The last thing which must be performed in order to join the two modifications is to join the dangerous edge lists. This allows for another modification to determine if it conflicts with either of the two joined conflicts. This is done using the *joinDangerousEdgeLists* method described earlier. Finally, the paused *Tester* is then killed, and the newly created *Tester* is started.

5.4.3 Correctness of the Agent Algorithms The specific goal of this section is to prove that the Agent functions correctly under normal operating conditions. Normal operating conditions implies only that all messages sent between cooperating agents are reliably delivered. The goal of the Agent is to ensure that every modification the system undergoes is eventually tested and when the system reaches a stable-state it will be consistent. The Agent does not ensure that every earlier test is consistent, or that every run of the *Tester* object is consistent. However, it does ensure that after all the modifications are received, every part of the modified system will have been tested and those tests will be consistent. Additionally, it attempts to reduce the number of redundant tests cases which need to be executed.

The goals of the system will be discussed more formalized. The algorithms must ensure that all modifications are eventually tested and this implies only that no modification is missed. This particular goal is straightforward to prove since we assume reliable delivery of all notification messages. Since no messages will ever be lost, the algorithms must only ensure that every modification is tested. Since the algorithms in place are safe, the algorithms will ensure that every part of the system which was modified will be tested. This is an important point, since now the algorithms must only ensure that the selected tests eventually get performed. The algorithms must ensure that upon reaching a stable state, the tests will be consistent. More specifically, the algorithms must ensure that if the system changes during testing the tests which covered the modification, and thus are inconsistent, are restarted. As mentioned earlier, the only tests which are inconsistent are those that are shared by two conflicting modifications. The algorithms must then ensure that if two modifications conflict, none of the shared test cases is allowed to remain in the *done_list* and all of the shared test cases are on the *to_test* list. This ensures the consistency requirement eventually because the shared tests always gets retested when a conflicting modification appears, and when there are no more modifications to report, the test cases which finish last are consistent.

Assuming that the overall system starts in a correct state, there are a number of cases which will be considered and these cases will be numbered for clarity. In case 1, the modification is local to the Agent. This algorithm is presented in figure 5.2. At any time, if a local modification occurs, the control-flow graph is regenerated as described in section 4.2.2.1 and sent to all interested parties in the notification list. This ensures that every other agent interested in the locally modified agent will receive a notification that this agent has been modified. This agent will always build the control-flow graph from the most recent view of the system and send it so that the control-flow graph it sends will be correct in terms of the most current view of this agent. This particular algorithm of the agent functions correctly in respect to ensuring that the local modification is both propagated to all interested parties and the most current.

The rest of the cases involve remote modifications and therefore the *receive(MSG)* algorithm of the Agent presented in figure 5.3. Case 2 occurs when the incoming modification is the first modification this agent has received, which is a trivial case. Upon receiving this message, it accepts the modification, adjusts the watermark, builds the new control-flow graph, sends it to all subscribers, computes the dangerous edge lists, selects test cases, and creates a tester. Since there are no currently running tasks, the tester is simply started and added to the currently running tasks. Assuming that a steady-state has been reached with this modification, the algorithm ensures that at least this modification will be tested. If this is the only modification, it also ensures that when the system is finished testing this modification the tests will have been consistent. They are consistent since there can be no conflict because there is only one modification.

Case 3 is non-trivial because it involves the Nth modification to arrive at the agent. There are several parts to this case which will each be considered separately as sub-cases. Case 3.1 involves an incoming modification which carries a lower timestamp that the one last received for that operation. If this happens, the incoming message is discarded. If the incoming message has a lower timestamp than the one which was last received the message was delayed and thus arrived after a message which was sent after it. The message is discarded because the control-flow graph of the less recent modification message is part of the more recent control-flow graph. This is true regardless of whether or not either of the two modifications conflict with any of the running testers. The algorithm still ensures that every modification will eventually get tested because the modification will have existed in another modification notification which allows us to safely assume that the modification is tested upon receiving the other modification notification which still ensures that the modification will be tested. The algorithm additionally ensures consistency with this approach because even if the modification caused a conflict, the conflict and the resolution would be correctly handled by the other notification and thus would be redundant to be done again. All other sub-cases of Case 3 will assume that the modification has a higher timestamp than the last received message.

Case 3.2 entails an incoming modification which does not conflict with currently running tasks. Upon receiving the modification, the algorithm builds a new control-flow graph, sends it to all the subscribers, computes the dangerous edge list, and then begins comparing it to all the currently running tasks. In order to correctly handle these situations, the operation must be able to determine if the incoming modification conflicts with one of the running conflict. As discussed earlier, this is correctly determined by the method *coverageConflict*. In this case, the *coverageConflict* method determines that they do not conflict and since it does not conflict with any of the currently running tasks, the algorithm performs the same steps as the steps incurred when there were no other modifications in the system. Again, assuming that this modification leads to a steady-state, the algorithm ensures that all modifications are tested and those tests are consistent. All modifications will be tested since no tester is stopped and all tests will be consistent since there was no coverage conflict.

The last and most interesting case, Case 3.3, involves an incoming message which does conflict with a currently running task. Upon receiving the modification, the algorithm builds a new control-flow graph, sends it to all the subscribers, computes the dangerous edge list, and then begins comparing it to all the currently running tasks. As described earlier, the *coverageConflict* method is used to determine if the incoming modification conflicts with any of the currently running tasks. Since the method correctly determines that there is conflict if it exits, it determines that the incoming modification does conflict with a currently running task. As mentioned earlier, when two modifications are in conflict, the agent will deal with the conflict by merging the two conflicting modifications. The merging that takes place merges both their dangerous edge lists and their test cases. So after conflict is determined, the *Tester* object the incoming modification conflicted with is paused and the newly created *Tester* must be merged with the conflicting *Tester.*

In the algorithm, the test cases are merged first. The first part of this merging process involves the *to_test* list. The newly created Tester already contains the test cases the incoming modification selected and the non-conflicting test cases from the conflicting *Tester* are added to the list. This is performed by taking a union of the *to_test* list of the incoming modification and the *to_test* list of the conflicting modification. This is performed because any test which was not finished and on the *to_test* of both the incoming modification and conflicting modification will be on the *to_test* list of the merged result. This procedure ensures two things: 1) any finished shared tests will be added to the *to_test* list of the result ensuring that these inconsistent tests will be retested and 2) that any finished non-shared tests will not be added to the *to_test* list of the result ensuring that no finished test case which was consistent will be missed.

The second part of this merging process is the handling of the *done* list. The newly created Tester must be set to be the union of the *done* list of the incoming Tester (may not be empty, especially if this modification conflicted with another earlier) and the *done* list of the conflicting tester minus the *to_test* list of the newly created Tester. This is performed to ensure that any non-conflicting test case which was finished in either remains on the *done* list and that any conflicting test case does not. This procedure also ensures two results: 1) any finished non-shared tests results remain on the *done* list ensuring that the result of all such consistent results are not lost and 2) any finished shared test results do not remain on the *done* list ensuring that no consistent test will be reported.

The two procedures which handle the merging of the test cases and their results ensures that no consistent result is lost and all results non-consistent will be lost and that any consistent test cases will not be run again and that any tests which were made inconsistent by the incoming modification will be run again. The merging of test cases described earlier ensures the future consistency of the test cases the two conflicting modifications share as well as the consistency of the test cases they do not share. However, suppose there is another running task which conflicts with the incoming modification. In this type of scenario, the dangerous edge list which is used to determine conflict must also be merged so that other currently tasks and incidentally future incoming modifications can determine if they conflict with either of the two conflicting modifications. As described in section 5.4.1, the *mergeDangerousEdgeLists* method correctly performs this function. In doing so, it ensures that the rest of the currently running tasks can be checked for coverage conflict using the method *coverageConflict*. It additionally, ensures that later incoming modifications will be correctly handled.

Since the algorithm correctly determines whether or not an incoming modification conflicts with a currently running modification, the two cases 3.2 and 3.3 are the only two possibilities, and in both of those the algorithm correctly functions. Additionally, supposing that the incoming modification conflicts with more than one currently running task, this is handled correctly by the merging of the two modifications. The incoming modification is merged with the first conflicting task and then the merged modification is then merged with the second. Since the merging does correctly ensure that the merged modification does conflict with the second conflicting task, the second merging is guaranteed to take place. Additionally, since the merging process correctly ensures that no shared test is left finished and no non-shared test is lost, it correctly ensures the future consistency of the tests across all merged modifications. Again, assuming that this modification leads to a steady-state, the algorithm ensures that all modifications are tested and those tests are consistent. All modifications will be tested since even though the tester is stopped and killed, the finished consistent tests are guaranteed to be on the newly created merged Tester and thus never lost. All tests will also be consistent since the newly created merged Tester ensures that the shared tests will be run again ensuring their final consistency. *QED*

Chapter 6: Empirical Study of Proposed RTS Technique

As mentioned in section 2.3 of the background, regression test selection is only beneficial when the cost of running all tests is greater than the costs of running the tests selected by the regression test selection process and the regression test selection process. This chapter will present an empirical study of the outlined approach for applying a safe regression test selection technique to Web services.

6.1 Introduction

Unlike traditional applications, there are no standard frameworks available which can be used as benchmarks to test regression test selection techniques on Web services which would present a set of challenges to any empirical study performed on Web services. The frameworks for which the empirical study will rely on were developed for the sole purpose of the empirical study. The validity of the study will be assured by the manner in which the frameworks are tested and the manner in which the results are presented. However, the validity of the approach to apply empirical study to these systems is seriously and negatively impacted by the lack of previous studies along this vein for Web services. There will be a number of systems presented along with descriptions of why these systems were selected for use in this study.

The empirical method and study is based on the study presented by Rothermel and Harrold, in which they performed an empirical analysis of the performance of their regression test selection technique for monolithic applications. This empirical study will follow their approach very closely. In their study, they used a variety of standard systems, augmented with the requisite information (control-flow graphs, test cases, and coverage information), modified the system, and then ran their regression test selection mechanism. Specifically they compared the cost of performing the regression test selection algorithm added to the cost of executing the selected test cases with the cost of executing all test cases. Their measure of cost was what the authors termed “wall clock time”, which meant that they simply timed the results using a start time subtracted from a finish time. Since no standard systems are available in a Web services world, a variety of systems which are representative of real-world Web services were developed for this purpose.

In the work by Rothermel and Harrold, since they were augmenting code which had not been developed by them with test cases they were not provided with, they followed a rule which states that every path in the system must have at least 30 test cases to exercise it. This provides an even distribution of test cases throughout the system, which although it is not representative of a real system due to the inherent priority of different paths, it is a requirement since the tester will not always know which paths through the system would be more important than others and an even distribution ensures equal consideration for each path. This particular rule is strictly enforced in this work.

Lastly, they used a random group of people to alter the code in random ways and then used the modified versions of the code to perform regression test selection. They performed this experiment a number of times recording the costs. They then compared those costs to the cost of executing all test cases. This analysis will proceed very similarly: The underlying control-flow graphs will be altered at random, and the resulting control-flow graphs will be used in the regression test selection process.

The regression test selection system presented by this work will be evaluated in the following way: The cost of performing all tests throughout the system will be compared with the cost of performing regression test selection everywhere necessary along with the cost of performing only the selected tests throughout the system. This cost comparison is indicative of comparing the test-all approach to the selective test approach, which was performed by Harrold and Rothermel. Since the regression test selection and regression testing processes are performed in a distributed and concurrent fashion, the results of such an study will present an idea of the cost savings in terms of overall work performed.

This cost comparison will be evaluated by first augmenting each of the five systems with test cases, coverage information, and control-flow graphs, which is performed as described in section 4.2. Once this is done, the experiments were performed. Each experiment was performed in three basic steps. First, the test harness randomly selected a node, which could be in any one of the participating services, and modified it. Second, regression test selection is performed for each of the affected services and the time taken to do so is recorded. Third, the mechanism ran the selected test cases from the second step and recorded the time required for the entire set to run.

Even though Web services are by nature distributed entities which can be thought of as running on many separate machines as easily as on the same machine, all tests were run on the same machine as the service, and all services were on the same machine. This was done for two reasons: 1) the agents themselves are expected to be as near as possible to reduce costs and 2) including transmission costs would only add to the cost of running the test cases which would only deepen the comparison not change it, since as the cost of running tests increase the likelihood of regression test selection winning the comparison increases. Lastly, the cost of performing the algorithm is added to the cost of running the selected test cases and recorded. This final recorded cost is to be compared to the cost of running all test cases for each of the services, which was recorded at the onset, and thus was only run once.

Five systems will be discussed in the five sections following this one. Each of the five systems will be introduced, including descriptions of the architecture of the system, the test augmentation, and why each system was chosen for use in this study. The results of the empirical study will be presented and discussed in the final section.

6.2 Purchase Order System

In this section, the approach will be applied to a group of Web services in a simplified purchase order system. This system was developed to shed light on the approach, considering this system is used to describe many parts of the work thus far. Additionally, it will show that even small simple systems can benefit from regression test selection. Purchase order systems are used in a variety of case studies presented in a number of books [49, 50, 51] as well as a number of technical articles about Web services [52, 53]. The authors of these works use them in their work because they are indicative of the way the real-world Web services interact, and are also fairly intuitive to describe and understand. A simplified purchase order system was selected for use in the study for these very reasons.

The simplified purchase order system, which is shown in figure 6.1, consists of four Web services, which are: 1) a hardware service which accepts orders of hardware related products only, 2) a software service which accepts orders of software related products only, 3) a service which only accepts office supply orders which are not software or hardware, and 4) an ordering service which accepts an order and forwards the order to the correct service depending on the order. This particular framework relies very heavily on the content-based routing pattern [54], which changes the route of a message based on the content of the message. In this case, the ordering service routes the orders to the suppliers which will supply the order based on the order itself. This system is dramatically simplified in terms of capabilities and complexity, and in terms of being only two layers deep thus ignoring vendors and subcontractors. On the other hand, its simplicity makes it possible to visualize the different parts of the regression test selection framework.

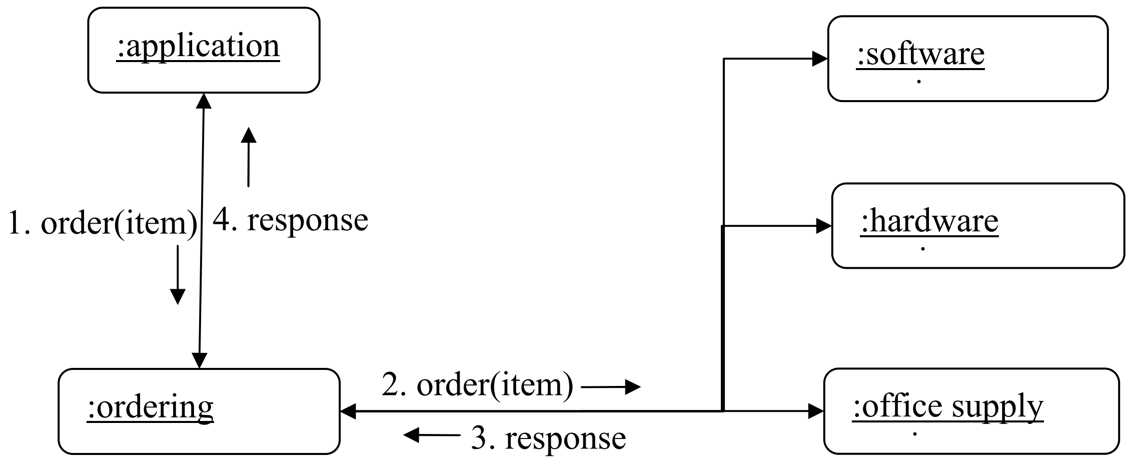


Figure 6.1: A modified UML diagram of the purchase order system

The control-flow graphs will be described next, including the control-flow graphs of each of the participating services. In order to reduce the size of the control-flow graphs, the granularities of the control-flow graphs shown for this system are at the block level. The control-flow graph for the hardware service which accepts hardware purchase orders and fulfills them is presented in figure 6.2 (a). Note that the control-flow graph presented shows a brick node which represents the entry point of the service and a checkerboard node which denotes the exit point of the service. The graph only has 4 nodes altogether and a total of three unique paths. The three unique paths require a total of 90 test cases to ensure that each path is covered by at least 30 test cases.

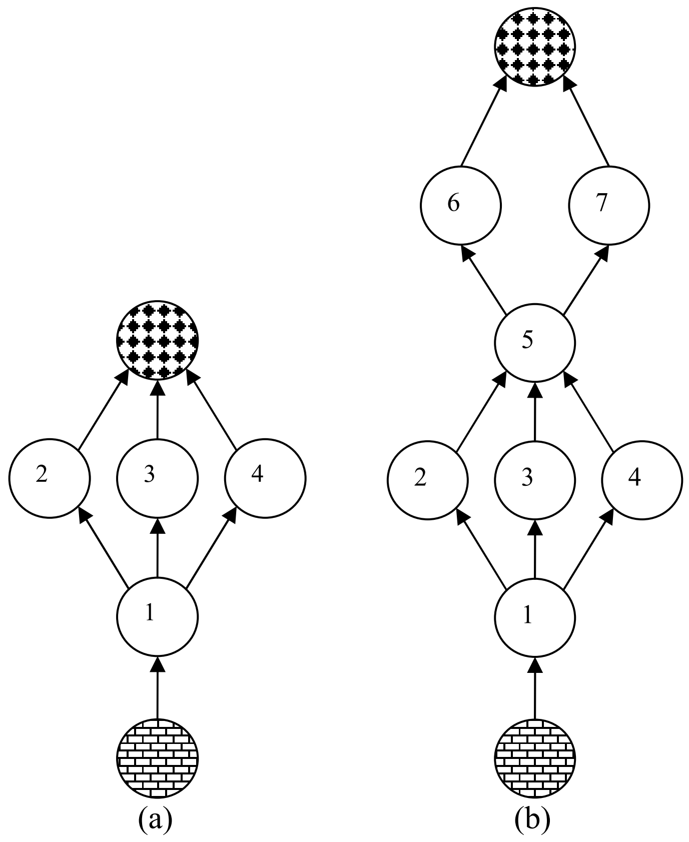


Figure 6.2: Control-flow graph of hardware (a) and office supply services (b)

The control-flow graph of the office supply purchase order service which accepts purchase orders which are neither hardware nor software orders and fulfills them is presented in figure 6.2 (b). This graph has a total of 7 nodes, not counting the brick entry and checkerboard exit nodes and has a total of 6 possible paths through the system, which requires a total of 180 test cases by the 30 test cases rule.

The next, and final simple service, is the software purchase order service and it accepts software-based purchase orders and fulfills them. The control-flow graph in figure 6.3 is the control-flow graph for the software purchase order service, and it has a total of 18 nodes with 11 total paths through the service which require a total of 330 test cases to ensure that each path is covered by at least 30 nodes.

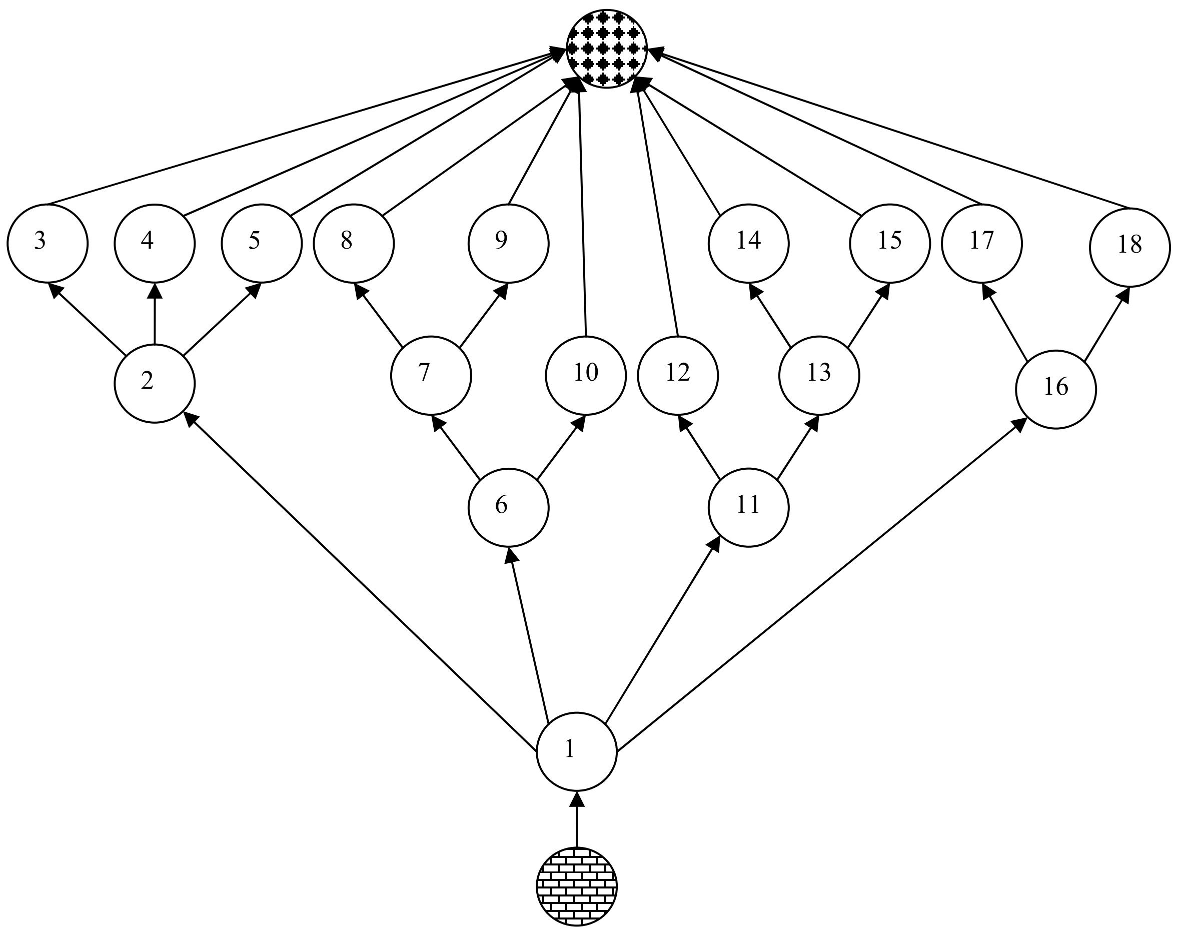


Figure 6.3: Control-flow graph of software service

The composite service accepts purchase orders and forwards them to the correct service depending on the order. The control-flow graph for this service was composed as described earlier and is shown in figure 6.4. Note that the parts of the graph which correspond to the individual terminal services are labeled (hpos, opos, and spos) and note how the numbering changed. The purchase order system first decides whether or not to call the hardware purchase order service, then the software purchase order service, and then the other purchase order system. This graph has a total of 30 nodes, not counting the brick entry and checkerboard exit nodes and has a total of 20 possible paths through the system, which requires a total of 600 test cases.

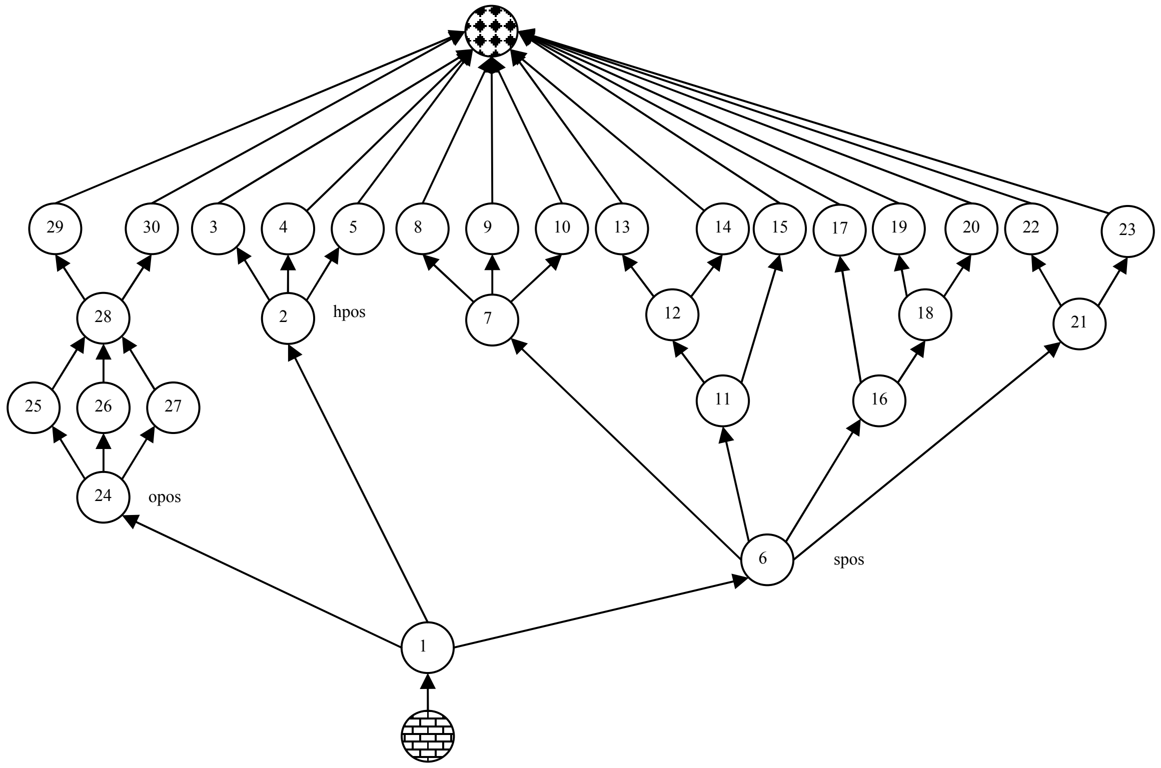
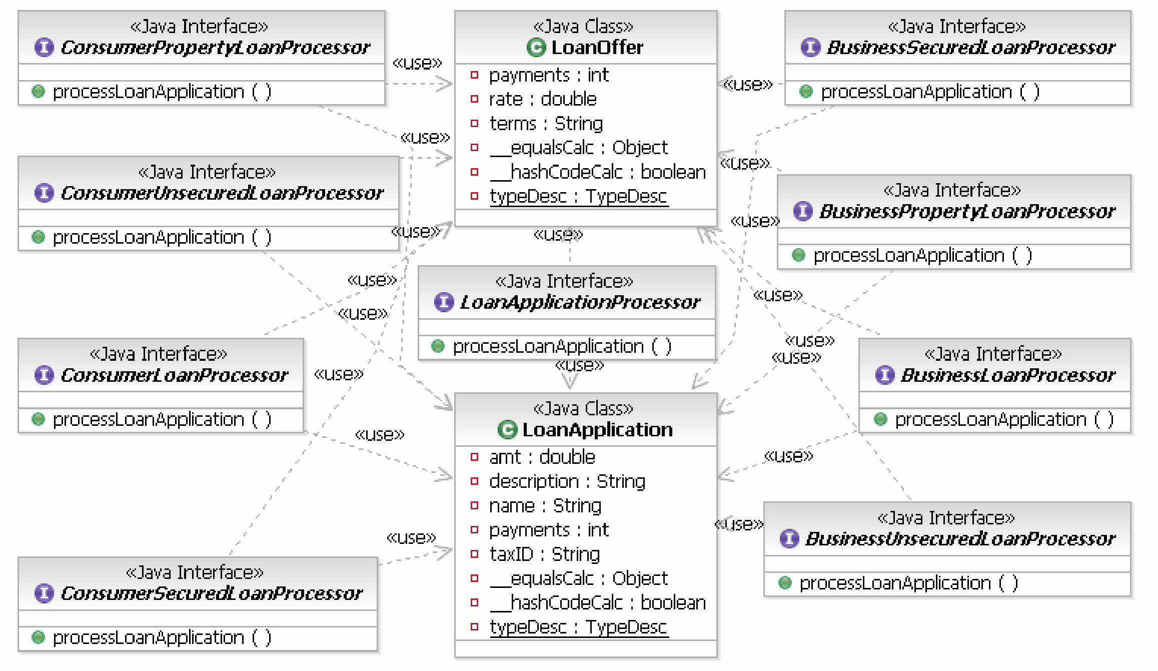


Figure 6.4: Control-flow graph of composite purchase order service

6.3 Loan Application System

In this section, the approach will be applied to a group of Web services in far more complex system than the previous system. This system is a bank loan system which accepts and processes loan applications. Loan applications are frequently used as case studies in a variety of books [55, 56, 57] and technical articles [58, 59] concerning Web services. The systems are used to showcase a variety of topics related to Web services for two basic reasons: 1) they are indicative of the way real-world Web services interact and 2) they tend to be fairly complex but fairly approachable since nearly everyone applies for a loan sometime.

A UML class diagram of the loan application system is shown in figure 6.5. Note that only the interfaces are shown to improve clarity.


«Java Interface» ConsumerPropertyLoanProcessor
processLoanApplication ( )
«Java Interface» ConsumerUnsecuredLoanProcessor
processLoanApplication ( )
«Java Interface» ConsumerLoanProcessor
processLoanApplication ( )
«Java Interface» ConsumerSecuredLoanProcessor
processLoanApplication ( )
«Java Class» LoanOffer
payments : int
rate : double
terms : String
__equalsCalc : Object
__hashCodeCalc : boolean
typeDesc : TypeDesc
«Java Interface» LoanApplicationProcessor
processLoanApplication ( )
«Java Class» LoanApplication
amt : double
description : String
name : String
payments : int
taxID : String
__equalsCalc : Object
__hashCodeCalc : boolean
typeDesc : TypeDesc
«Java Interface» BusinessSecuredLoanProcessor
processLoanApplication ( )
«Java Interface» BusinessPropertyLoanProcessor
processLoanApplication ( )
«Java Interface» BusinessLoanProcessor
processLoanApplication ( )
«Java Interface» BusinessUnsecuredLoanProcessor
processLoanApplication ( )
«use»
Figure 6.5: UML class diagram of loan application system

This particular loan processing system is part of a bank which has multiple lines of credit and accepts applications for each of them through a centralized loan acceptor. The end-users may also call the services responsible for business and personal loans separately, as well as the services which handle the individual lines of credit, such as property, secured, and unsecured loans. This system also relies on the content-based routing pattern [54], and in this case, the type of loan being applied for will decide which service to send each request to. This system will show that even moderately large systems can benefit from regression test selection.

The bank loan application system consists of nine Web services, which are: 1) a loan acceptor service which accepts all loan requests for the bank, 2) a loan acceptor service which accepts all business loan requests for the bank (which originate from businesses, 3) a loan acceptor service which accepts all consumer loan requests for the bank (which originate from customers of the bank), 4) a business property loan acceptor service which accepts and processes business property loan requests, 5) a business secured loan acceptor service which accepts and processes business secured loan requests, 6) a business unsecured loan acceptor service which accepts and processes unsecured loan requests, 7) a loan acceptor service which accepts and processes consumer property loan requests, 8) a loan acceptor service which accepts and processes consumer secured loan requests, and 9) a loan acceptor which accepts and processes consumer unsecured loan requests. This system is very indicative of the way modern business processes interoperate considering that each of the loan types would be handled very differently both in terms of processing as well as in requirements.

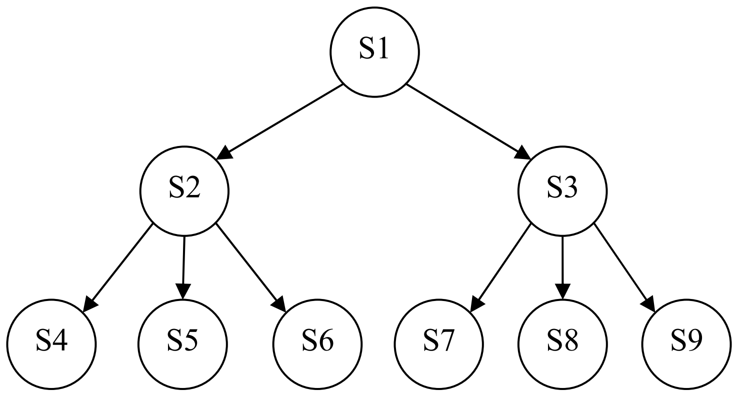


Figure 6.6: Call graph of loan application system

Figure 6.6 shows the call graph of the loan application system. The labels of the call graph correspond to the services which use the numbering in the description prefixed with a S. The granularity of the control-flow graphs for this system is all block level, meaning that there will be one node for each block. Table 6.1 lists the paths, nodes, and test cases for each of the services along with a total for all of the services in the Loan Application System.

Service	Nodes	Paths	Test Cases
S1	177	111	3390
S2	103	71	2130
S3	72	41	1230
S4	31	22	660
S5	36	25	750
S6	28	19	570
S7	19	12	360
S8	26	15	450
S9	25	13	390
Total			9930

Table 6.1: Loan application system totals

6.4 Loan Brokerage System

In this section, the approach will be applied to a group of Web services which form a loan brokerage system. This system is a significant expansion of the previous section, the bank loan application system. The bank loan application system accepted applications for a single bank with multiple lines of credit, which implies that the acceptor application determined which line of credit processor to send the application to. The loan brokerage system accepts a loan application and sends the application to a number of competing banks and returns the best offer in terms of what the customer is looking for, such as the rate. Even though there is a central loan application processor, there are several member banks which the application may send the application to. Additionally, unlike the previous loan system, this system uses a centralized credit score service which retrieves the credit score of the applicants. Loan brokerage systems, very much like loan application systems are used in a variety of books and technical articles for the same reasons. However, this system relies heavily on the gateway pattern [54], which is really a combination of two smaller patterns (scatter and gather). The scatter pattern defines a mechanism by which one entity sends a message to many other entities. The gather pattern defines a mechanism by which one entity collects a set of related messages from many entities. Putting the two together, one system sends a request to many entities and collects the responses from those entities and based on some criteria selects only one of the responses. Lastly, this particular system was developed to show the effect that different granularities has on the approach if any.

The UML class diagram of this system is shown in figure 6.7. Again, only the interfaces for each of the services are shown for clarity.

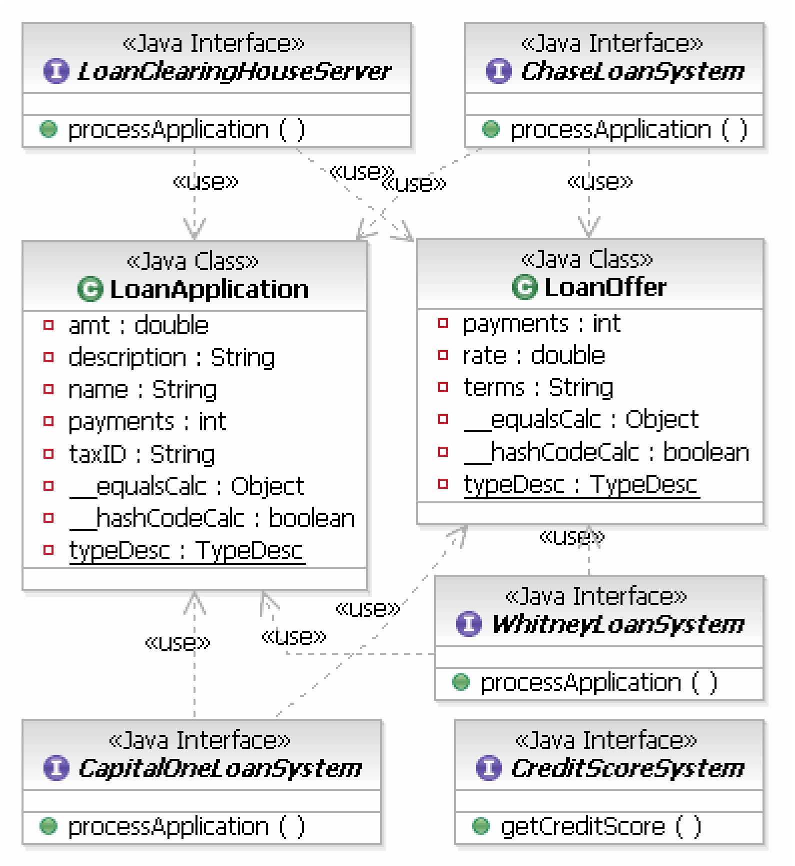


Figure 6.7: UML class diagram of loan brokerage system

The loan brokerage system is comprised of five Web services, which are: 1) a loan acceptor service which accepts loan requests, sends them to the three competing banks, and returns the best offer depending on customer needs, 2) a loan acceptor service which accepts and processes loan requests for the first bank, 3) a loan acceptor service which accepts and processes loan requests for the second bank, 4) a loan acceptor service which accepts and processes loan requests for the third bank, and 5) a credit score service which retrieves credit scores for loan applicants. This system is very indicative of the way modern business processes interoperate considering that each of the banks would have their own logic for determining their rates based on credit score.

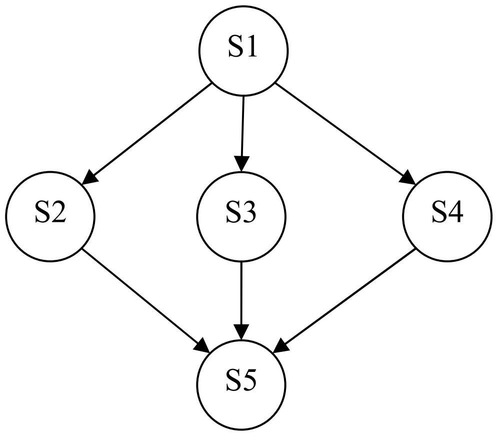


Figure 6.8: Call graph of loan brokerage system

Figure 6.8 shows the call graph of the loan brokerage system. The labels of the call graph correspond to the services which use the numbering in the description prefixed with a S. The terminal control-flow graphs will be described next, but the diagrams of the control-flow graphs will not be part of the discussion. Since the control-flow graphs of the majority of the services are too large to show directly, they will appear in one of the appendices. In this system, the granularities of the control-flow graphs are not all identical. This is by design to show that the approach can handle, as well as how it handles, mixed granularities.

The following table, Table 6.2, lists the paths, nodes, test cases, and granularity for each of the services along with a total for all of the services.

Service	Nodes	Paths	Test Cases	Granularity
S1	188	567	17010	Block
S2	132	198	5940	Statement
S3	49	225	6750	Block
S4	1	1	30	Operation
S5	13	9	270	Block
Total			30000	

Table 6.2:Loan brokerage system totals

6.5 Supply Chain Management System

In this section, the approach will be applied to a group of Web services which form a supply chain management system. Managing a supply chain implies that each location maintains a given amount of the item which is necessary to process the next step in the system. Many retailers and manufacturers even use real-time systems to handle this very functionality. This system was developed to ensure consistency in the application of the approach by providing a group of alternatives to the set of systems seen thus far. In this particular system the number of simple services greatly outnumbers the composite services. In fact, the ration between them is two-to-one in favor of simple services. It will show that in scenarios which a few composite systems use a large number of simple services can benefit from regression test selection.

Lastly, the supply chain management system is also used in a variety of case studies related to Web services, including two IBM Redbooks [60, 61] as well as the system designed to showcase the WS-I (Web Services Interoperability) standard [62, 63]. Supply chain management is a very real need for business today and is a fairly hot topic in the business world. Since one of the key tenets of Web services is their ability to model real world business services, they are well-designed to handle a business's supply chain management needs. The supply chain management system was also selected because of the popularity of supply chain management systems, their inherent complexity, and their inherent composability.

The UML class diagram corresponding to this system is shown in figure 6.9 with only the interfaces shown for clarity.

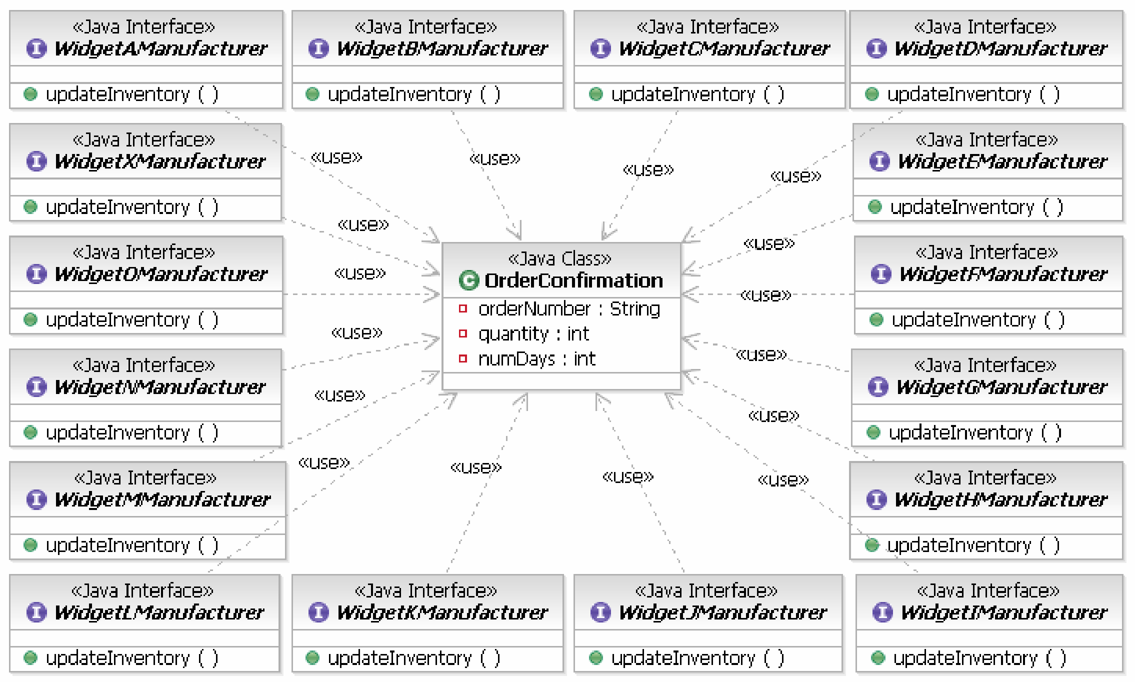


Figure 6.9: UML class diagram of supply chain management system

This system consists of sixteen services, each of which is said to manufacture some items each represented by letters of the alphabet, and the goal of each manufacturer is to update their inventory on demand to ensure that their plant operates smoothly without interruption. This system is indicative of the way modern business processes interoperate considering that all supply chains regardless of their length require management at each location. They require management at each location simply because of the dependence on one another. For instance, suppose B and C are manufactured from raw materials, but A is manufactured using B and C as ingredients. The goal is to ensure that the production speed of B and C do not affect the production speed of A, which is done by enforcing rules regarding the levels of inventory of B and C at A. This is necessary at all levels of production regardless of what is being produced.

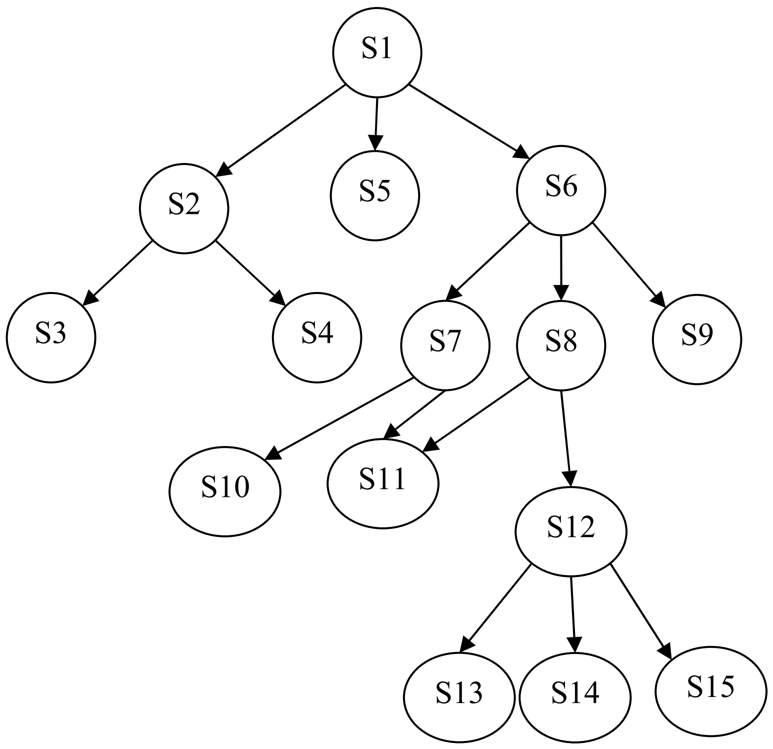


Figure 6.10: Call graph of supply chain management system

Figure 6.10 shows the call graph of the supply chain management system. The labeling of the call graph will be discussed along with the statistics of each individual service. The control-flow graphs for each service will be described, but the diagrams of the control-flow graphs will not be part of the discussion because the control-flow graphs of the majority of the services are too large to show directly. In this system, the granularities of the control-flow graphs are all at the block level. The following table lists the paths, nodes, test cases, and granularity for each of the services along with a total for all of the services.

Service	Nodes	Paths	Test Cases	Manufactures
S1	151	97	2910	X
S2	23	14	420	A
S3	12	7	210	B
S4	11	7	210	C
S5	17	11	330	D
S6	109	72	2160	E
S7	33	22	660	F
S8	57	39	1170	G
S9	13	9	270	H
S10	13	12	360	I
S11	11	6	180	J
S12	39	29	870	K
S13	9	5	150	L
S14	13	15	450	M
S15	15	8	240	N
Totals			10590	

Table 6.3: Supply chain management system totals

6.6 Supply Chain Management System (revisited)

In this section, the approach will be applied to a group of Web services which form another supply chain management system. The difference between this system and the previous system is that the first system had a few composite services being composed of a large number of simple services, and this system has a large number of composite services being composed of a few simple services. In fact, the ratio between them is three-to-one in favor of simple services.

Very much like the previous supply chain management system, this system was selected for its inherent complexity, inherent composition, and the popularity of supply chain management systems in the business world. Additionally, this system was developed to provide contrast to the previous system to determine if the approach is impacted by the ratio of simple services to composite services. This experimental system shows that systems with such a ratio can benefit from the approach.

The UML class diagram corresponding to this system is shown in figure 6.11 with only the interfaces shown for clarity.

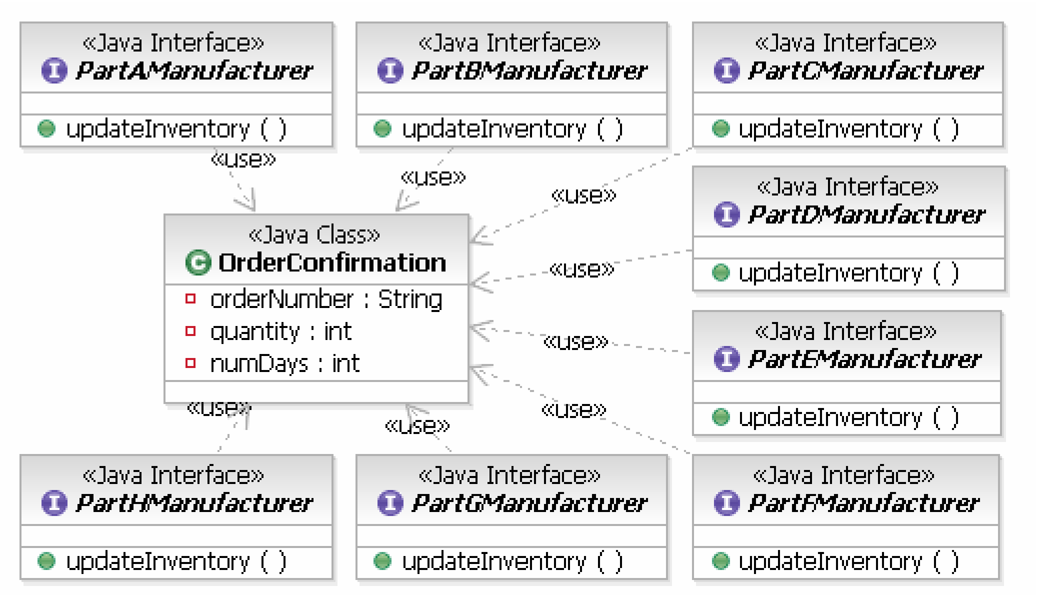


Figure 6.11: UML class diagram of second supply chain management system

Figure 6.12 shows the call graph of the second supply chain management system. The labeling of the call graph will be discussed along with the statistics of each individual service.

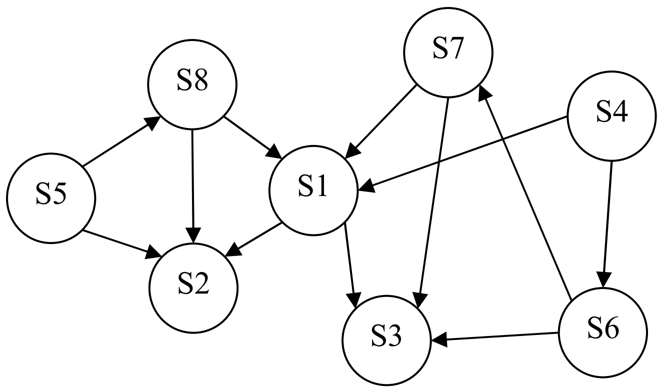


Figure 6.12: Call graph of second supply chain management system

The terminal control-flow graphs for each service will be described, but the diagrams of the control-flow graphs will not be part of the discussion because they are too large to show directly. In this system, the granularities of the control-flow graphs are all at the block level. The following table lists the paths, nodes, test cases, and granularity for each of the services along with a total for all of the services.

Service	Nodes	Paths	Test Cases	Manufactures
S1	38	23	690	A
S2	15	8	240	B
S3	13	9	270	C
S4	130	78	2340	D
S5	87	50	1500	E
S6	84	51	1530	F
S7	60	36	1080	G
S8	62	36	1080	H
Totals			8730	

Table 6.4: Supply chain management system totals

6.7 Results of Empirical Study

In this section, the results of the empirical study will be presented and discussed. Each of the experiments was run one hundred times, meaning that a random node was selected, then modified, the process performed, and finally the cost was recorded one hundred times. The purchase order system, the loan application system, and the loan brokerage system will be briefly discussed first, and then the two supply chain management systems will be briefly discussed, and this section will then conclude.

The results will be shown using histograms to help illuminate the results. For each and every experiment, the results are a percentage of performing the approach (including selecting test cases and executing the selected test cases everywhere necessary) to not performing the approach (executing all test cases). The histogram will show the distribution of these experiments across the percentages.

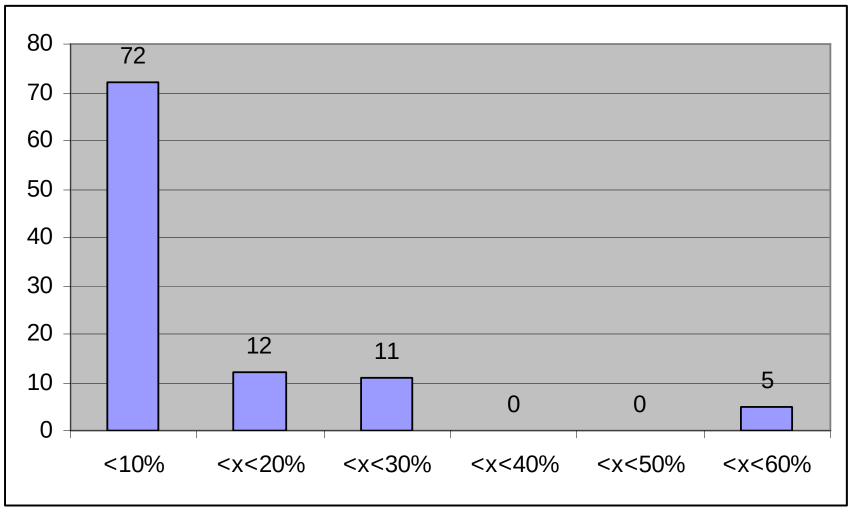


Figure 6.13: Results of purchase order system

For example, the graph in Figure 6.13 shows the results of the empirical study for the purchase order system. Along the y-axis is the number of experiments which fall into each of the categories listed along the x-axis. Each of these categories is a range of percentages, which for each experiment is the percent cost of performing the approach versus not performing the approach.

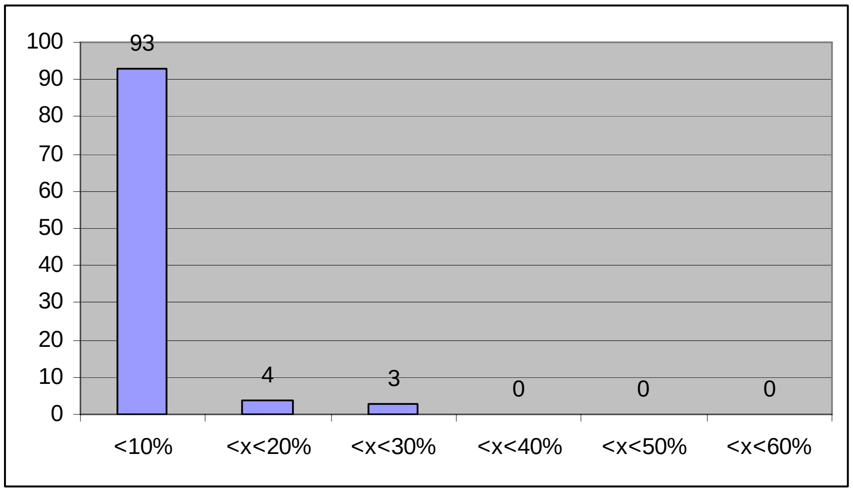


Figure 6.14: Results of loan application system

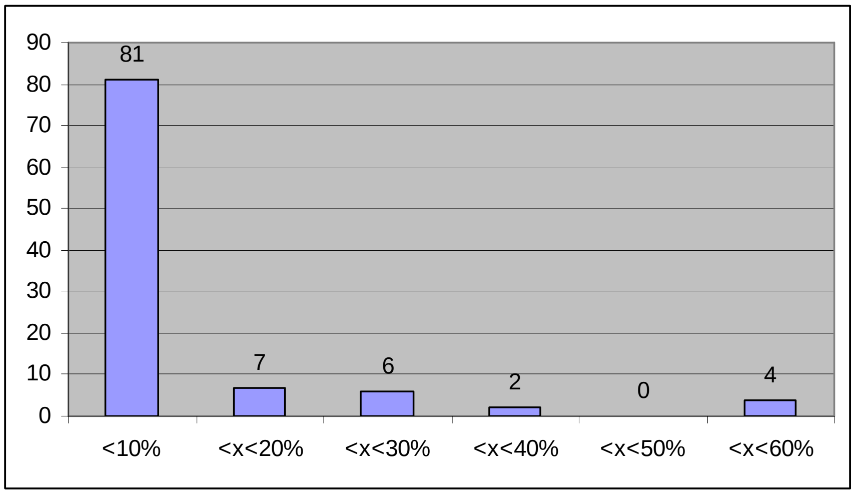


Figure 6.15 Results of loan brokerage system

The purchase order system and the loan brokerage system perform much worse than the loan application system. The worst result the loan application experiment reported was around 30% which is exceptional since it implies that performing the approach will save 70% of the cost associated with retesting even in the worst case. Even though the other two systems did not perform as well, each of them having worst results higher than 50%, the systems did show a high cost reduction potential since the worst case still saves 50% of the costs associated with retesting.

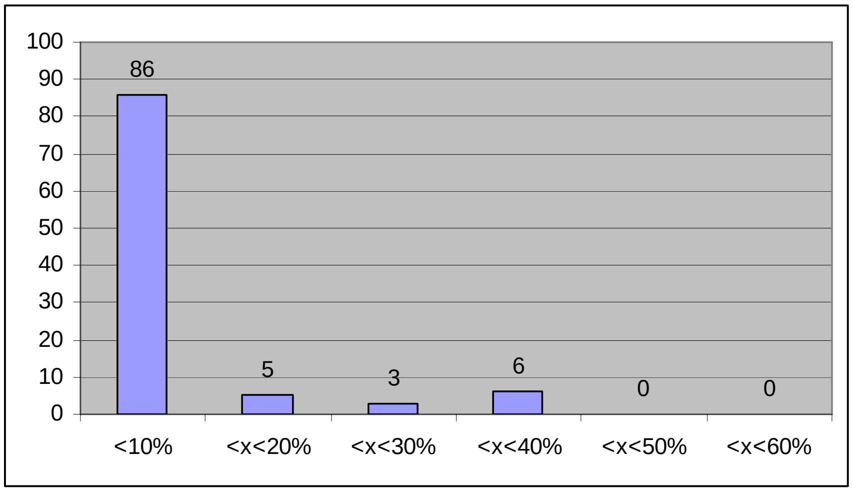


Figure 6.16: Results of supply chain management system 1

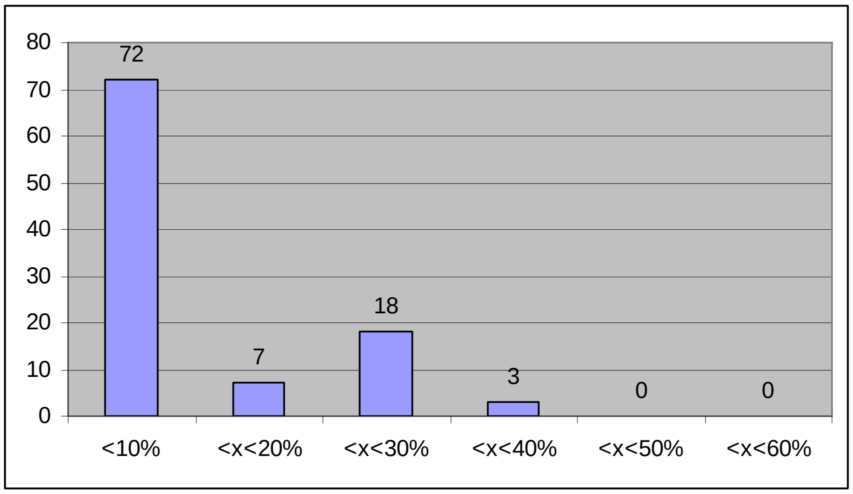


Figure 6.17: Results of supply chain management system 2

The two supply chain management systems perform very well overall with very little difference in their performance. The second system has a slightly faster growth rate and this is largely due to the slightly higher likelihood of the modification existing in a simple service causing the entire system to need testing. This growth rate difference is evidenced by the large number of experiments which cost between 20% and 30% in the second experiment, but virtually none in the first. However, both of the results indicate that their worst results are less than 40% which implies that the approach will save approximately 60% of the costs associated with retesting.

System	Worst	Mean	Median		System	Worst	Mean	Median
POS	55%	12%	5%		POS	45%	88%	95%
LAS	30%	3%	1%		LAS	70%	97%	99%
LBS	60%	9%	4%		LBS	40%	81%	96%
SCMS1	36%	6%	2%		SCMS1	64%	94%	98%
SCMS2	35%	9%	4%		SCMS2	65%	81%	96%

Table 6.5 Aggregate totals for all systems

Table 6.5 lists the worst, mean, and median cost percentages for each of the five systems along with their associated savings. The percentage of cost is listed on the left and the cost savings is listed on the right. In the table, it is important to note that the not one of the median costs is higher than 5% and not one of the mean costs is higher than

12%. Additionally, note that all systems show a positive savings, which implies that for each of the selected systems measurable cost savings was achieved. The ultimate goal of this empirical study is not to prove that the approach is beneficial to all systems, but rather that the approach is beneficial to some systems, and the results do indicate this for the developed systems.

Chapter 7: Conclusion and Future Work

Assuring the quality of Web services has become increasingly more important. Organizations which depend on Web services to fulfill their business process needs must verify that those needs are being met even as the business processes evolve especially for mission critical systems such as those which directly involve customers. Developers seeking to ensure that these complex systems continue to operate with a high level of confidence must employ techniques such as regression testing. As the system evolves and changes, more comprehensive services and the desire for higher levels of confidence require more test cases, which directly increase the cost of performing the regression testing process. Therefore, regression test selection techniques will become increasingly important to any enterprise seeking to ensure that their services remain of the highest quality. For mission critical systems, safe regression test selection techniques will also become more important because they reduce the costs without reducing the quality of the regression testing in terms of finding faults.

As described earlier, there were no existing solutions for performing regression test selection which fit the criteria set forth in the introduction in section 1.2. Therefore, a framework was developed to perform regression test selection and regression testing for the verification of Web services based on the proposed approach which is safe, distributed, automated, end-to-end, and handles the composability and interoperability aspects of Web services. A unique accomplishment of this approach is that the participants retain control of the information being exchanged. The approach recognizes that security and ownership protection are major concerns of the participating service providers who wish to protect their intellectual property. The shielding of the source code and allowing the provider to determine the granularity of the control-flow graph they share ensure the highest level of participation of service providers because the two methods provide security for the intellectual property of the service providers.

Another major accomplishment is that the issues related to concurrent modifications were discovered and solved for the first time. The approach recognizes three different types of issues, namely coverage conflict, test inconsistency, and communication issues. The approach provides a solution to these issues in the form of a set of algorithms for the agents to follow which as proven in section 5.4.3 ensures that the agents will ensure that any and all modifications will be tested consistently once the system reaches a stable-state.

Lastly in order to show that the approach to perform regression test selection is both feasible and beneficial, an empirical study was performed. In that empirical study, not one of the five systems for any of the random tests incurred a penalty for performing the regression test selection and then executing the selected tests as compared to not performing the selection and executing all tests. This demonstrated that for each of the selected systems measurable cost savings was achieved and since the goal was to prove that the approach is beneficial to some systems, the presented regression test selection approach is both feasible and can reduce the cost of regression testing.

The future avenues to extend this subject entail looking at the limitations of this work such as static composition and WSDL modifications. The approach was limited to static compositions of Web services because of test case determinism. Each and every test case must return a given output for a set of inputs every time it is run, which is the notion of test case repeatability. Additionally, in order to test a specific piece of code, a specific test case must cover that specific piece of code always. If dynamic composition were to take place, there is no guarantee that the test case would cover the correct piece of code or return the correct result. It may be possible to overcome this limitation by augmenting the framework with a UDDI controller.

WSDL modifications were ignored in this particular work, but they cannot be ignored in an industry setting so the framework must be augmented with a WSDL modification detection and notification system. If the WSDL does change, the notion that interface modifications being outside of the scope of regression testing still applies but the developers will be notified that a modification occurred and action is necessary.

Other avenues for extending this work are developing other algorithms to ensure fault locatability and extending this work to data-flow based regression test selection. Developing algorithms to ensure fault locatability provides more information, such as which modification caused which faults, but reduces the amount of concurrency in the system. This may be ideal in situations in which change-control databases must be kept updated with the system. In fact, there may be a way to integrate the process of updating the change-control database into the framework so that it always happens automatically. Extending this work to data-flow based regression test selection is a natural extension of control-flow based regression test selection techniques. One of the barriers to implementing a data-flow approach would be the amount of information about the implementation that such an approach requires.

References

[1] Rothermel, G., and Harrold, M. J., “A Safe, Efficient Regression Test Selection Technique”, ACM Transactions on Software Engineering Methodology, vol. 6, no. 2, pp. 173-210, Apr. 1997.

[2] Rothermel, G., and Harrold, M. J., “Analyzing Regression Test Selection Techniques”, IEEE Transactions on Software Engineering, vol.22, no.8, pp.529-551, Aug. 1996.

[3] Kreger, H., et al, Web Services Conceptual Architecture: WSCA 1.0, http://www-306.ibm.com/software/solutions/webservices/pdf/WSCA.pdf, May 2001.

[4] Bray, T., et al, Extensible Markup Language (XML) 1.0, W3C Recommendation, http://www.w3.org/TR/2004/REC-xml-20040204/, Feb. 2004.

[5] Gudgin, M., et al, SOAP Version 1.2 Part 1: Messaging FrameworkW3C Recommendation, http://www.w3.org/TR/soap12-part1, Feb. 2004.

[6] Gudgin, M., et al, Web Services Description Language (WSDL) Version 2.0 Part 2: Predefined Extensions, http://www.w3.org/TR/wsdl20-extensions/, Oct. 2004.

[7] Clement, L., et al, UDDI Version 3.0.2, UDDI Spec, http://www.oasis-open.org/committees/uddi-spec/doc/spec/v302.htm, Oct. 2004.

[8] Pfleeger, S., Software Engineering: Theory and Practice, Second Edition, Prentice Hall, 2001.

[9] DeMillo, R., et al, Software Testing and Evaluation, Benjamin/Cummings, 1987.

[10] Tsai, W. T., et al, “Scenario-based Modeling and its Applications”, Proceedings of the 7th International Workshop on Object-Oriented Real-Time Dependable Systems, (WORDS 2002). pp.253-260, San Diego, CA, Jan. 2002.

[11] Fischer, K.F., "A Test Case Selection Method for the Validation of Software Maintenance Modifications", Proceedings of Computer and Software Applications Conference, (COMPSAC 1977), pp. 421-426, Chicago, IL, Nov. 1977.

[12] Crowder, H., Johnson, E.L., and Padberg, M., "Solving Large-Scale Zero-One Linear Programming Problems", Operations Research, vol. 31, no. 5, pp. 803-834, Sept. 1983.

[13] Leung, H. K. N., and White, L. J., “A Study of Integration Testing and Software Regression at the Integration Level”, Proceedings of the International Conference of Software Maintenance, (ICSM 1990), pp. 290-300, San Diego, CA, Nov. 1990.

[14] Laski, J., and Szermer, W., "Identification of Program Modifications and Its Applications in Software Maintenance", Proceedings of the Conference on Software Maintenance, (ICSM 1992), pp. 282-290, Orlando, FL, Nov. 1992

[15] Chen, Y.F., Rosenblum, D.S., and Vo, K.P., "TestTube: A System for Selective Regression Testing", Proceedings of the International Conference on Software Engineering, (ICSE 1994), pp. 211-222, Sorrento, Italy, May 1994.

[16] Harrold, M. J., et al, “Regression Test Selection for Java Software”, Proceedings of ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications, (OOPSLA 2001), pp 312-326, Tampa Bay, FL, Oct. 2001.

[17] Offutt, J., and Xu, W., “Generating Test Cases for Web Services Using Data Perturbation”, Workshop on Testing, Analysis and Verification of Web Services, SIGSOFT Software Engineering Notes, vol 29. Issue 5, pp. 1-10, Sept. 2004
[18] Siblini, R., Mansour, N., "Testing Web Services", Proceedings of the ACS/IEEE Conference on Computer Systems and Applications, pp 135-142, Cairo, Egypt, Jan. 2005.
[19] DeMilli, R.A., and Offutt, A.J., "Constraint-Based Automatic Test Data Generation", IEEE Transactions on Software Engineering, vol. 17, no. 9, pp. 900-910, Sept. 1991.
[20] Martin, E., Basu, S., Xie, T., “Automated Robustness Testing of Web Services”, The International Workshop on SOA and Web services Best Practices, Portland, OR, Oct. 2006.
[21] Fu, C., Ryder, B.G., Milanova, A., and Wonnacott, D., “Testing of Java Web Services for Robustness”, Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis, (ISSTA 2004), pp. 23-34, Boston, MA, Jul. 2004.
[22] Stepien, B., Schieferdecker, I., “Automated Testing of XML/SOAP-based Web Services”, Proceedings of Fachkonferenz der Gesellschaft f¨ur Informatik Fachgruppe KiVS, Leipzig, Germany, Feb. 2003.
[23] TTCN-3, http://www.ttcn-3.org/
[24] Xiong P., Probert R., Stepien, B., “An Efficient Formal Testing Approach for Web Service with TTCN-3”, Proceedings of the International Conference on Software, Telecommunications and Computer Networks, Split, Marina Frapa (Croatia), Sept. 2005.
[25] Schieferdecker, I., Ru Dai, Z., Grabowski, J., Rennoch, A., “The UML 2.0 Testing Profile and Its Relation to TTCN-3”, Proceedings of The IFIP International Conference on Testing of Communicating Systems, pp. 79-94, Sophia Antipolis, France, May 2003.
[26] Dustdar, S., Haslinger S., “Testing of Service Oriented Architectures – A Practical Approach”, Proceedings of Net.ObjectDays, Springer LNCS, Erfurt, Germany, Sept. 2004.
[27] Bruno, M., et al, “Using Test Cases as Contract to Ensure Service Compliance across Releases”, Proceedings of International Conference of Service Oriented Computing, (LNCS 3826), pp. 87-100, Amsterdam, Netherlands, Dec. 2005.
[28] Heckel, R., Lohmann, M., "Towards Contract-based Testing of Web Services", Electronic Notes in Theoretical Computer Science, Vol. 116, pp. 145-156, 2005.
[29] Briand, L.C., Labiche, Y., Soccar, G., "Automating Impact Analysis and Regression Test Selection Based on UML Designs", Proceedings of the International Conference on Software Maintenance, (ICSM 2002), pp. 252-261, Montreal, Canada, Oct. 2002.
[30] Briand, L.C., Labiche, Y., O'Sullivan, L., "Impact Analysis and Change Management of UML Models", Proceedings of International Conference on Software Maintenance, (ICSM 2003), pp. 256-265, Amsterdam, The Netherlands, Sept. 2003.

[31] Agrawal, H. and Horgan, J.R., “Dynamic Program Slicing”, Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 246-256, White Plains, New York, Jun. 1990.
[32] Hartmann, J., Imoberdorf, C., and Meisinger, M., “UML-Based Integration Testing”, Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis, (ISSTA 2000), pp. 60-70, Portland, Oregon, Aug. 2000
[33] Tsai, W.T., Chen, Y., Paul, R., Liao, N., Huang, H., “Cooperative and Group Testing in Verification of Dynamic Composite Web Services”, Proceedings of the International Computer and Software Applications Conference, (COMPSAC 2004), pp. 170-173, Hong Kong, China, Sept. 2004
[34] Tsai, W.T., Paul, R., Wang, Y., Fan, C., Wang, D., "Extending WSDL to Facilitate Web Services Testing”, Proceedings of the International Symposium on High Assurance Systems Engineering, (HASE 2002), pp. 171-172, Tokyo, Japan, 2002.
[35] Tsai, W. T., Bai, X., Paul, R., Shao, W., Agarwal, V., "End-To-End Integration Testing Design", Proceedings of the International Computer Software and Applications Conference, (COMPSAC 2001), pp. 166-171, Chicago, IL., Oct. 2001.
[36] Tsai, W.T., Paul, R., Song, W., Cao, Z., "Coyote: An XML-Based Framework for Web Services Testing", Proceedings of IEEE International Symposium on High Assurance Systems Engineering, (HASE 2002), pp 173-174, Tokyo, Japan, Oct. 2002.
[37] Paul, R., Yu, L., Tsai, W. T., Bai, X., "Scenario-Based Functional Regression Testing", Proceedings of the International Computer Software and Applications Conference, (COMPSAC 2001), pp. 496-501, Chicago, IL, Oct. 2001.
[38] Tsai, W.T., Yu, L., Saimi, A., Paul, R., “Scenario-based object-oriented test frameworks for testing distributed systems”, Proceedings of the IEEE Workshop of Future Trends of Distributed Computing Systems, pp. 288-294, San Juan, Puerto Rico, May 2003.
[39] Tsai, W. T., Paul, R., Yu, L., Saimi, A., Cao, Z., “Scenario-Based Web Service Testing with Distributed Agents”, IEICE Transactions on Information and Systems, v.E86-D, no.10, pp.2130-2144,2003.
[40] Tsai, W.T., Paul, R., Cao, Z., Yu, L., Saimi, A., "Verification of Web services Using an Enhanced UDDI Server", Proceedings of the International Workshop on Object-Oriented Real-Time Dependable Systems, (WORDS 2003), pp. 131-138, Guadalajara, Mexico, Jan. 2003.
[41] Tsai, W.T., Wei, X., Chen, Y., Xiao, B., Paul, R., Huang, H., “Developing and Assuring Trustworthy Web Services”, Proceedings of the International Symposium of Autonomous Decentralized Systems, pp. 43-50, Chengdu, China, Apr. 2005.
[42] Tsai, W.T., Wei, X., Chen, Y., Paul, R., “A Robust Testing Framework for Verifying Web Services by Completeness and Consistency Analysis”, Proceedings of IEEE International Workshop on Service-Oriented System Engineering, (WORDS 2005), pp. 151-158, Beijing, China, Oct. 2005.
[43] OWL-S, http://www.daml.org/services/owl-s/

[44] Huang H., Tsai W. T., Paul R., Chen Y., "Automated Model Checking and Testing for Composite Web Services", Proceedings of International Symposium on Object-Oriented Real-Time Distributed Computing, pp. 300-307, Seattle, WA, 2005.
[45] Zheng, J., Robinson, B., Williams, L., Smiley, K., “An Initial Study of a Lightweight Process for Change Identification and Regression Test Selection When Source Code is Not Available”, Proceedings of the International Symposium on Software Reliability Engineering, (ISSRE 2005), pp. 225-234, Washington, D.C., Nov. 2005.
[46] Tarhini, A.; Fouchal, H.; Mansour, N., “Regression Testing Web Services-based Applications”, Proceedings of the IEEE International Conference on Computer Systems and Applications, (COMPSAC 2006), pp. 163- 170, Dubai, United Arab Emirates, Mar. 2006.
[47] Benedusi, P., Cimitile, A., and De Carlini, U., "Post-Maintenance Testing Based on Path Change Analysis", Proceedings of the Conference on Software Maintenance, (ICSM 1988), pp. 352-361, Phoenix, Arizona, Oct. 1988.
[48] Orso, A., Harrold, M.J., Rosenblum, D., Rothermel, G., Soffa, M.L., Do, H., “Using Component Metacontent to Support the Regression Testing of Component-based Software”, Proceedings of the IEEE International Conference on Software Maintenance, (ICSM 2001), pp. 716-725, Florence, Italy, Nov. 2001.
[49] Graham, S., et al, Building Web Services with Java: Making Sense of XML, SOAP, WSDL, and UDDI, Second Edition, Sams, Jun. 2004.
[50] Monson-Haefel, R., J2EE Web Services, First Edition, Addison-Wesley Professional, Oct. 2003.
[51] Ballinger, K., .NET Web Services: Architecture and Implementation with .NET, Addison-Wesley Professional, Feb. 2003.
[52] Tyagi, S., “Realizing Strategies for Document-Based Web Services With JAX-WS 2.0: Part 3 in a Series”, http://java.sun.com/developer/technicalArticles/xml/jaxrpcpatterns3/, Dec. 2005.
[53] Tyagi, S., “RESTful Web Services”, http://java.sun.com/developer/technicalArticles/WebServices/ restful/, Aug. 2006.
[54] Hutchinson, B., et al, “SOA programming model for implementing Web services, Part 4: An introduction to the IBM Enterprise Service Bus”, http://www-128.ibm.com/developerworks/library/ws-soa-progmodel4/, Jul. 2005.
[55] Chappell, D., et al., Java Web Services, First Edition, O'Reilly Media, Inc., Mar. 2002.
[56] Hohpe, G., Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions, Addison-Wesley Professional, Oct. 2003.
[57] McGovern J., et al., Java Web Services Architecture, Morgan Kaufmann, Apr. 2003.
[58] Fou, J., Web Services and the Banking Industry, http://webservicesarchitect.com /content/articles/fou03.asp, Feb. 2002.
[59] Balani, N., Model and build ESB SOA frameworks, http://www-128.ibm.com/ developerworks/web/library/wa-soaesb/, Mar. 2005.
[60] Endrei, M., et al, Patterns: Service-Oriented Architecture and Web Services, IBM Redbook, http://www.redbooks.ibm.com/redbooks/pdfs/sg246303.pdf, Apr. 2004.

[61] Endrei, M., et al, Patterns: Direct Connections for Intra- and Inter-enterprise, IBM Redbook, http://www.redbooks.ibm.com/redbooks/pdfs/sg246933.pdf, Feb. 2004.
[62] Ballinger, K., et al, Web Services Interoperability (WS-I) Basic Profile 1.3, http://www.ws-i.org/Profiles/BasicProfile-1.2.html, Mar. 2007.
[63] Glover, M., et al., Web Services Interoperability (WS-I) Sample Apps, http://www.ws-i.org/deliverables/workinggroup.aspx?wg=sampleapps, Jun. 2004.

Vita

Michael Edward Ruth was born in New Orleans, Louisiana. He received his B.S. from the University of New Orleans in December of 2002. In July 2004, he was awarded the Crescent City Doctoral Scholarship and began working as a Research Assistant under Dr. Shengru Tu. In April 2005, he published a paper at the International Computer Software and Application Conference. He received his M.S. in Computer Science from the University of New Orleans in May 2005. Since then, he has published four other conference papers and two journal articles have been accepted for publication. His research interests include Distributed Systems, Web Applications, Service-Oriented Architecture including Web Services, and Software Engineering.