12-17-2004

Towards a Regression Test Selection Technique for Message-Based Software Integration

Sriram Kuchimanchi
University of New Orleans

Recommended Citation

Kuchimanchi, Sriram, "Towards a Regression Test Selection Technique for Message-Based Software Integration" (2004). *University of New Orleans Theses and Dissertations*. 189.
https://scholarworks.uno.edu/td/189

TOWARDS A REGRESSION TEST SELECTION TECHNIQUE FOR MESSAGE-BASED SOFTWARE INTEGRATION

A THESIS

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

by

Sriram Kuchimanchi

B.E Bangalore University, 1999
M.S University of New Orleans 2004

December 2004

Acknowledgements

When I was given an opportunity to work on Regression Testing and test case selection, I was initially excited about entering an unknown area. Over the past one year, I went through ups and downs. This area being one of those grey areas, research in this has been more academic and theoretical rather than practical. To make things further difficult, hardly any assistance was available for object oriented languages especially Java. PeopleSoft posed another angle of difficulties.

Through out this whole exercise, though, Dr.Shengru Tu has been extremely helpful and supportive. If I had hit a road block and needed some relief, he would always have the supportive smile. Not just this, but, towards the end, when I had to type in this entire document and get ready for the defense, he helped out with many late nights and night-outs. I can't thank him enough for all this. His strong fundamental knowledge about distributed programming and theories like RPC was very helpful, not to mention his experience in writing technical papers.

I would also like to thank Eric Normand for being a very able team partner through the difficult time with PeopleSoft and Web Services. We forged a very good partnership throughout this thesis. Together we worked on 2 technical papers under Dr. Tu's guidance and one of them making it to the IEEE symposium was one of the highlights of this effort.

I would like to thank the committee members; Dr.Chiu and Dr.DePano for allocating time for me and supporting me through this research.

Then, I would like to thank all my friends who have been very supportive and helpful when I had technical queries to bug them. Some provided with the required shoulder to rest when needed.

I would like to thank my parents and family for their support and blessings, without who I would not have been here today.

Table of Contents

List of Figures

Abstract

Regression testing is essential to ensure software quality. Regression Test-case selection is another process wherein, the testers would like to ensure that test-cases which are obsolete due to the changes in the system should not be considered for further testing. This is the Regression Test-case Selection problem. Although existing research has addressed many related problems, most of the existing regression test-case selection techniques cater to procedural systems. Being academic, they lack the scalability and detail to cater to multi-tier applications. Such techniques can be employed for procedural systems, usually mathematical applications.

Enterprise applications have become complex and distributed leading to component-based architectures. Thus, inter-process communication has become a very important activity of any such system. Messaging is the most widely employed inter-module interaction mechanism. Today's systems, being heavily internet dependent, are Web-Services based which utilize XML for messaging.

We propose an RTS technique which is specifically targeted at enterprise applications.

Chapter 1 – INTRODUCTION

As a software system ages, the cost of maintaining the software dominates the overall cost of developing the software. The cost of maintenance is increasing. In the 1990s, the estimated expenditure exceeds 70% of total software costs [2, 3]. As software reuse has been more and more emphasized in practice, the life cycle of software components has been constantly prolonged. While reuse helps reduce overall development costs, it tends to increase the percentage of software maintenance costs.

A large percentage of maintenance expense is devoted to testing especially regression testing. Testing is widely used to build confidence in the correctness of software and to increase software reliability. Testing is a validation process that determines the conformance of the software's implementation to its specification. It is an important phase in software development life cycle. On one hand, the modern business processes have become more and more dynamic and adaptive, which requires more and more changes to their software. On the other hand, software systems and software components' life cycles are prolonged for economically reasons. Both phenomena contribute more change requests to existing software systems. Upon each set of changes, a thorough test must be carried out for the modified software system again. Thorough tests for large systems are very expensive.

Regression testing is a testing process that is used to determine if a modified program still meets its specifications or if new errors have been introduced. By regression, we mean to use the test cases for the original system again. Regression testing saves the cost for test case generation by reusing the existing test cases. Even so, the cost to rerun the existing test cases for large systems is still very high.

The most important improvement in the regression testing process is regression testing selection (RTS) approach, which helps reduce the cost of testing by algorithmically select a subset of test cases from the original test case set. The smaller the chosen subset is, the lower the cost of the testing will be. At the same time, the chosen subset must establish the equal correctness confidence as the original test case set. Regression testing is an important type of testing. Onoma et al. reported that the secret in delivering quality software is good regression testing [1].

Although a number of well-known methods for RTS have been extensively studied (discussed with details in Chapters 3 and 4), the RTS techniques are inadequate for the software development in the Internet era. Gone are the days when software was written on the fly. Reusing software, especially, *integrating* existing systems (including legacy systems) using new components and bridges has been an industry norm. The most important feature of this is *communication*. Inter-process communication has thus, gained significance and has risen from mere academic research topic to an industry mandate when it comes to software development. To my knowledge, very few attempts were made at implementing an RTS technique for enterprise software integrations.

Our goal is to develop a regression test selection technique and implement corresponding tools for systems composed of enterprise software integrations. Our focus is on the integration components rather than the enterprise software products themselves. This ambitious goal requires extensive studies, research, preparation, experiments and development. The very first challenge is that experiments of this kind would require comprehensive environments including multiple enterprise software product installations. Secondly, the enterprise integrations are usually involved companies' proprietary intelligence or trade secretes; it is difficult to get the underlying systems. Thirdly, obtaining or establishing the integration only provides the possibility of experiments. RTS requires nontrivial test cases. Realizing the mounting tasks for our goal, I laid out a feasible work scope for this thesis which includes the following three objectives. More research efforts will be continued by other graduate students.

- To carry out a thorough survey on RTS;
- To establish an experiment underlying system, an integration of multiple enterprise software systems;
- To propose a framework of RTS for enterprise software integrations.

Chapter 2 - BACKGROUND

2.1 Introduction

This chapter would explain the concepts based on which this thesis work has been built. In particular, this effort is directed towards the area of regression testing and the difficulties in test case selection for message based integration in enterprise applications. In this regard, the following terms have to be explained before we delve further into subsequent chapters.

1. **Regression testing**
2. **Regression test case selection**
3. **Messaging-based systems and integration**

Before we get into the details of each, I would like to talk on some of the basics on *software testing*.

- *Reliability* is defined as the probability that a system functions according to its specification for a specified time and in a specified environment. It gives a measure of confidence in the system for the user.
- A *failure* occurs when the run-time system behavior does not match the specifications. Reliability is a statistical study of the failures.
- A *fault* is a static component of the software that causes the failure. A fault causes failure only when that part of the code is executed. So, not all faults result in failures.
- An *error* is a programmer action or omission and results in a fault.
- A *domain space* for the variable is defined as the set of all possible legal values that the variable can assume during system operation.
- A *Test Case* is defined as a single value mapping for each input of the program that enables a single execution of the software system.
- *Test Script* is a set of conditions specified on each input variable for guiding the automatic generation of test cases.
- A *Test Suite or a Test Run* is a set of test cases that are executed sequentially.

2.2 Goals of Testing

There are two main goals of testing software. On one hand, testing can be viewed as a means of achieving required quality of the system. The main aim here is to probe the software for defects and fix them. This is also called *debug testing* and is assumed to be very effective in uncovering potential faults. On the other hand, testing can also be viewed as a means of assessing the existing quality of the system and provide fault coverage measurement, which consists of studying the potential faults generated by a given test suite. This method is called *operational testing* and is proved to be effective in predicting the future reliability of the system. In debug testing, a systematic approach is followed for selecting test cases based on situations likely to produce the most number of errors. The drawback of this approach is that it may uncover failures with negligible rates of occurrence and risk, and the test effort may not worth the reliability improvement achieved. Another drawback is that it does not provide complete mathematical and technical validity of the reliability assessments.

In operational testing, a test case is selected based on the probability of its occurrence in the real operating environment. Hence, this method is likely to uncover failures with highest probability of occurrence first and provide accurate assessment of current reliability. This is done after the debugging phase, and the objective is to assert that the system is reliable and gain confidence for the final release of the system. The method is not very effective in improving the reliability of the system, and at the same time it is very difficult to generate an accurate operational profile.

2.3 Selecting the Test Cases

The input and output domain space for a complex system is typically very large and completely impractical for manual testing of each combination of the input values within the domain space. An automated test tool for generating test cases based on user specified criteria is needed to cover as much of the test domain as possible. Automated testing also allows much more tests to be run than manually testing and permits tests to be easily run over and over again. This repeating of previous tests is an important step in testing and is referred to as regression testing. Regression testing attempts to assure that software failure correction does not introduce additional failures.

2.4 Run the Test Cases

Once the test suite is selected, it must be run in the simulated environment sequentially and the results must be captured. Any failures generated also have to be captured and reported to the user. This involves integrating the automated test tool with the simulated environment.

2.5 Analyze the Test Results

This is the most important part of testing and should be done very carefully. The test results are used as a mechanism for identifying the defects in the software or the model and should be used as a mechanism to quantify the software reliability. The test results have to be analyzed to assess the quality of the software and to determine potential problem areas that require more testing and to identify portions of the input domains that have not been tested. An important consideration here is to leverage measurements against multiple independent test runs. A decision regarding the end of the test phase is to be made based on all the test runs.

With this behind us, let me now get into the details of afore mentioned topics. I shall attempt at defining and explaining these topics so that the subsequent chapters become easier to understand.

2.6 Regression Testing

Regression testing is one of the necessary maintenance tasks. It is defined as *the process of validating modified software to provide confidence that the changed parts of the software behave as intended and that the unchanged parts of the software have not been adversely affected by the modification* [Harrold]. The adverse impact of a change is often called the “ripple effect”, and is known to be a serious cause of software defects as the result of a change [Probert]. It is widely accepted that efficient and effective regression testing can reduce the frequency and cost of software maintenance.

As a software application gets updated regularly, each iteration would change the functionality and features of the system. This in turn, would have a direct effect on the test applications or the testing system. Changes in the System Under Test (SUT), is a normal process. An obvious after effect is that some of the existing test cases would become redundant. Other functionality of the latest version of the SUT could potentially go untested or unattended. Regression Testing deals with the aspect of testing a periodically changing software system.

With widespread usage of object-oriented programming techniques, more and more projects follow an *evolutionary* process model, also called an *incremental model* [McGregor]. An *increment* is a deliverable that provides some of the functionality required for the system, including models, documentation and code. Under this process model, a system is developed as a sequence of *increments*. For each increment, the development process feeds new and/or revised designs and implementations into the testing process. The testing process feeds identified failures back into the development process. The development process and the testing process form a continual feedback loop.

In the *incremental model*, successive *increments* add and sometimes revise system functionality while keeping most of the functionality of previous *increments*. *Regression testing* is typically done between *increments* to ensure that changes do not adversely affect correctly working code of the previous versions.

Reuse has been proposed as a general solution to chronic software development problems: namely, lengthy development time and high cost, unacceptably frequent failures, low maintainability, and low adaptability [Binder]. A major precondition of reuse is to ensure that the reused components match the new requirement and do not conflict with old components. An example of disasters caused by incorrect reusing of components was the explosion of Ariane 5 on June 4, 1996. The rocket was on its first voyage, after a decade of development costing $7 billion. Estimates of the total cost of the destroyed rocket and its cargo vary from a low of $350 million in a European Space Agency press release to a high of $2.5 billion reported in Florida Today Space Online.

The main reason of the explosion was that a time sequence based on the requirement of Ariane 4 was reused, but this did not match the requirement of Ariane 5 [SIAM]. Effective *regression testing* should have caught this problem before the disastrous launch.

2.6.1 Definition

The most widely used representation of the concept of regression testing in relation with software development is given below -

Let *P* be a procedure or program, let *P'* be a modified version of *P*, and let *T* be a test suite for *P*. A typical regression test proceeds as follows:

1. Select $T' \subseteq T$, a set of test cases to execute on *P'*.
2. Test *P'* with *T'*, establishing *P'*'s correctness with respect to *T'*.
3. If necessary, create *T''*, a set of new functional or structural test cases for *P'*.
4. Test *P'* with *T''*, establishing *P'*'s correctness with respect to *T''*.
5. Create *T'''*, a new test suite and test execution profile for *P'*, from *T*, *T'*, and *T''*.

This can be pictorially depicted using the following figure –

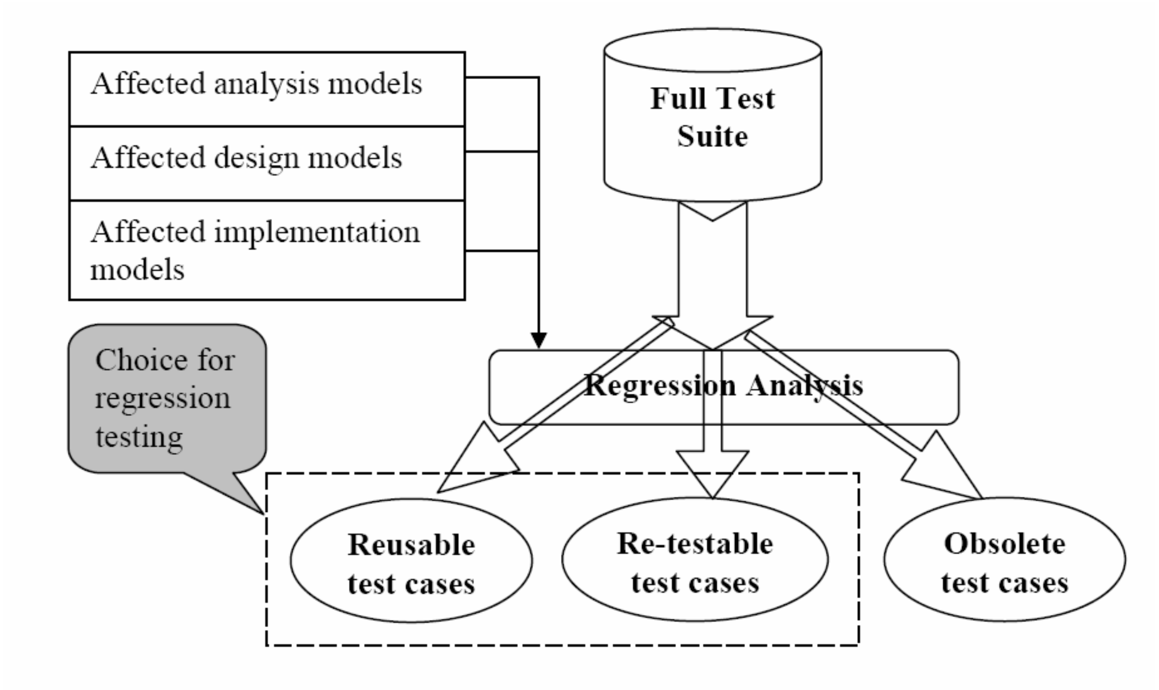

Figure 2-1: Regression Testing and Test Case Selection

Further, I would now briefly discuss various levels of regression testing. Three types of software testing are applied during the software development life cycle: *unit testing*, *integration testing*, and *system testing*. Testing at the unit, integration and system levels reveals different types of failures.

2.6.2 Unit Level

Unit testing is the process of testing each software module to ensure that **its** performance meets its specification. Most existing regression testing techniques focus on unit testing. Yau and Kishimoto [5] developed a method based on the input partition strategy. This approach divides the input domain of a module into a set of input partitions (classes) using the information from the program specification and code. The input partition P is derived by intersecting the two input partitions Ps and Pc which are generated respectively from the program specification and code [5]. The retest criterion of the testing method is to ensure that each new or changed input partition is executed by at least one test case. A cause-effect graph of the specifications is constructed and a test information table is created to help in selecting test cases. The specification partitions are the different possible combinations of input conditions from the graph [5].

The code partitions are the paths in the code that correspond to the specification partitions [5]. A test case is chosen if it exercises one of these paths. If the selected test cases do not satisfy the testing criterion, new test cases are generated. The authors demonstrated the feasibility of their method on FORTRAN programs that perform mathematical computations.

2.6.3 System Level

System testing is testing of the entire software system against the system specifications. System testing must verify that all system elements have been properly integrated and perform allocated functions. System testing can be performed without the knowledge of the software implementation at all. *TestTube* is a system developed at AT&T Bell Laboratories to perform system-level regression testing [8]. *TestTube* is an example of a selective retesting technique. It can be used with unit-level regression testing as well.

In *TestTube*, test selection is based on analysis of the coverage relationship between test cases and the system under test. It partitions a software system into basic code entities, then monitors the execution of a test case, analyzes its relationship with the system under test, and determines which subset of the code entities the test covers. For a C program, each test case is associated with function definitions, global variables, type definitions and preprocessor macro definitions that it covers or possibly covers. If a change is made to one of the entities associated with a test case, then the test case is selected. TestTube is an example of how automatic regression testing can be applied. If a program and its revision each have a database, then a program is run to determine differences in the databases and thus what has been changed.

2.7 Messaging Based Systems

Messaging is one very important communication mechanism. Messaging can be implemented in many ways using different kinds of technologies, but, here, I would like to discuss inter-process communication using remote method invocation when exchanging XML messages. My work employs Web services technology which have XML messages exchanged using underlying SOAP protocol.

In the following paragraphs, I will explain the Web services technologies. Java has been used throughout this work as the development language. Platforms for implementing and deploying the web services were many, ranging from Apache AXIS and TOMCAT to IBM Websphere.

2.7.1 Web Services

Despite the fact that the term Web Services has rapidly gained a lot of momentum, there is no single, universally adopted definition of Web Services. Several major Web services infrastructure providers have published their definitions for a Web Service. The World Wide Web Consortium (W3C) defines Web Services as below [13] –

A Web service is a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically WSDL). Other systems interact with the Web service in a manner prescribed by its description using SOAP-messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards.

We, thus derive a reasonable understanding as follows –

- A Web Service is a programmable application, accessible as a component via standard Web protocols like HTTP, XML and SOAP;
- Web Services are a combination of the powerful Remote Method Invocation clubbed with the ease of HTTP based communication.
- A Web Service is published, located, and invoked across the Web;
- A Web Service is XML-based with standards which enables simplicity, extensibility, and interoperability, programming language and platform independency;
- A Web Service works through existing proxies and firewalls.

Before we get into explaining some of the technologies which form the infrastructure, let us get a closer look at why are web services becoming so popular? The reason can be attributed to chief characteristics of Web Services, namely;

1. *Loose Coupling* - Web services are loosely coupled. They have well defined, published interfaces and can be easily accessed from remote systems over the internet. They require a simple level of coordination between systems and the underlying technology behind the web service can be changed and replaced without impacting the systems that invoke it.

   Previous attempts at distributed computing such as CORBA, DCOM, and Distributed Smalltalk are not appropriate for low overhead, ubiquitous B2B Internet processing because they require tight coupling between systems. These tightly coupled integration technologies require that the developer thoroughly understands and has control over both ends of the connection. The problems with tightly coupled integration are not unique to internet-based integration but also apply in many instances to intranet-based integration.

   This loose coupling nature of web services simplifies the integration process, lowering the cost of integration and making it easier to integrate applications than techniques used in the past.

2. *Internet Accessible* - Another issue with past integration technologies is that they were not designed to work securely over the internet. Web services make use of the existing, ubiquitous transport protocol of the internet: ***HTTP***. This is the same transport protocol that is used to deliver content over the web. Piggybacking on HTTP, web services leverage existing infrastructure and can comply with corporate firewall policies.

The principle behind the potential of web services is its architecture. Devoid of any particular standard, it provides flexibility to the implementation. Even then, interoperability is still a key requirement for large-scale adoption of the architecture. With industry support being very heavy, companies like IBM, Microsoft and Sun

Microsystems have been pushing an architecture which would have a technology stack as shown below –

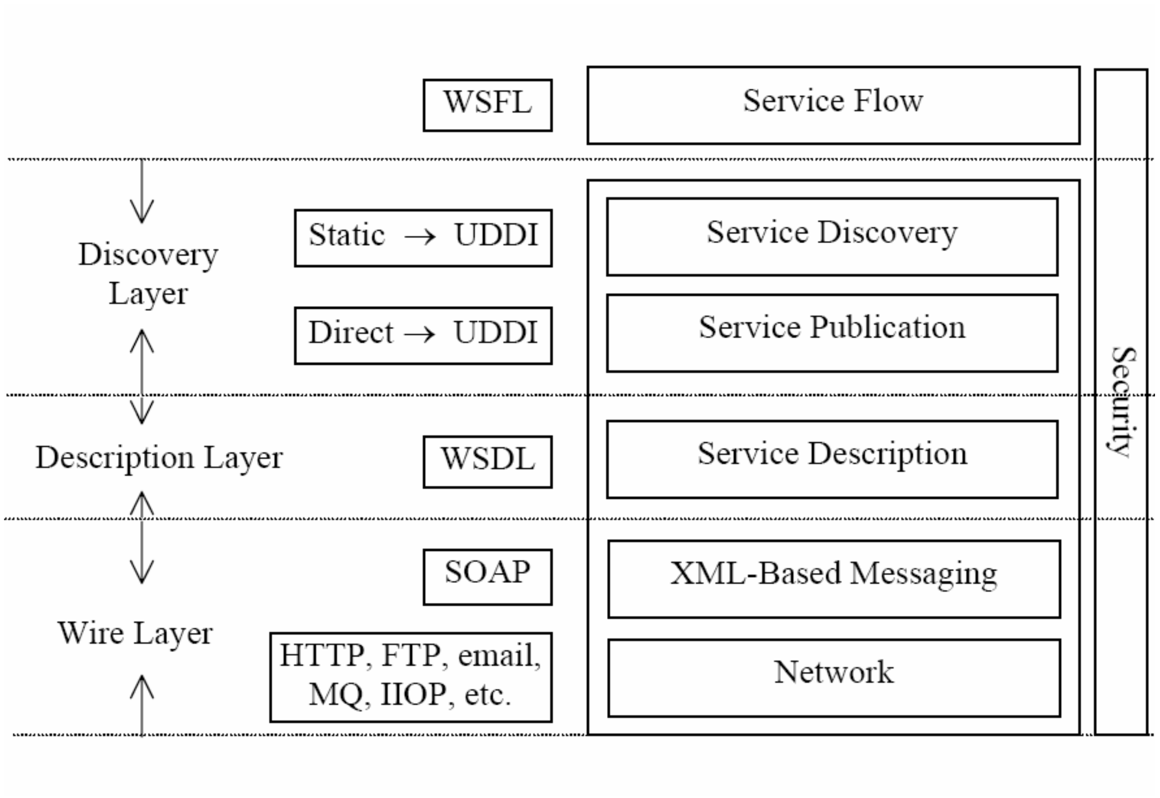

Figure 2-2: Web Services Technology Stack (IBM & Microsoft)

Before we get into the technical details, let me explain this figure.

(a) *Wire Layer* - The wire layer defines the messaging format and components between the service requestor and the service provider. The base of the wire layer is the network, which makes Web services accessible to a service requestor. Internet-available Web Services use commonly deployed network protocols. HTTP is the *de facto* standard network protocol for Internet-available Web Services. Other Internet protocols, such as SMTP and FTP, are also supported. Reliable messaging and call infrastructures, such as MQSeries, IIOP, and so on, are supported for Intranet Web Services. SOAP is chosen as the de facto XML messaging protocol for general IT Web services.

(b) *Description Layer* - The Description layer consists of Web Services description documents. XML is not only the basis of the Wire layer, it is also the basis of service description. Web Services Description Language

(WSDL) is the de facto standard for Web Services description in the IT Industry. WSDL defines the interface of a Web service and mechanisms of service interaction, which specifically include the operations supported by the Web service, the input and output of the service, the bindings to concrete network, and data encoding schemes. It is the minimum standard service description necessary to support interoperable Web Services. Additional description is needed to specify other properties of Web services, such as quality of service, service-to-service relationships, and so on. Web Services Flow Language (WSFL) is used to describe Service composition and flow.

(c) *Discovery Layer* - A Web service needs to be discovered in the first place before it can be invoked. Service discovery closely depends on service publication since a service cannot be discovered if it has not been published.

The simplest way of service invocation is a static & direct way in which the service provider sending a service description, i.e., the WSDL file(s), directly to the service requestor. Many a times this is the case when the parties know each other and it is strictly a private affair. On the other hand, in a complex scenario, a service provider publishes the service description to a local service registry or UDDI service registry which would then be the source of the WSDL file for the requestor.

In the remaining part of this chapter, I am going to summarize WSDL and SOAP.

1. *WSDL* - WSDL is an XML-based description of how to connect to a web service. It references a schema which describes the inputs and outputs of a web service and the URL to post requests to in order to invoke the web service. Simply put, it describes the technical invocation syntax of a Web service.

A complete WSDL service description provides two pieces of information: an application-level service description, or abstract interface, and the specific protocol-dependent details that users must follow to access the service at concrete service end points. This separation accounts for the fact that similar application-level service functionality is often deployed at different end points with slightly different access protocol details. Separating the description of these two aspects helps WSDL represent common functionality between seemingly different end points.

The W3C has a well defined schema for WSDL introducing several high level or major elements in the language. They are briefly explained below:

a) **PortType** – A Web Service's abstract interface definition where each child operation element defines an abstract method signature.
b) **Message** – Defines a set of parameters referred to by the method signatures or operations. A message can be further decomposed into parts.
c) **Types** – Defines the collection of all the data types used in the Web service as referenced by various message part elements.
d) **Binding** – Contains details of how the elements in an abstract interface (portType) are converted into a concrete representation in a particular combination of data formats and protocols.
e) **Port** – Expresses how a binding is deployed at a particular network endpoint.
f) **Service** – A collection of ports.

To summarize, the *portType* (with details from the message and type elements) describes the *what* of the Web Service. The *binding* element describes the *how*, and the *port and service* elements describe the *where* of the Web Service.

Many developers split their WSDL designs into two parts, each placed in a separate document. The service interface definition, containing the types, message, portType, and binding elements, appears in one file. You can then place this file, for example, on a well-known Web site (on an e-marketplace, for example) for everyone to view. Each organization that wants to implement a Web service conformant to that well-known service interface definition would describe a service implementation definition, containing the port and service elements, describing how that common, reusable, service interface definition was, in fact, implemented at the network endpoint hosted by that organization. Finally, WSDL is not a requirement for a web service. It simply makes it easier for the programmer who wants to invoke the web service to understand it.

2. *SOAP* - SOAP is the primary web services protocol. It uses XML over HTTP for messaging and RPC-style communications. It is a simple, flexible, and highly extensible XML-based messaging protocol [22]. It is implementation language or platform neutral. His includes the Web Service implementation details. Rather than defining a new transport protocol, SOAP works on existing network protocols, such as HTTP, SMTP, FTP, and so on. No surprise hence, that, since its introduction in late 1999, SOAP has become the *de facto* standard for Web services messaging and remote procedure calls.

A SOAP Message is an ordinary XML document that consists of three sections:

1. **Envelope** – The top element of the XML document representing the message and defines the content of the message. It defines the framework of what is in a message, how to process it, who should deal with it and whether it is optional or mandatory.
2. **Header** – This section is optional. It contains header information and attributes that can be set to encode and further identify the type of processing and additional features of the message.
3. **Body** – This section contains the call and response information intended for the recipient of the message. This is the message "payload".

These can be depicted using the SOAP stack diagram below –

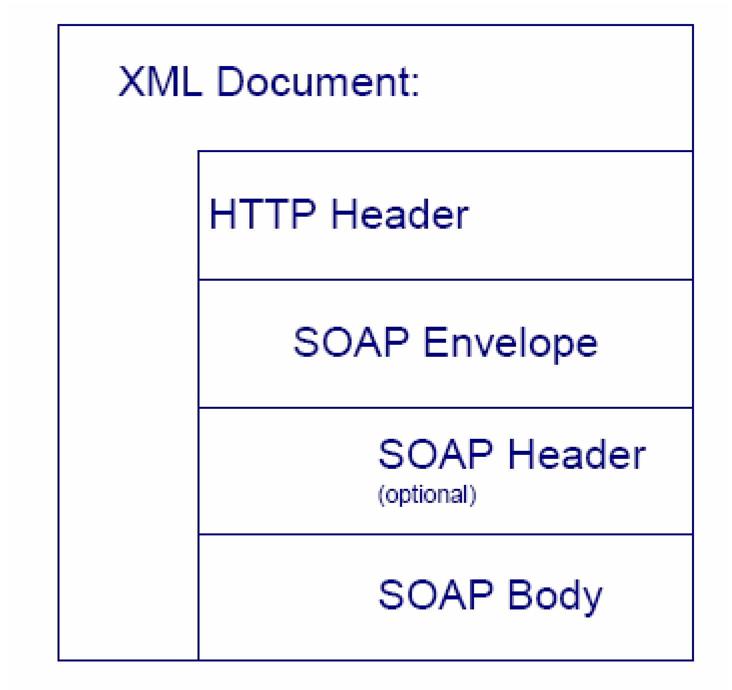

Figure 2-3: SOAP sections in an XML document

There are two types of messaging pattern using SOAP: the Remote Procedure Calls and the Conversational Message Exchanges.

Remote Procedure Calls (RPCs) with SOAP is used when there is a need to model a certain programmatic behavior, with the exchanged messages conforming to a pre-defined description of the remote call and its return. To use SOAP for RPCs, you must define an RPC protocol, including [16]:

- how typed values can be transported back and forth between the SOAP representation (XML) and the application's representation (such as a Java class for a ticket), and
- where the various RPC parts are carried (object identity, operation name, and parameters).

The Conversational Message Exchange is a request-response pattern, in which XML-based content conforming to some application-defined schema are exchanged via SOAP messages. In the simplest case, the user can only send the SOAP request to the service for processing.

SOAP is not a requirement for exposing a web service, but should be used when appropriate when implementing a web service. Many web services today expose functionality over XML/HTTP without using SOAP. SOAP implementations exist for several programming languages, including C, Java, and Perl, which automatically generate and process the SOAP messages. Assuming the messages conform to SOAP specifications, they can thus be exchanged by services implemented in different languages.

2.8 Problem Specification

In layman terms, the problem of regression test case selection can be described as follows – For a constantly changing system making corresponding changes to the test cases and subsequently running them is not easy. Some test cases which could have been originally written for a specific functionality might just become redundant and hence would have to be removed from the test suite. The problem however becomes compounded when this task would have to be performed automatically. Most types of testing jobs are/have been automated for easy and human intervention free execution.

Based on the understanding from the definition of regression testing, I stated in this chapter in section 2.6, I would like to explain the problem we would be taking up as part of this thesis.

In trying to perform the steps described in the definition of section 2.6, we identify the following problems –

- Step (1) involves the *regression test selection problem*: the problem of selecting a subset *T'* of *T* with which to test *P'*. This problem includes the sub-problem of identifying tests in *T* that are *obsolete* for *P'*. Test *t* is obsolete for program *P*' if and only if *t* specifies an input to *P*' that is invalid for *P*', or *t* specifies an invalid input-output relation for *P'*.
- Step (3) addresses the *coverage identification problem*: the problem of identifying portions of *P'* that require additional testing.
- Steps (2) and (4) address the *test suite execution problem*: the problem of efficiently executing tests and checking test results for correctness.

➢ Step (5) addresses the *test suite maintenance problem*: the problem of updating and storing test information.

Although each of these problems is significant, we restrict our attention to the regression test selection problem. We further confine ourselves to messaging based systems. Messaging based enterprise systems would in many ways manifest the changes in their components through changes in the messages exchanged. This would in turn require test cases to change and subsequent re-execution of the test cases. The challenge lies in trying to isolate and identify the modified components and identifying the test cases which actually interact with multiple components. This scenario is more commonly understood as the inter-module communication.

So, RTS technique on such systems has not been attempted much and the industry surely requires one such architecture which would cut down on the costs as well as, time required by the Software Quality Assurance team of an organization to certify an application before it can be used.

The objective of the thesis can be depicted in the diagram below –

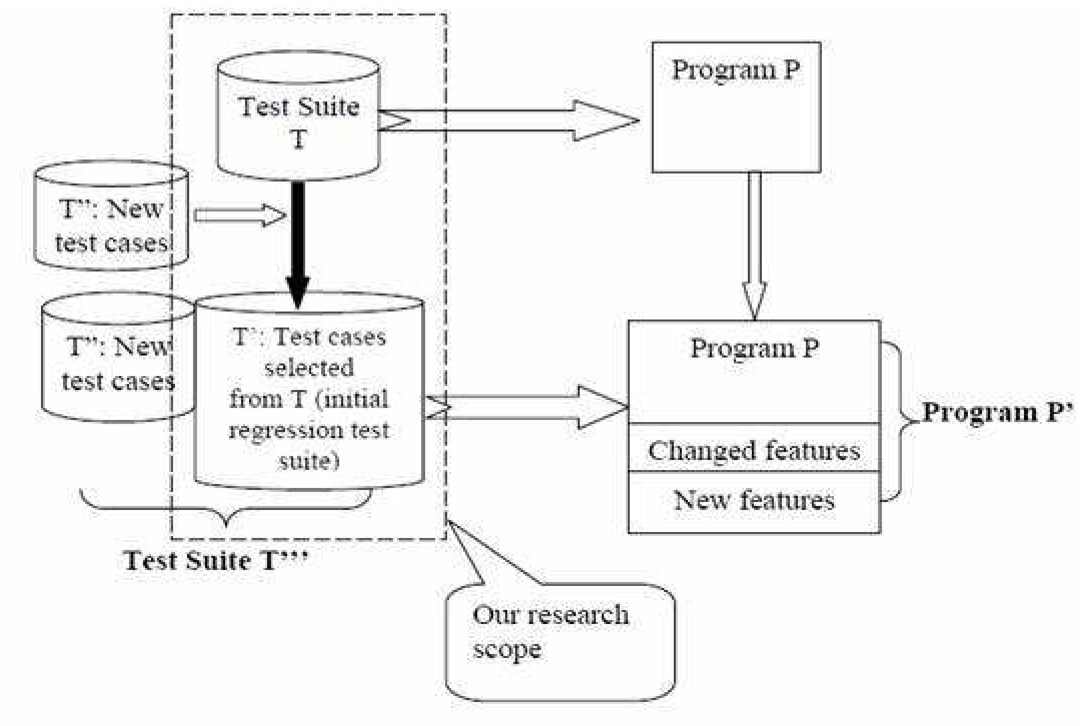

Figure 2-4: Thesis Scope Diagram

Chapter 3 - A SURVEY ON REGRESSION TEST SELECTION

In this chapter, I discuss various theories and methodologies for Regression Test Case Selection. Some of these are empirical studies with implementation used as case studies. Each of these has its own advantages and disadvantages. While this is not an attempt at rating any of them, it would throw light on the research done on RTS.

3.1 Safe RTS Techniques

Rothermel and Harrold [1996] presented a framework that is widely accepted and employed for use in comparative analyses of safe RTS techniques. *Test Tube* and *Dejavu* are two such techniques which were built keeping this framework in mind. It consists of four criteria:

- *Inclusiveness* measures the extent to which an RTS technique selects test cases from T (test suite) that are modification-revealing for the SUT (system under test). And, for the safe techniques 100% inclusiveness is required.
- *Precision* measures the extent to which a technique omits test cases from T that are not modification-revealing for the SUT and are thus unnecessary.
- *Efficiency* measures the time and space requirements of a technique.
- *Generality* measures the ability of a technique to function in a practical and sufficiently wide range of situations: in particular, a general safe technique does not require strong or difficult-to-satisfy preconditions for safety.

3.1.1 Test Tube Methodology [2]

It is a system for selective retesting that identifies which subset of a test suite must be rerun to test a new version of a system. *Test Tube* combines static and dynamic analysis to perform selective retesting of software systems. It first identifies which functions, types, variables and macros are covered by each test unit in a test suite. Each time the system under test is modified, *Test Tube* identifies which entities were changed to create the new version. Using the coverage and change information, *Test Tube* selects only those test units that cover the changed entities for testing the new version. The working of *Test Tube* relies on a premise that all value creations and manipulations in a program can be inferred from static source code analysis of the relationships among the functional and non-functional entities. This premise is valid for languages without pointer arithmetic and type coercion.

The method underlying *Test Tube* is simple. First the system under test is partitioned into basic code entities. These entities are defined in such a way that they can be easily computed from the source code and monitored during execution. The execution of each test unit is then monitored; its relationship with the system under test is analyzed, and in this way determines which subset of the code entities it covers. When the system is changed, the set of changed entities are identified and then the previously computed set of covered entities for each test unit are examined and check to see if any has changed. If none has changed, the test unit need not be rerun. If a test unit is rerun, its set of covered entities must be recomputed. Note that the notion of what constitutes a change in the system is programming language-dependent. Especially, when it comes to testing enterprise applications, the underlying architecture followed, for instance, J2EE or .NET, would make a difference when defining a change, before regression can be performed.

Test Tube differs from these previous approaches in a number of ways and has some noteworthy advantages which make it the first choice to describe here. First, *Test Tube* can be used with any chosen test generation and test suite maintenance strategy. Second, the analysis employed in *Test Tube* is performed at a granularity that makes it suitable for both unit-level and system-level testing, Third, the analysis algorithms employed in *Test Tube* are computationally inexpensive and thus scale up for retesting large systems with large numbers of test units. Fourth, the analysis that is performed in *Test Tube* produces byproducts that can be used for purposes other than selective retesting.

3.1.2 DejaVu Methodology [3]

Rothermel and Harrold developed another safe Regression Test case Selection technique, a tool named, *DejaVu*. *DejaVu* constructs control-flow graph (CFG) representations of the procedures in two versions a *P* (program), in which individual nodes are labeled by their corresponding statements. *DejaVu* assumes that a *test history* is available that records, for each test case *t* in *T* and each edge *e* in the CFG for *P*, whether *t* traversed *e*. This test history is gathered by instrumentation code inserted into the source code of the *SUT*. Construction and storage of the CFGs for *P* and the test history are preliminary phase costs. Construction of the CFGs for the other version of *P*, however, is a critical-phase cost.

DejaVu performs a simultaneous depth-first graph walk on a pair of CFGs for each procedure and its modified version in *P* and its modified version, keeping pointers ^*N* and ^*N'* to the current node in each graph.

The tool begins with the entry nodes of both the graphs; it then executes a recursive depth-first search on both CFGs. Upon visiting a pair of nodes *N* and *N'*, *DejaVu* examines each edge *e* leaving *N* to determine whether there is an equivalently labeled edge in the other CFG. If not, then the tool places *e* into a set of *dangerous edges*. If there is a corresponding edge in the second CFG, and that edge enters an already traversed portion of the graph at the same node in both CFGs, then the current recursion is terminated. Otherwise, ^*N* and ^*N'* are moved forward to point to a new pair of nodes. The tool then checks to see if the labels on the nodes pointed to by ^*N* and ^*N'* are lexically equivalent (textually equivalent after non-semantically meaningful characters such as white space and comments have been removed). If they are not, *DejaVu* places the edge it just followed into the set of dangerous edges and returns to the source of that edge, ending that trail of recursion. After *DejaVu* has determined all the dangerous edges that it can reach by crossing non-dangerous edges, it terminates. At this point, any test case *t in T* is selected for retesting the modified SUT if the execution trace for *t* contains a dangerous edge. *DejaVu* guarantees safety as long as equivalent execution traces for identical inputs imply equivalent behaviors.

However, in certain situations *DejaVu* is not as precise, as it seems. Especially with programs in environments that cannot be controlled, *DejaVu* can provide only a guideline for safety. For instance, in a system in which the state of the operating system can non-deterministically affect program behavior, and in which the state of the operating system cannot be controlled or simulated, *DejaVu* could omit a potentially fault-revealing test case. This aberration not withstanding, *DejaVu* is a very precise RTS technique.

Test Tube and *DejaVu* have been two very popular safe RTS techniques which prompted a lot of researchers and industry experts alike to attempt implementations. A comparative study of these two techniques would reveal that they have very little to chose. In summary, *TestTube* and *DejaVu* were often comparable in their costs, although for some programs, one tool or the other could prove superior. For example, when considering applications/programs which were not too big, *TestTube* proved the more efficient choice. On the other hand, when considering programs with more than 50,000 lines of code, *DejaVu* consistently reduced re-verification time by a significant amount, whereas *TestTube* varied in its behavior.

Study reveals that there are two main factors that can contribute to the success or failure of the two safe RTS tools.

- First, the program's size (or more probably the complexity of the program's control structure) seems to be an important factor. Presumably, larger programs may decompose more easily into different control paths, which, in turn, will allow *DejaVu* and *TestTube* to more readily separate out test cases with potentially changed behaviors.
- Second, the costs of executing and validating test cases can have a significant effect on the cost-effectiveness of both methods. Where test execution is inexpensive, analysis will generally assume a larger percentage of the re-verification cost. For a tool such as *DejaVu*, for which the bulk of the cost is in analysis of the code itself, large test suites can amortize the cost of analysis, in which case the tool might be cost-effective even for test suites with test cases that require only seconds to execute and validate. Most of *TestTube*'s cost, on the other hand, is derived from the time required to select test cases using the difference file; thus it is likely that *TestTube* will suffer if test cases are inexpensive to execute and validate. On the other hand, when test cases are sufficiently expensive to execute and validate, experiments suggest that both methods can make substantial gains in efficiency.

- Further research is necessary to determine if these are the only factors contributing to the success or failure of these tools, or even the most important ones.

The study on these two techniques was conducted mostly using large componentized programs written in C. While this is not exactly the interest of this thesis, studying these techniques gave a good understanding of safe regression techniques. A pretty obvious conclusion drawn from this is that, a tool built with both *Test Tube* and *DejaVu* as basis, a hybrid one, would give significantly improved results and might be an apt technique for such applications to perform Regression Testing and automating the test case selection.

3.2 Selective RTS Techniques

Several selective regression testing techniques have been proposed. Selective regression testing techniques have been extensively studied because researchers believe that selective techniques can reduce the cost of regression testing. *Selective* regression techniques allow the testers to reuse a subset of the original test cases. However, if we consider the time and resources required for test selection, we find that the *selective* techniques do not always reduce the total cost of regression testing. If the cost in analysis for test case selection is too high, or the number of selected test cases is not significantly smaller than the number of original test cases, the selection is not cost effective. Leung and White [4] analyzed the various factors that can affect the cost of regression testing, and proposed a simple model to compare the cost between the *selective* regression testing strategy and the *retest-all* strategy. They claimed that a benefit is accrued only if the effort spent in test selection is less than the costs for executing the extra test cases and for checking the results of the extra test cases used by the *retest-all* strategy [4]. This model established a basis for cost comparison between the two regression testing strategies. However, because this model makes some simplifying assumptions that may be inappropriate for some systems, it has limitations. Rosenbhim and Weyuker [5] extended Leung and White's model [4] by developing cost-effective predictors to determine the cost effectiveness of a *selective* regression testing strategy. Their design of the predictors is based on some fundamental assumptions about the nature of test coverage and the nature of changes made to a system.

The selection in selective regression testing is driven by two kinds of analysis: *coverage analysis* and *change analysis*. Rosenbhim and Weyuker's experiences lead them to find that, in practice the relationship between the test suite and the entities the test suite covers (i.e., the coverage of tests) in a software system changes very little during maintenance, except when new features are added to the system [5]. This means that the coverage of a test case in different versions of a system changes very little. They also observed that the ability of a method to find a subset of test cases from a test suite is governed by the nature of the coverage relationship [5]. For example, if there is a great deal of overlay in the sets of entities of the system that each test case covers, they do not expect a safe strategy to be able to exclude very many test cases [5]. Based on those considerations, Rosenblum and Weyuker presented their algorithm for predicting the cost-effectiveness of a selective regression testing technique for a given software system [5]. The algorithm is the following: *given a system under test, a candidate selective regression testing method, and a stable coverage relation, it suffices to evaluate the cost-effectiveness of the testing method on one version of the system in order to predict its cost-effectiveness for all future versions of the system* [5]. For a safe strategy, their predictor first predicts the cost-effectiveness of the strategy when a change is made to a single entity. If it shows the strategy is cost-effective, then further computation will be performed when changes are made to multiple entities. During the two steps, if there are any data indicating that the testing strategy is not cost-effective, the candidate strategy will be discarded [5]. Rosenblum and Weyaker's work is significant since it provides a method for determining whether or not a selective regression testing technique is cost effective.

3.3 Inter-Procedural Data Flow RTS Techniques

Data flow techniques can be broadly divided into two kinds. One for *intra*-procedural situations and the other is for *inter*-procedural situations. Although a lot of research went into data flow methodologies, [7,8] they were addressing data dependency that existed within a single process or procedure i.e., *intra-procedural*. Testing the data dependencies that exist among different processes or procedures i.e., *inter-procedural* requires information about the flow of data across procedure boundaries, including both calls and returns. The data dependencies that exist between procedures both directly over single calls and returns and indirectly over multiple calls and returns are needed. The current data flow testing tools either use intra-procedural data flow analysis typically employed in compiler optimization to determine the data dependencies or determine the definition-use pairs from the source code by building and then searching the program's def-use graph. Although inter-procedural data flow analysis algorithms do exist, [9,10] they do not provide the detailed information (i.e., the locations of definitions and uses that reach across both procedure calls and returns) needed for the inter-procedural data flow testing. Also, the methods for guiding the actual data flow testing do not currently handle the renaming of variables that is required when performing inter-procedural testing.

The underlying premise of all of the data flow testing criteria is that confidence in the correctness of a variable assignment at a point in a program is dependent on whether some test data has caused execution of a path from the assignment (i.e., definition) to points where the variable's value is used (i.e., use). Test data adequacy criteria are used to select particular definition-use pairs or sub-paths that are identified as the test case requirements for a program. Then, test cases are generated that satisfy the requirements when used in a program's execution. Thus, inter-procedural data flow testing consists of -

(1) Determining the definition-use information for definitions that reach across procedure boundaries (both calls and returns) to meet the adequacy criteria and

(2) Guiding the selection and execution of test cases that meet the requirements.

The problems of determining inter-procedural definition-use information include the development of an efficient technique. The technique should be procedure-call-site specific and handles reference parameters, global variables and recursion for direct and indirect data dependencies. Direct dependencies exist when either:

(1) a definition of an actual parameter in one procedure reaches a use of the corresponding formal parameter in a called procedure or

(2) a definition of a formal parameter in a called procedure reaches a use of the corresponding actual parameter in the calling procedure.
Conditions for indirect dependencies are similar to direct dependencies except that multiple levels of procedure calls and returns are considered. When a formal parameter is passed as an actual parameter at a call site, an indirect data flow dependency may exist. In this case, a definition of an actual parameter in one procedure may have uses in procedures more than one level away in the calling sequence, considering both calls and returns. For the selection and execution of the test cases, the presence of reference parameters requires incorporating the renaming of variables as procedures are called and returned, which complicates the design of a testing tool.

Most of the techniques concentrate on providing sets of variables that are used or modified by procedure calls using either flow sensitive or flow insensitive information. Although this information is useful in optimizations and parallelization, it does not provide the locations of the definitions and uses of variables that reach across procedure boundaries. A new method for computing the inter-procedural definition-use information would involve the development of an efficient representation of the procedures within a module and an algorithm to propagate data flow information throughout a module, taking into account reference parameters. One possible way to represent the program is by in-line substitution of procedures at call sites. In addition to the obvious problem of the memory requirements, inline substitution has other inherent problems. Both scoping of local variables in procedures and binding of formal and actual parameters are difficult because the entire module is viewed as one procedure. Additionally, recursive procedures cannot be represented. Another possible representation is the traditional *call graph* of a module where nodes in the graph represent procedures and edges represent call sites. However, the *call graph* is not sufficient for computing the definition-use information across procedure boundaries because it has no return information and provides no information about the control flow in individual procedures.

Module representation design would then lead one to the next problem which is - the data propagation throughout the module. Here, the requirements are that the algorithm be efficient in both memory requirements and execution time, and that it handles inter-procedural definition-use computation for both recursive and non-recursive procedures. Definitions that reach across procedure boundaries include definitions of global variables that reach a call or return site, definitions of actual parameters that reach a call site and definitions of formal parameters that reach a return site. A similar situation exists for uses. Most of the existing inter-procedural data flow analysis techniques make worst case assumptions at the call sites that involve recursion due to the fact that incomplete information is known about the called procedure. And, when it comes to enterprise applications, it can be a much bigger problem as we have seen in Chapter 3.

Other problems deal with guiding the testing of a module to meet the test case requirements that were computed using the results of the inter-procedural data flow analysis. A procedure is instrumented to record the execution path that occurs when the procedure is executed with particular test data as input. The testing application then searches the execution path for the desired definition and use, making sure that the variable is not redefined on the sub-path between them. Thus, an important component of such an application is the information about the locations of all definitions of a variable being considered. With global variables, the names of the variables remain the same throughout the module and thus, the locations of the definitions can be easily computed during inter-procedural analysis and used in the testing.

However, for reference parameters, this is not the case. While searching for a definition and use pair, the execution path moves from procedure to procedure, causing the name of a definition to change. In order to ensure that the sub-path from the definition to the use has no redefinition of the associated variable, the pairings of the actual and formal parameters must be handled when the execution path reaches a call or return site. A problem occurs when a definition and use are separated by a number of procedure calls. The actual parameter associated with a use in a procedure must be bound to the appropriate formal parameter on the returns along the call chain.

This brings one to the next issue when considering data flow techniques of inter-procedural situations i.e., the actual testing. The inter-procedural data flow testing is performed in two parts:

(1) static analysis of the module to compute the inter-procedural definition-use information for the test case requirements and
(2) dynamic testing that guides the testing of the module to meet the requirements.

Some assumptions are necessary for such an implementation. They are,

(a) the user has chosen the '*all-uses*' criterion for the testing and
(b) only the definitions of reference parameters that have inter-procedural uses are considered. Global variables can be handled similarly.

Computation of data in such a technique by Harrold [12] represents the module by a graph, the inter-procedural graph (IFG) that is based on the program summary graph. [6] The algorithm, by Callaghan, computes the inter-procedural definition-use information using the IFG for modules with both recursive and non-recursive procedures. The technique has four steps which are summarized below -

Step I - Construction of IFG sub-graphs to abstract control flow information for each procedure in the program. A sub-graph is constructed for each procedure where nodes represent regions of code associated with points that are of interest inter-procedurally, and edges represent the control flow in the procedure. Local information is computed for non-local variables and is attached to appropriate nodes in the graph.

Step 2 - Construction of an IFG to represent the inter-procedural control flow in the program. The sub-graphs of the procedures, obtained in step 1, are combined to create the IFG which is constructed by creating edges that represent the bindings of formal and actual parameters in both called and calling procedures. Preserved information is computed for each procedure, using the IFG, and edges that represent this information are added to the graph.

Step 3 - Propagation throughout the graph to obtain global information. The local information at each node is propagated in two phases throughout the graph resulting in the inter-procedural definitions that reach, and the inter-procedural uses that can be reached from, the parts of the program represented by the node in the graph.

Step 4 - Computation of the inter-procedural def-use and use-def chains. Inter-procedural def-use and use-def chains are computed using both the local information and the propagated global information.

This work would have been complete with cost analysis especially with some experimentation to determine the cost of applying inter-procedural data flow testing, especially when changes are made within a module. In addition, the same principle of summarizing information can be further applied to determine the problems of developing an inter-modular data flow testing technique.

Chapter 4 - RTS FOR APPLICATIONS

Regression testing is an important activity of software maintenance, which ensures that the modified software still satisfies its intended requirements. It is an expensive testing process that attempts to revalidate modified software and ensure that new errors are not introduced into previously tested code. At application level, RTS implementation has been attempted at various levels in an enterprise application. This chapter considers the implementations of RTS for specific types of software systems.

4.1 RTS for Database Systems [18]

Software revalidation involves essentially four issues: change impact identification, test suite maintenance, test strategy, and test case selection [17]. Database systems have been accepted as a vital part of the information system infrastructure. In addition to traditional software testing difficulties, database application features such as SQL, exception programming, integrity constraints, and table triggers pose some difficulties for maintenance activities; especially for regression testing that follows modifications to database applications. DBM Systems are of various kinds. Relational, Distributed and Network are just a popular few to name. But, this study limits its scope to the relational database systems.

The following procedure is a 2-phase approach. Phase 1 involves detecting modifications and performing change impact analysis. The impact analysis technique localizes the effects of change, identifies all the affected components and selects a preliminary set of test cases that traverse modified components. Phase 2 involves running a test case reduction algorithm to further reduce the regression test cases selected in phase 1. The algorithm employed here is an adaptation of the firewall regression testing technique on the inter-procedural level that utilizes data flow dependencies.

Many technical difficulties pose as challenges when dealing with databases. Two such issues are discussed here, namely, *control flow* and *data flow* issues.

Building *control flow graphs* for database modules differs slightly from building control flow graphs for conventional software. This difference results from the extensive usage of exceptions and condition handlers and the nature of the SQL language that is a key feature of database modules. Therefore, we should devise new modeling techniques to model the control transfers that are available in database modules. The semantic of all SQL statements make them behave like micro-transactions in that either they execute successfully, or they have no effects at all on the stored data [19].

A database module consists of one compound statement in which other compound statements are nested. Each compound statement has its exception handler. During execution, if an exception is raised from an SQL statement then the control is transferred from the current statement to the exception handler according to the type of the exception raised.

A solution to counter this problem would be to make each statement be represented by a node in the control flow graph. These statements are either SQL statements or control statements or others. A compound statement contains a list of statements with one exception handler for all of these statements. Each of these statements is represented by a node. The compound statement contains two end statements one for successful endings and the other for unhandled exception results. If exception handling is not available, then all the exception links of these nodes will be linked to the unhandled exception end node. If exception handling is available then the exception handler is modeled by a primary handler switch node to which all the exception links of the compound statement nodes are linked. Each specific exception handler is modeled by a predicate node that checks for the type of the exception. The exception predicate has two links: the first one is to the start node of the exception handler block and the second to the next handled exception.

On the other hand, Data flow analysis focuses on the occurrences of variables within the program. Each variable occurrence is classified as either definition occurrence or as use occurrence [14]. The database plays an important role in holding the state of computation in database modules. The data generated by a statement is used by other statements in the same module or other modules; thus creating data flow relations. The main source of data in a relational database is tables.

Thus, to define the data flow relations created from the database usage the database variables have to be designed based on a level of granularity which would help in tracing their definition and later use.

A solution to this would be to choose the level of granularity at the column level. Since the number of columns is fixed and columns are used in SQL statements using their unique names, we can determine the column usage statically. A drawback of this choice is the fact that it does not discriminate between the usage of one particular column value belonging to some row and the usage of the same column but of a different row. SQL statements use columns directly and indirectly or, in other words, explicitly and implicitly. These usages are either definition or retrieval. A table participating in master detail relations has a group of its columns referencing the primary key columns of the master table. Whenever these columns are defined the database implicitly checks that the master table contains a record that has its primary key column values matching the foreign key column values of the newly added record. So, whenever a new record is created the primary key columns of the master table are used. This solution differentiates between five main usages of database columns. They are *delete*, *insert*, *reference*, *select*, and *update*. Reference and select usages are computational usages denoted by c-use. Update, delete and insert usages are define uses denoted d-use.

4.1.1 Phase I – Change Identification & Impact Analysis

A change made to one component affects other database components due to component dependencies. Therefore, to identify the impact of change, we should identify the dependencies that exist between database application components and then find the wave effect of change due to the transitivity of the dependency relations.

Before impact identification is done, it is essential to understand the potential changes in a database system. We differentiate between two types of changes in the database applications environment:

a) **Code** - This involves changes that can be made to the code of the database modules

b) **Database Component -** This change involves the changes that could be made to the definition of the database components in general.

The technique explained below for impact identification and to determine the affected database components is known as the *Component Firewall* technique. This is an algorithm whose goal is to perform the change impact analysis, including localizing the effects of change, identification of affected components and make a preliminary test case selection of those which traverse all components.

A component firewall is a set of affected modules when some changes are made to any of the database components. A database component is marked as modified and is included in the component firewall if one of the following conditions is satisfied -

- Its definition is modified.
- It is deleted.
- It is dependent on a modified or deleted component.
- It became dependent on new or modified components in the new system such as triggers and constraints.

All database components selected by the Component Firewall Algorithm are marked as affected components. Affected module components are classified alone so that one can select a test case passing through them to become a part of the results acquired in phase 1 of this regression testing methodology.

4.1.2 Phase II – Test Case Reduction

The component firewall does not give us hints to discriminate between the test cases passing through a module included in the firewall. The test cases passing the modules that are included in the firewall are selected for regression testing. This will result in a large number of test cases.

Hence, this work is extended to optimize on this result. A new technique is suggested to reduce number of test cases selected from the previous phase. This is called the Call Graph technique. It has been adapted from a firewall regression testing technique [20].

Leung and White [21] present a selective retest technique aimed specifically at inter-procedural regression testing that deals with both code and specification changes. This technique determines where to place a firewall around modified code modules. The test cases selected by the firewall regression testing technique are composed of two types of tests: *unit tests* and *integration tests*.

Unit tests are tests used to test only the directly affected modules. Integration tests are test cases passing to the directly affected modules from higher modules in the call graph. These test cases are selected when there are data flow interactions between modules in the call graph.

Where test selection from regression test suite is concerned, the technique selects unit tests for modified modules that lie within the firewall, and integration tests for groups of interfacing modules that lie within the firewall. This technique was further extended to handle interactions involving global variables. Implementing the firewall concepts for database applications has three requirements -

- Database application call graph.
- Data flow dependencies between interfacing modules resulting from database tables usages.
- List of modified database modules.

The call graph links a database module to all the modules that it calls. It should include links to table triggers modules in case the module contains statements that causes these triggers to execute. Based on some of the results obtained by some experimenters, the Call Graph Firewall technique is effective with modular applications.

4.2 RTS for Java based applications [22]

Although object-oriented languages have been available for some time, very few safe regression-test-selection algorithms that handle features of object-oriented software have been developed. However, these approaches are limited in scope and can be imprecise in test selection. Rothermel, Harrold, and Dedhia's algorithm [23] was developed for only a subset of C++, and has not been applied to software written in Java. The algorithm does not handle some features that are commonly present in object-oriented languages; in particular, it does not handle programs that contain exception-handling constructs. Furthermore, the algorithm must be applied to either complete programs or classes with attached drivers. For classes that interact with other classes, the called classes must be fully analyzed by the algorithm. Thus, the algorithm cannot be applied to applications that call external components, such as libraries, unless the code of the external components is analyzed with the applications. Finally, because of its treatment of polymorphism, the algorithm can be very imprecise in its selection of test cases. Thus, the algorithm can select many test cases that do not need to be rerun on the modified software.

The work described below is the first safe RTS technique for Java that efficiently handles the features of the Java language, such as polymorphism, dynamic binding, and exception handling. This method is an adaptation of Rothermel and Harrold's graph-traversal algorithm [23], which uses control-flow-graphs to represent the original and modified versions of the software and hence selects the test cases to keep. This technique's representations are modeled with the effects of unanalyzed parts of the software, mainly libraries. Hence, it can be used for safe regression test selection of applications without requiring complete analysis of the libraries that they use. This provides significant savings during regression testing, considering the heavy dependency of Java based software on libraries. Also, the technique provides a new way to handle polymorphism that can result in the selection of a smaller and safe, subset of the test suite.

4.2.1 Assumptions - Regression Bias [24]

Some assumptions about the code under test, the execution environment, and the test cases in the test suite for the original program are essential for the algorithm to be safe without being too inefficient and too conservative. Also known as *regression bias,* they cover the following areas of coding in Java.

a) *Reflection* - This technique assumes that reflection is not applied to any internal class or any component of an internal class. The assumption is that the methods that inspect the information about a specific class, such as the methods in java.lang.Class, as a form of reflection as well. If a statement uses information obtained through reflection about either an internal class or its members, the behavior of this statement may be affected by several kinds of changes occurring in the class and/or its members. In such cases, the identification of all the points in the code affected by a change may require sophisticated and expensive analysis of all the reflection constructs in the code. Moreover, if a statement in an external class uses reflection to inspect the information about an internal class, then the external class must be analyzed to identify the code affected by a change of the internal class.

b) *Independent external classes* – The assumption here is that external code has no knowledge of the internal classes. That is, the external classes can be compiled without the internal classes, and that external classes do not load any internal class explicitly by invoking a class loader with the class name as a parameter. This assumption guarantees that the external classes interact with the internal classes only through a set of predefined virtual methods. Thus, this assumption reduces the types of interactions between the internal and external classes that must be considered. This is a logically correct and sensible assumption as, in practice; the external classes are often library classes that are developed independent of, and prior to, the development of the applications that use them.

c) *Deterministic test runs* – Finality to a test case. This assumption is that a test case covering the same set of statements, and produces the same output, each time it is run on an unmodified program. This guarantees that the coverage information obtained by running the original program with the test cases does not depend on any one specific test run. Under this assumption, if the internal classes are represented correctly, based on the information in the coverage matrix, then, test cases that do not traverse modifications can be safely excluded. One possible threat to this assumption is a change in the execution environment. This could lead to unforeseeable changes in results. Therefore, the tester must ensure that elements such as the operating system, the Java Virtual Machine, the Java compiler, the external classes, databases and network resources possibly interacting with the program are fixed. Again, this is quite logical, as one wouldn’t want to have environmental changes from development to test to production setups. Another possible threat to this assumption is the presence of nondeterministic behavior. The assumption holds for sequential programs, which contain only one thread of execution, and for those multithreaded programs in which the interaction among threads does not affect the coverage and the outputs (e.g., an ftp server whose multiple threads are just clones created to handle multiple clients). The assumption, however, does not hold in general for programs that contain multiple threads of execution. So, a safe assumption is to use special execution environments that guarantee the deterministic order in which the instructions in different threads are executed.

4.2.2 RETEST [22]

Based on the above mentioned assumptions, following is a brief description of an RTS technique developed by Harrold and team to test Java based software applications, called *RETEST*. This is based on the existing Control Flow Graph technology, with its limitations for object oriented software, it is modified to suit this work and called – *Java Interclass Graph (JIG)*.

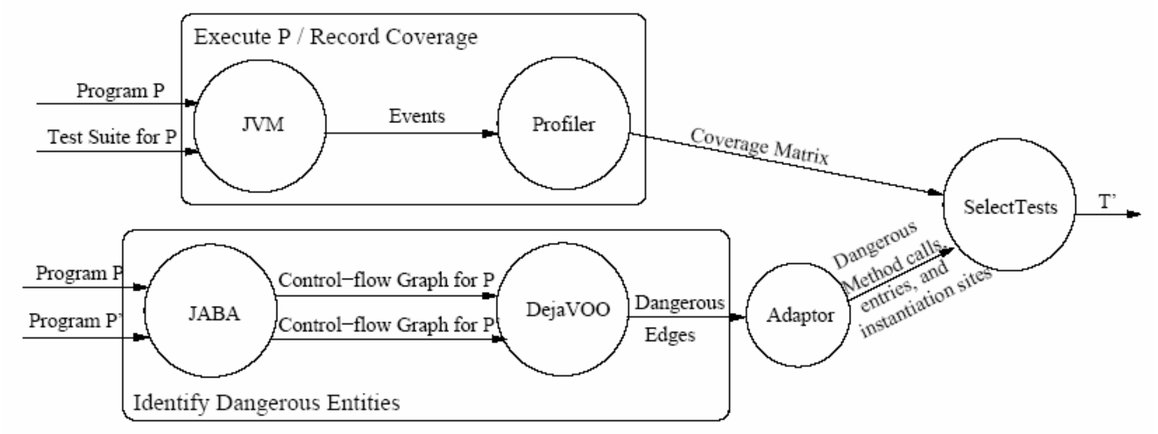

Figure 4-1: RETEST Architecture

A JIG accommodates the Java language features and can be used by the graph-traversal algorithm to find dangerous entities by comparing the original and modified programs. A JIG extends the CFG to handle five kinds of Java features: (1) *variable and object type information*; (2) *internal or external methods*; (3) *interprocedural interactions* *through calls to internal or external methods from internal methods*; (4) *interprocedural interactions through calls to internal methods from external methods*; and (5) *exception handling*.

Variable/Object type Information – The CFG solution for representing any changes in global declarations is to mark the edges from the entry node to the declaration node of the variable as dangerous, thus automatically selecting all the test cases traversing that edge. But, in RETEST, the attempt at making it more precise, led to the idea of the global variable/object representation to include the *type* of the variable also in it. For instance, a variable, "f", if it is a float, would be stored as "f_float". And, in the same vein, extending this idea, any object would be represented using a globally qualified, full class name. This method for representing class hierarchies also pushes the changes in class hierarchies to the locations where the affected classes are instantiated.

Internal or external methods – Each internal method in a class is in turn represented as a CFG inside of a JIG. Thus JIG could be a compilation of CFG(s). But, there are subtle modifications made to the traditional CFG. In that, each call site is expanded into a *call* and a *return* node. Also, there is a *path* edge between the call and return node that represents the path through the called method. For instance, consider the scenario illustrated in the Figure 4-2. The node labeled "p.m()" represents a call node; it is connected to the return node with a path edge. These are used to represent calls to made to third party libraries. Usually, the source code for the external classes is not available, and, even if it were available, analyzing it is out of scope and expensive. Of course, the assumption for this being, the external classes do not change. Thus, each collapsed CFG consists of a method entry node and a method exit node along with a path edge from the method entry node to the method exit node. The path edge summarizes the paths through the method.

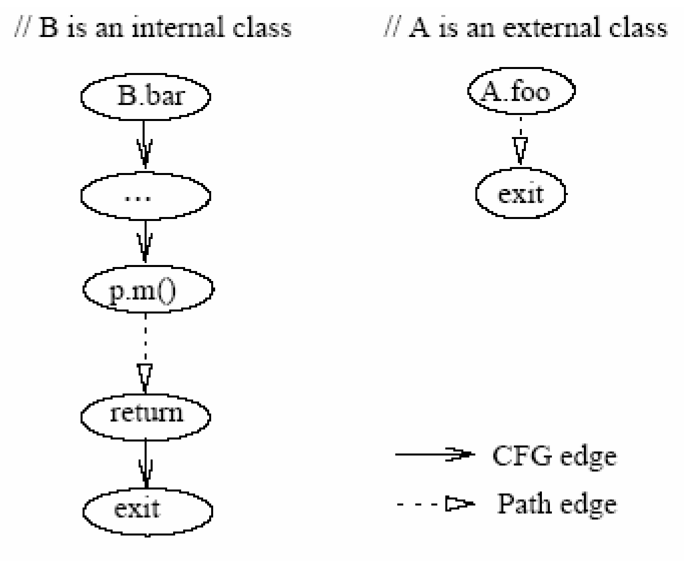

Figure 4-2: CFG in JIG

Inter-procedural interactions through internal method calls - The JIG represents each call site as a pair of call and return nodes that are connected with a path edge. The call node is also connected to the entry node of the called method with a *call edge*. If the call is not *virtual* the call node has only one outgoing call edge. If the call is *virtual* the call node is connected to the entry node of each method that can be bound to the call. Each call edge from the call node to the entry node of a method *m* is labeled with the type of the receiver instance that causes *m* to be bound to the call. To represent virtual method calls correctly, we must compute, for each virtual call site, the set of methods to which the call may be bound. Such information can be computed using various type-inferencing algorithms or points-to analysis algorithms [25]. The precision of this computation determines the efficiency of the representation. In this technique, class hierarchy analysis is employed to resolve the virtual calls. Using this representation, it can identify, by traversing the JIGs constructed for the original and modified programs, the internal method calls that may be affected by a program change. This leads to the conclusion that the edge is dangerous.

Inter-procedural interactions through external method calls - Potential subtle interactions between internal classes and external classes may lead to different behavior in the program, as a consequence of apparently harmless changes in the internal classes. Therefore, in the case of incomplete programs, one must consider the possible effects of the un-analyzed parts of the system. In particular, unforeseen interactions between internal classes and external classes may be caused by calls from external methods to internal methods. To handle this situation, in a summarized form, potential calls to internal methods from external methods are explicitly represented. A *class entry node* for an internal class *A* represents an entry point to the internal code through an object of type *A*, and is connected to the entry of each method that can be invoked by external methods on objects of type *A*. The only internal methods that can be invoked by external code are those methods that override an external method. Therefore, one must create a class entry node for (1) each class that overrides at least one external method, and (2) each class that inherits at least one method overriding an external method. In addition, for each class entry node, we create an outgoing default call edge labeled "*", and we connect it to a *default node*. The *default node* for a class A represents all methods that can be invoked through an object of a type A, but that are externally defined. This representation lets us correctly handle modifications that involve addition or removal of internal methods that override external methods. Using this representation, this algorithm can identify, by traversing the JIGs constructed for the original and modified programs, the external method calls that may be affected by a program change. Thus, this algorithm identifies this edge as dangerous.

Exception Handling - A JIG explicitly represents the try block, the catch blocks, and the finally block in each *try* statement. For each try statement, we create a *try* node in the CFG for the method that contains the statement. We represent the try block of the try statement using a CFG. There is a CFG edge from the try node to the entry of the CFG of the try block. We create a *catch node* and a CFG to represent each catch block of the try statement. A catch node is labeled with the type of the exception that is declared by the corresponding catch block. A CFG edge, labeled "caught", connects the catch node to the entry of the catch CFG.

A path edge, labeled "exception", connects the try node to the catch node for the first catch block of the try statement. That path edge represents all control paths, from the entry node of the try block, along which an exception can be propagated to the try statement. A path edge labeled "exception" connects the catch node for a catch block *bi* to the catch node for catch block *bi*+1 that follows *bi*. This path edge represents all control paths, from the entry node of the try block, along which an exception is (1) raised, (2) propagated to the try statement, and (3) not handled by any of the catch blocks that precede *bi*+1 in the try statement. Using this representation, it can identify the changes in exception-handling code by traversing the JIGs constructed for the original and the modified programs.

4.2.3 Implementation Issues

Given a JIG for a program P, we can instrument P or modify the execution environment to record the edges covered by each test case. The coverage information lets the regression-test-selection technique select test cases that cover the dangerous edges identified by the traversal algorithm. However, because some edges (e.g., path edges for exception handling) do not represent actual control flow from one statement to another, we cannot instrument the program or the execution environment to find the test cases that cover such edges. Moreover, because recording the coverage information for each edge can be very expensive, we may want to record coverage information for coarser-grained entities, such as methods, classes, or modules. Thus, some dangerous edges must be mapped to another set of entities whose coverage information is recorded in the coverage matrix.

In general, we need an adaptor that takes a set of dangerous edges from the traversal algorithm and maps them to a set of dangerous entities whose coverage information is recorded in the coverage matrix. The adaptor must be designed together with the instrumenter because it needs to know the entities whose coverage information is being recorded. To get a better trade-off between precision and efficiency, the instrumenter also needs to know which entities are of interest to the adaptor.

Instrumentation is done using two kinds of methods, namely;

a) *Edge-level* instrumentation techniques record, for the internal methods, the CFG edges that are covered by each execution of a program P. In a JIG, edges representing calls from external methods and path edges representing the control paths on which exceptions are raised need to be mapped to actual CFG edges and nodes, so that the instrumenter can record the test cases that cover such edges.

b) *Method-level* instrumentation techniques record the internal methods that are covered by each execution of the program. Using this instrumentation technique, the adaptor maps each dangerous edge in the JIG for a method *m* to *m*. For each call edge *e*, if the target of *e* is the entry node of an internal method *m*, the adaptor maps *e* to *m*. Otherwise, if the target of *e* is the entry node of an external method, the adaptor maps *e* to the method that contains the source of *e*.

Instrumentation at a coarser level of granularity is more efficient than instrumentation at a finer level of granularity. In particular, method-level instrumentation is more efficient than edge-level instrumentation. Also, the coverage matrix computed using method-level instrumentation is smaller than the one computed using edge-level instrumentation. However, using a coverage matrix computed by method-level instrumentation, a test-selection algorithm may select more test cases than using a coverage matrix computed by edge-level instrumentation.

The results obtained using Retest, were but being conclusive, and were encouraging. They suggest that the technique can be effective in reducing the size of the test suite but that the reduction varies across subjects and versions. These results are consistent with results reported for C programs and for specific Java applications. However the results cannot be generic for all Java based software applications.

4.3 RTS for GUI applications [26]

Graphical User Interfaces (GUIs) are pervasive in today's software systems and constitute as much as half of software code. The correctness of a software system's GUI is paramount in ensuring the correct operation of the overall software system. One way, and the common way, to gain confidence in a GUI's correctness is through comprehensive testing. GUI testing requires that test cases (sequences of GUI *events* that exercise GUI *widgets*) be generated and executed on the GUI.

When a GUI is modified, the test cases in a test suite fall into one of two categories: usable and unusable. In the "**usable**" category, the test cases are still valid for the modified GUI and can be rerun. In the "**unusable**" category, the test cases cannot be rerun to completion. For example, a test case may specify clicking on a button that may have been deleted or moved. In early capture/replay tools that represented user events in terms of pixel coordinates of GUI widgets (also known as *analog mode* [1]), a moved widget could not be identified. However, modern capture/replay tools do not rely solely on coordinates for test case execution but maintain extra information such as the handle, type, and label (if any) of the widget, enabling the replayer to locate the widget when it has been moved. However, even with these modern tools, a large number of test cases are made unusable because of GUI layout changes such as the creation of a new menu hierarchy, moving a widget from one menu to another, and moving a widget from one window to another. A survey suggested that on an average more than 74% of such test cases eventually become unusable.

Memon and Soffa suggest a very novel RTS technique for GUIs. The crux of the solution is not to throw away test cases that are unusable for the modified GUI but to automatically repair them so they can execute on the modified GUI. Using this new repairing technique, a tester can

(1) rerun test cases that are usable for the modified GUI, as currently done,

(2) repair and rerun previously unusable test cases, and

(3) create new test cases to test new functionality.

This technique consists of two parts: a *checker* that categorizes a test case as being usable or unusable; if unusable, it also determines if the test case can be repaired. The second part is the *repairer* that repairs the unusable, repairable test case. Although for ease of explanation, these two parts are treated individually, they could be merged together in an implementation.

Details of this method are skipped here as it goes out of scope for this thesis work. But, below is a diagrammatic representation of this technique. Repairing old test cases to reuse them is a novel approach not done by any other researchers.

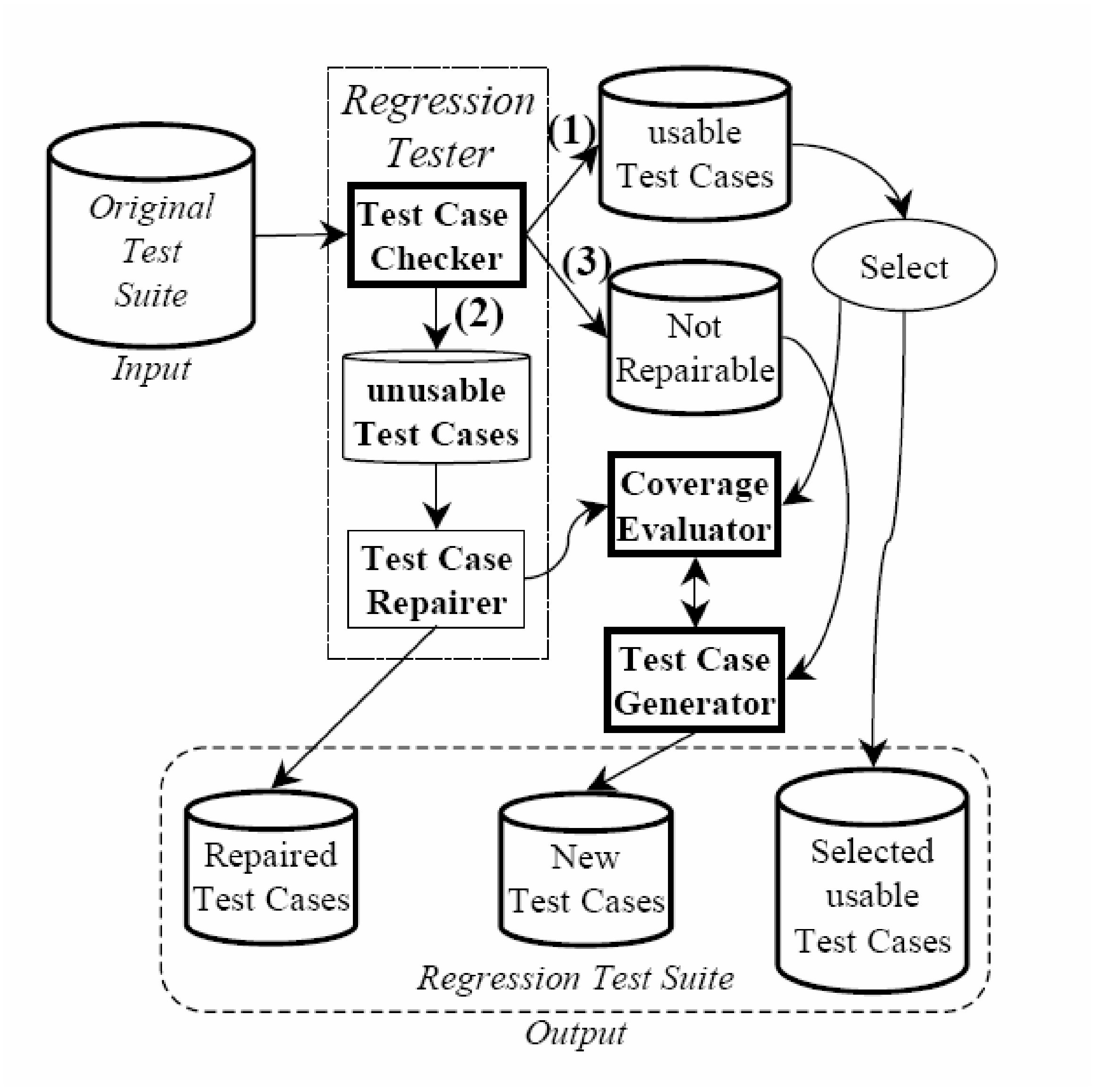

Figure 4-3: GUI Regression Tester's Architecture

With some of the salient areas of enterprise software applications, like database, object oriented language like Java and GUI covered, this concludes a study of RTS techniques for complex applications.

Chapter 5 - A CASE STUDY OF INTEGRATION WITH MESSAGING

5.1 Introduction

To my knowledge, very few attempts were made at implementing an RTS technique for Enterprise software integrations. This was one motivating point when we began our work on a messaging system implementation. In principle certain RTS techniques, when customized sufficiently should be able to carry out RTS for software integration.

The keen interest in trying to get a good messaging system which encapsulates all the potential pitfalls implemented was a challenge. PeopleSoft was one such complex system which offered many such challenges and hence, our choice of the messaging system.

PeopleSoft is a popular ERP solution provider. In fact, it is the second leading ERP package sold and customized in the world.

Integrating existing software to perform towards one goal is the business in the industry. Service Oriented Architecture (SOA) is the primary area of concentration involved in distributed computing. Many diverse applications perform various tasks. These existing modules can be integrated to achieve some new functionality. This approach is very economical and fast. All leading software vendors make their applications "integration ready". The idea is to create systems which can seamlessly integrate with any other system.

PeopleSoft has been one of the front-runners in this drive. *PeopleSoft Integration Broker* (PIB) has been created just for this job and forms the heart of all third-party communication with the core software of PeopleSoft.

It is this power of PeopleSoft which prompted us to use it for the enterprise application. So, before I get into the details of what exactly was implemented, I would like to explain about PIB.

5.2 PeopleSoft Integration Broker (PIB)

The PIB is an XML messaging hub which is used as a broker to publish and subscribe messages. The message itself is basically an XML message. This has been done in keeping with the idea of platform independence and to fully utilize the extensibility offered by XML. Thus, PIB becomes an ideal candidate for supporting Web Services and web service based architecture or SOA. Tapping this resource has been a challenge of sorts in this thesis.

The much needed enterprise application which we intended to use as the SUT to experiment on an architecture for some RTS technique had to be reasonably complex and also support good message inter-changing mechanism. It is here, that our research on PeopleSoft helped so that we could identify it as the candidate. So, before I go into the specifics of what we implemented, I would like to briefly explain the PeopleSoft's architecture and support for Web Services technology [27].

PIB interacts with other components by transporting messages to and from the systems using XML over HTTP. But, with the advent of the web services technology, it started native SOAP support for sending and receiving messages with other systems that communicate using SOAP. Using the Integration Broker technologies, PeopleSoft applications can be both web service clients and web service servers - a PeopleSoft application can invoke a web service or a PeopleSoft application can act as a web service. Based on this infrastructure, the PIB supports the following kinds of web service invocations –

1. Synchronous Web Services Support
2. Asynchronous Web Services Support
3. PeopleSoft to Mainframe Web Services Integration
4. Mobile Agent Synchronization and Web Services

Each of the above mentioned scenarios are supported well, because, PIB has native support for both WSDL & UDDI.

Using PeopleTools 8.4, developers can generate the XML Schema for a PeopleSoft Enterprise Integration Point (EIP). W3C, DTD, and BizTalk format schemas are supported. This XML Schema, along with their Integration Broker configuration information, can be used to generate WSDL for an EIP.

An example scenario can be stated here, using the diagram below. This is the flow of *customer profile* information from one module of PeopleSoft Customer Relations Manager (CRM) to another module Software Configuration Manager (SCM).

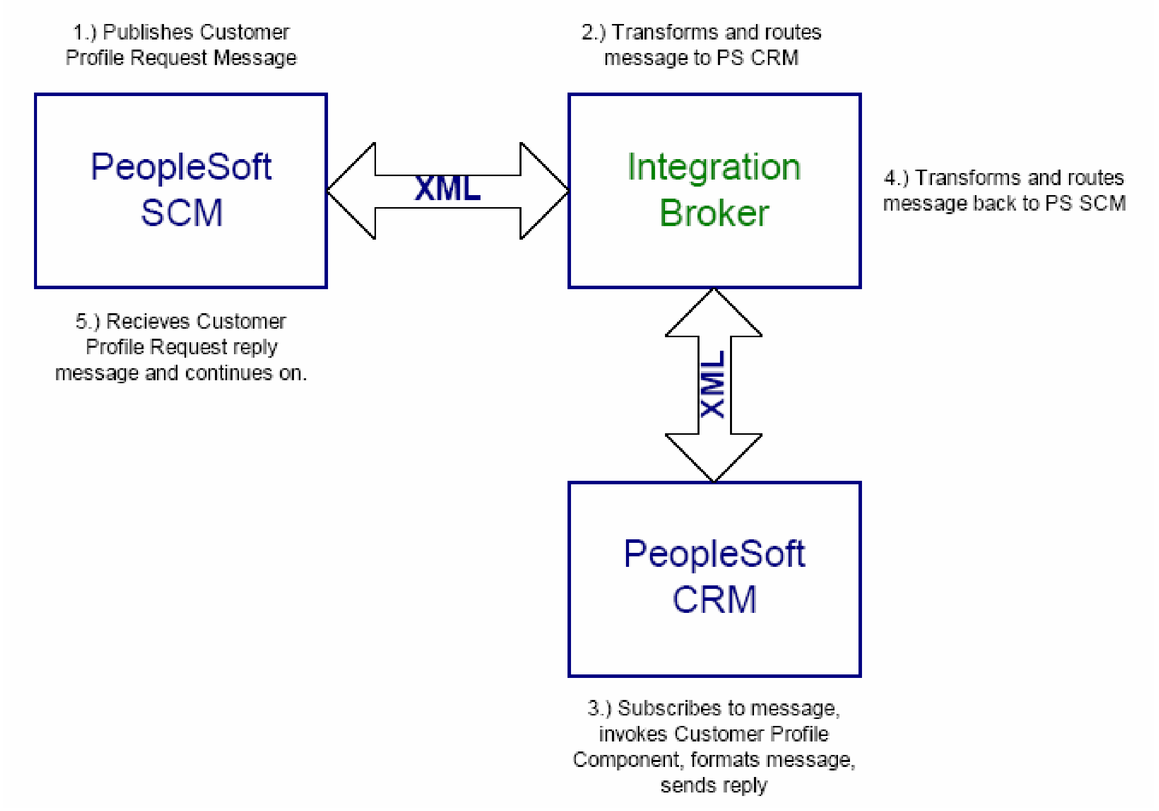
1.) Publishes Customer Profile Request Message
2.) Transforms and routes message to PS CRM
PeopleSoft SCM
XML
Integration Broker
4.) Transforms and routes message back to PS SCM
5.) Recieves Customer Profile Request reply message and continues on.
XML
PeopleSoft CRM
3.) Subscribes to message, invokes Customer Profile Component, formats message, sends reply

Fig 5-1: Synchronous PIB interactions

PeopleSoft SCM fetches a Customer Profile from PeopleSoft CRM using the following steps -

1. PS SCM publishes a synchronous Customer Profile Request message which contains the customer key data. This message is transported to the Integration Broker over XML/HTTPS.
2. The Integration Broker receives the XML message, does any message transformation that is necessary, and routes the request to PS CRM over XML/HTTPS.

3. PS CRM subscribes to the message, invokes the Customer Profile component using the customer key data that is passed in the XML message, updates the message structure with the Customer Profile data, and replies to the XML request from the Integration Broker.
4. The Integration Broker then replies back to the original PS SCM XML request, passing it the Customer Profile data.
5. PS SCM receives the Customer Profile Request reply and continues on with its processing, using the Customer data fetched from PS CRM.

It should be noted that in the above example, either PeopleSoft systems could be replaced by non-PeopleSoft systems that communicate with the Integration Broker using SOAP or XML/HTTP. This was one of the key concerns of PeopleSoft. They wanted to provide an architecture which allows seamless interaction between PeopleSoft as well as, non-PeopleSoft components in a grander scheme of things. The reason I choose this example to explain this is the fact that we tapped on this power of PeopleSoft to set-up our application.

So, with the high-level architecture out of our way, I would like to explain how PeopleSoft supports system integration and application messaging as part of it.

5.2.1 System Integration

System integration encompasses a diverse range of requirements that vary - depending upon the systems involved. For example, when using asynchronous integration, a message-based interface is appropriate. In others, a synchronous request/reply, component interface can be the optimal solution. The direction of the data flow is also critical. Whether a system acts as a client or a server is an important consideration for selecting the appropriate integration technology. Unfortunately, there is no “silver bullet” for integration. No single solution or technology can accommodate all of the various types of integration required across today’s enterprise. So to address the numerous integration scenarios in today’s enterprise, PeopleSoft Internet Architecture delivers five integration technologies that support the full spectrum of integration both inside and outside the organization -

- *Application Messaging*: Publish/subscribe messaging architecture for asynchronous integration into and out of PeopleSoft applications.
- *Component Interfaces*: Object-oriented, request/reply, component architecture that allows third-party applications to synchronously invoke PeopleSoft business logic.
- *Business Interlinks*: Plug-in framework that enables PeopleSoft applications to invoke third-party APIs over the internet.
- *Application Engine*: Robust file processing capabilities for file-based integration—still a common method for addressing integration requirements.
- *Java Integration*: Application server integration with Java applications that enables third-party programmers and application developers to create business logic in Java.

5.2.2 Application Messaging

While each of the above mentioned offer wide variety of integration techniques, I shall concentrate on explaining application messaging, as I spent lot of time trying to understand the way they did this, and also employed this.

PeopleSoft Application Messaging is a server-based, publish and subscribe bus architecture that enables multiple PeopleSoft and non-PeopleSoft systems to integrate using a loosely coupled, message-based approach. This is depicted in the diagram below –

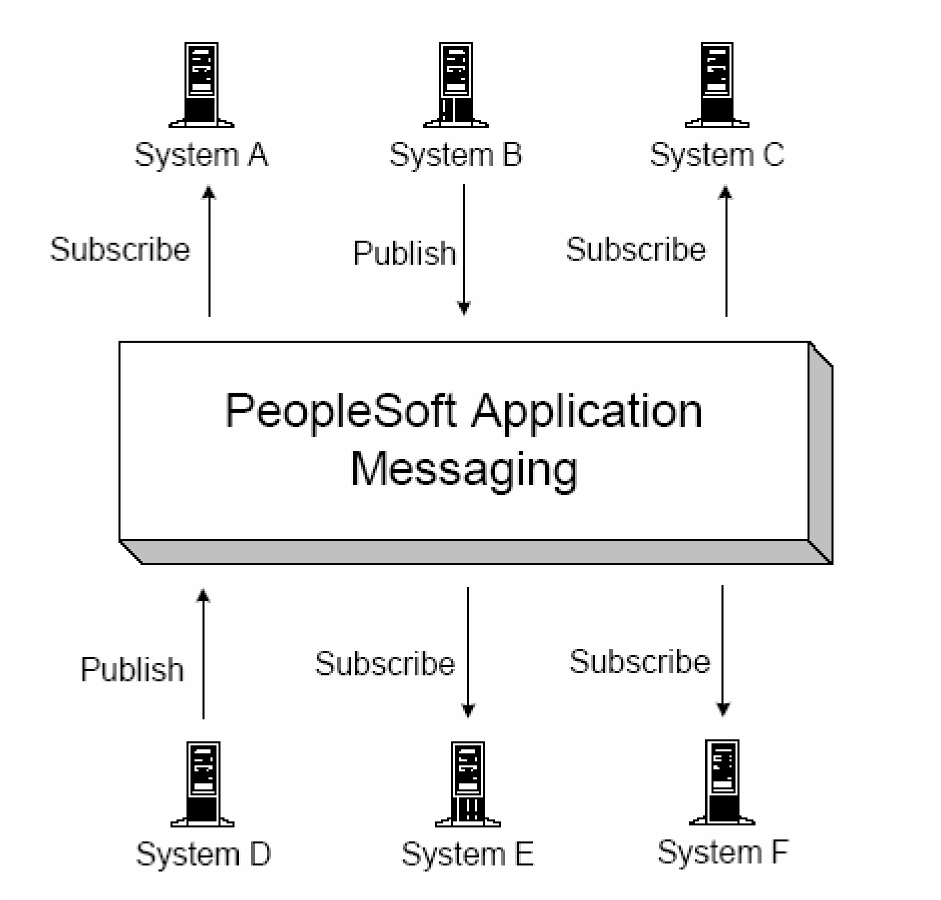

Fig 5-2: PeopleSoft Application Messaging Publish and Subscribe Bus Architecture

A key benefit of Application Messaging is that third-party systems can publish messages and subscribe to messages to and from the Application Messaging architecture over HTTP using XML. To publish a message, the third party simply performs an HTTP Post to the PeopleSoft Internet Application Server, passing the XML document. To subscribe to a message, the third party only needs to be able to receive an XML message over HTTP from the PeopleSoft Internet Application Server.

There are three new design tools within PeopleTools Application Designer dedicated to Application Messaging:

- *Message Designer* - Used to define the structure and the subscription processes of the message.
- *Channel Designer* - Channels are logical grouping of related messages (e.g., Expenses, Personal Data, and Accounting Entries). Content-based routing rules are defined at the channel level.

- *Node Designer* - Message nodes are the systems that application messages are published to and subscribed from. PeopleSoft HRMS, PeopleSoft EPM, SAP Financials, and Vantive CRM are examples of message node definitions.

5.3 Case Study [28]

As a case study for this project that requires the inter-enterprise integration, we carried out an experiment on integrating the GIS map imagery web services (TerraService) into the PeopleSoft Human Resource (HR) system. While the above explanation would suffice towards understanding the infrastructure of PeopleSoft, the details of GIS and the services used for it are not explained, as they are out of scope for this work. PeopleSoft HR is the most popular Commercial Off-The-Shelf (COTS) Human Resource management software in governmental organizations in the U.S. PeopleSoft uses a large database to store data and a set of comprehensive web-based interfaces to access the data. A user can access much of the information about employees through web forms. Developers can add or modify functionality to customize the system for their needs using the Application Designer.

As the software systems are integrated into more and more inclusive cycles, geophysical information naturally becomes the pivot point of integration. In recent years, with sophisticated requirements and evolving and ever-demanding industrial needs, urgent need for comprehensive and real-time information resources became imperative. Building new systems for these tasks would be costly. It is then that people realized integrating existing systems can deliver economical solutions. Many independent systems already deliver solutions for small problems. For example, to effectively stop spreading a disease from a known location, the public health agents need to immediately identify the potentially contaminated areas. Because each individual system contains its own information separately, the fragmented information cannot be utilized without integration. Considering that the municipal business bureau's database has the names of the companies around the contaminated area; the personnel information system of each of the companies has the address of every employee; and the general public GIS has the map to mark the potentially contaminated houses. In principle, web services should fit into this kind of integration well.

Our goal was to integrate the map imagery capability into the PeopleSoft HR system. By clicking on a PeopleSoft HR page showing an employee's information, we want the PeopleSoft page to trigger the display of an image map showing the employee's residence. An easy example of a requirement for this would be during an emergency situation caused due to a natural calamity or for the homeland security agent. To achieve this, we need to do two things:

(1) Get the address information out from PeopleSoft and find its geocode;

(2) Fetch images from a map image service such as TerraService and display them.

Geocode implies the geographical address of a location, i.e., the latitude and longitudinal values.

5.3.1 Information Capture from PeopleSoft

Capturing an employee's information on the PeopleSoft application and sending it out was the first task at hand. By sending it out, we mean to utilize the PIB and register a service outside of PeopleSoft and make the HR application send out a message to this service so that we can perform the necessary action subsequently.

Our primary design decision was to ensure that we make the least bit of a change in the existing PeopleSoft code. This led to the following initial attempt –

As a first attempt, we decided to send a message to an *Address_Storage* web service we created using *Apache Axis* and deployed on a *Tomcat Server*. This web service has the necessary business logic built in to accept data in a specific format and eventually convert that into its own custom format to store the name and address information of the employee who made the current change in PeopleSoft. We configured the PeopleSoft applications to send NAME_AND_ADDRESS_MESSAGE (a message type we created) to the Address_Storage web service. The message designer, which I explained above, was employed to create this. Refer Appendix I for the message.

Next, we had to actually send the message. To do that, we added code to a standard PeopleSoft event called *SavePostChange*. This event is called whenever a user clicks the *Save* button on a PeopleSoft screen with new data. As the name of the method suggests, it would take in the changed data which was posted by the user and save it to the persistent storage device. In this implementation, *SQL Server 2000* was employed as the persistent storage database.

PeopleSoft, being the COTS application that it is, has built-in pages which already have certain data capture abilities in them. Every module has many such pages, which are either already functionally ready, or would require minimal customization. For our requirement, we found one such page in PeopleSoft HR that allowed employees to modify their address information. This is called PERSONAL_DATA. We added the event handling code (event handler) to this page. This handler received the *rowset* for PERSONAL_DATA and then converted it into an XML format. The data now in XML format can be easily transported to our third party Address_Storage service. This is performed by inserting the XML data into a SOAP message destined to become the input for Address_Storage as it is sent using SOAP over HTTP.

The idea involved in making a generic design resulted in code which could be copied into any other *SavePostChange* event for any other PeopleSoft component and this feature could still have been used. Of course, the schema of the XML generated would involve some changes, but, that minimum customization cannot be worked around. This was an achievement of sorts, which lead us to believe in our integration idea approach and encourage code reusability.

A severe drawback of this method was the XML message generated for the generic Rowset was excessively verbose. The XML message was almost two megabytes in length for each person's address. The reason of course, was that the data which PeopleSoft allowed a user to change is huge and as we generated the XML message based on the rowset provided by PeopleSoft. Most of the information gathered here was useful for PeopleSoft, but, was irrelevant as far as getting geocode for the address is concerned. For example, data like phone number, email-id, marital status, etc could be edited out.

This could potentially lead to a delay in the process. As we were thinking of other possibilities, one obvious solution was to add PeopleSoft specific, People code, to the event handler and let it extract the data from the rowset. The idea was to eliminate all the unwanted data from the data provided by PeopleSoft before forming the final XML message. Fortunately, PeopleCode had XPath capabilities (XPath is a way to represent a path to an XML node in an XML document). This allowed us to navigate to the parts of the huge data to just identify data which was relevant for this exercise. So with a few lines of code we reduced the size of the message dramatically while still retaining the needed data. This dramatically reduced the size of the message.

The obvious drawback to this working solution was the loss of code reusability. If tomorrow this piece of code has to be employed in any other module of PeopleSoft, XML parsing code has to be re-written. In short, we would have to write different event handler for different data sets. Such a case-by-case customization would make the integration code less and less manageable.

This led us to rethink our strategy. We wanted to retain the code reusability we gained from the first approach. We then came up with another implementation idea. We let the event handler query the whole rowset as it is generated by the built-in PeopleSoft page, PERSONAL_DATA. Then instead of sending the rowset to the Address Storage web service, the event handler code writes the rowset to a file that was accessible by a third party utility. That is, it was written into a mutually accessible persistence area such that the web server could access it. So, now the event handler sends the URL of the file to the Address Storage service. The Address Storage service correspondingly does an HTTP GET on the URL, and obtains the complete XML file. It extracts the name and address information from the XML file, and converts the address into a latitude and longitude by looking it up from the TerraService. Then the Address Storage service stores the name and coordinates in an XML file. The main advantage of this approach is that we can always use a uniform event handler for any rowset; no message specific code is needed.

The third-party plug-in provision of PIB is depicted in Fig 5-3 which was the architecture followed for this job.

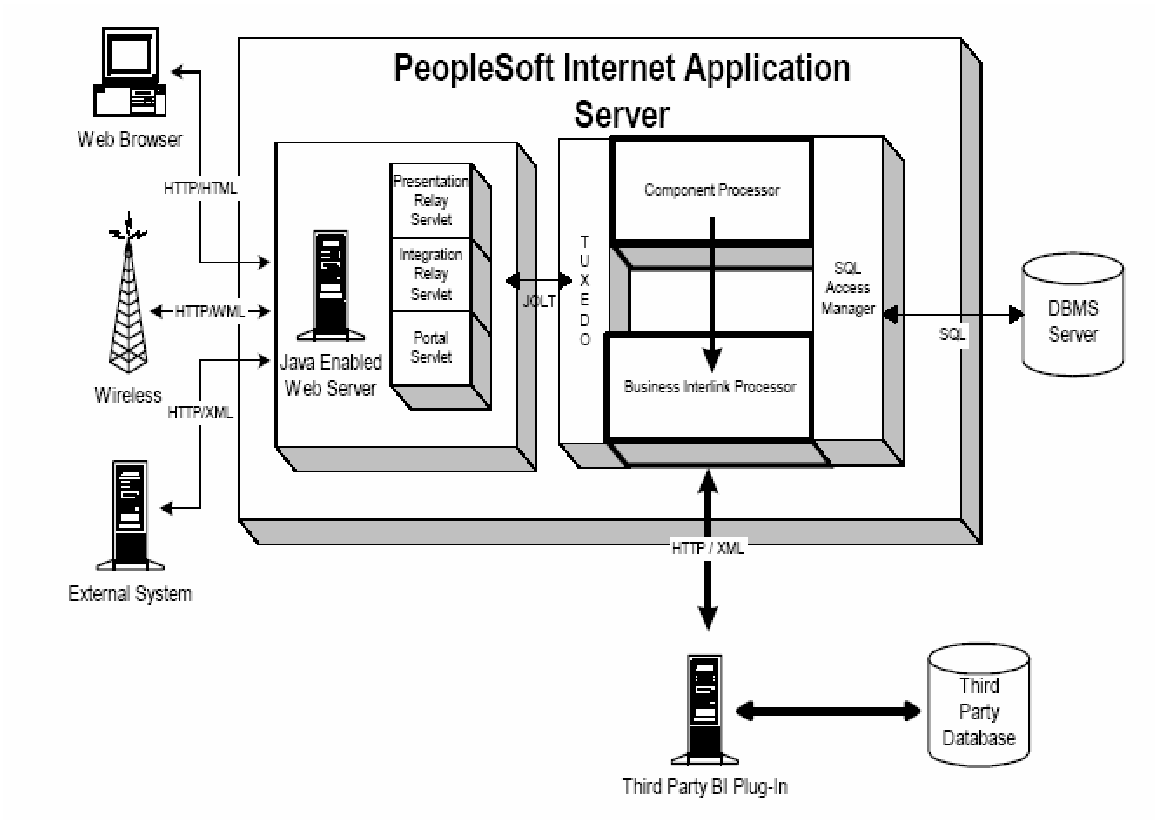

Fig 5-3: PIA integration with a third party plug-in

5.3.2 Fetch Image from TerraServer

The address obtained above then becomes the input to another Web Service – ArcSDE Service. This would provide the latitude and longitudinal values for the input address. The output object of the type *LonLatPt* would contain the geocode for the address.

The application we developed with make a synchronous call to this web service each time a user requests the map of his/her address. The flow would then make another synchronous service request to the TerraServer web service which would then return an array of 9 tiles which depict the image of the address.

The TerraService provided by Microsoft's TerraServer (www.terraserver-usa.com) has exemplified excellent design. The rich (coarsegrained) metadata approach helps clients reduce the number of requests effectively. For instance, the AreaBoundingBox object returned by the *GetAreaFromPt* service virtually satisfies all the conceivable needs for information of image maps including possible image cropping.

By assigning each data object (tile) a unique identifier (TileId) with application-level meaning (row and column), the data requests based on the metadata provided by the metadata services (such as *GetAreaFromPt*) are completely independent from the previous metadata request. With such an arrangement, the server (TerraServer) only needs to handle stateless requests, which is a key to scalability.

This power of TerraServer is tapped for the application. As stated above, we make a request to the TerraService with the LonLatPt data obtained from the first web service and eventually retrieve the final image of the address of the employ in PeopleSoft.

The service employed here has the following signature:

```
public Byte[ ] GetImageFromPt (LonLatPt center, Theme theme, Scale scale,
                                        int displayPixWidth, int displayPixHeight)
```

The service *GetImageFromPt* returns a large byte array that holds the image assembled from all the tiles surrounded by the four corner tiles. In all, there would be 9 tiles which can be assembled in the specified order to obtain the final image. A sample looks as below -

Fig 5-4: A sample image retrieved from TerraServer

The objective of developing a messaging based enterprise application which was web services based was thus achieved. This being a part of this thesis, our work eventually became part of a technical paper to get published. The complete high-level architecture of this effort is displayed below. The numbers follow the order of the flow of the application.

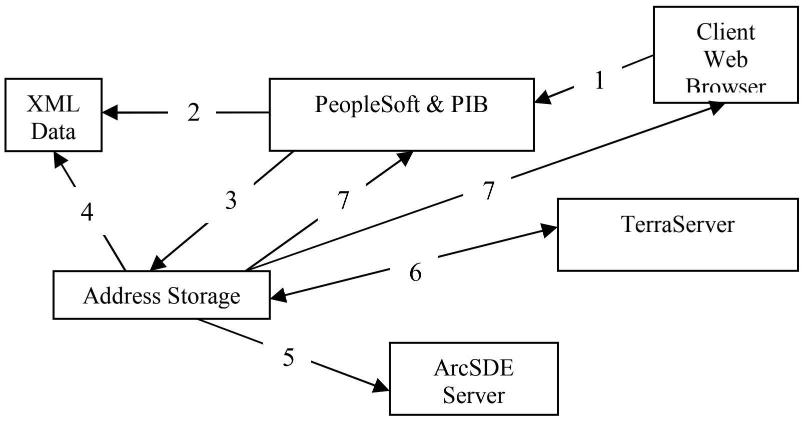

Fig 5-5: High-level Architecture

1. User uses the web browser to effect a user information change in PeopleSoft.
2. PeopleSoft writes the changed data into an exposed XML document.
3. PeopleSoft sends an asynchronous message to the Web Service – Address Storage containing information of the XML document.
4. Address Storage extracts address info from the XML file.
5. Address Storage converts the address obtained into geocode by making a synchronous call to ArcSDE Server.
6. Another synchronous request is made to TerraServer to obtain the image.
7. The image is either published directly on the Web Browser or sent to PeopleSoft using PIB.

Chapter 6 - RTS TECHNIQUE PROPOSAL

6.1 Introduction

As we observed in Chapters 3, 4 and 5, there have been various RTS techniques developed for different software systems. Most of these have been empirical formulae based proposals. Some are based on graphical representations of the system under test. Most of them address programs in C or PASCAL but not object-oriented applications. A reason could be that the research on regression testing has been primarily for real-time defense systems or other scientific computational applications, many of which were not written in object-oriented languages. Consequently, RTS techniques were also developed based on those kinds of applications.

Recent developments in Internet computing lead to the popularity of object oriented languages, not to mention Java has been a leader in this. With the flexibility Java provides, plethora of applications have been developed which are internet ready. Most of these applications are complex, multi-tier enterprise systems which are geographically deployed. While RTS techniques were proposed for some basic Java applications, no research on RTS techniques for large-scale enterprise applications has been published.

The above observation and our interests and experience in working on software integrations have been motivating us to choose enterprise applications as target "Systems Under Test" of Regression Testing. Considering that enterprise systems are in general modular and dependent on communication and messaging is the primary communicating mechanism in enterprise software integrations, I am to propose an RTS technique for testing message-based enterprise systems.

6.2 Architecture Diagram

The proposed technique is made of three phases. Each of the phases has a role to play in trimming the test suite and realize in an accurate final test case bucket which is also safe.

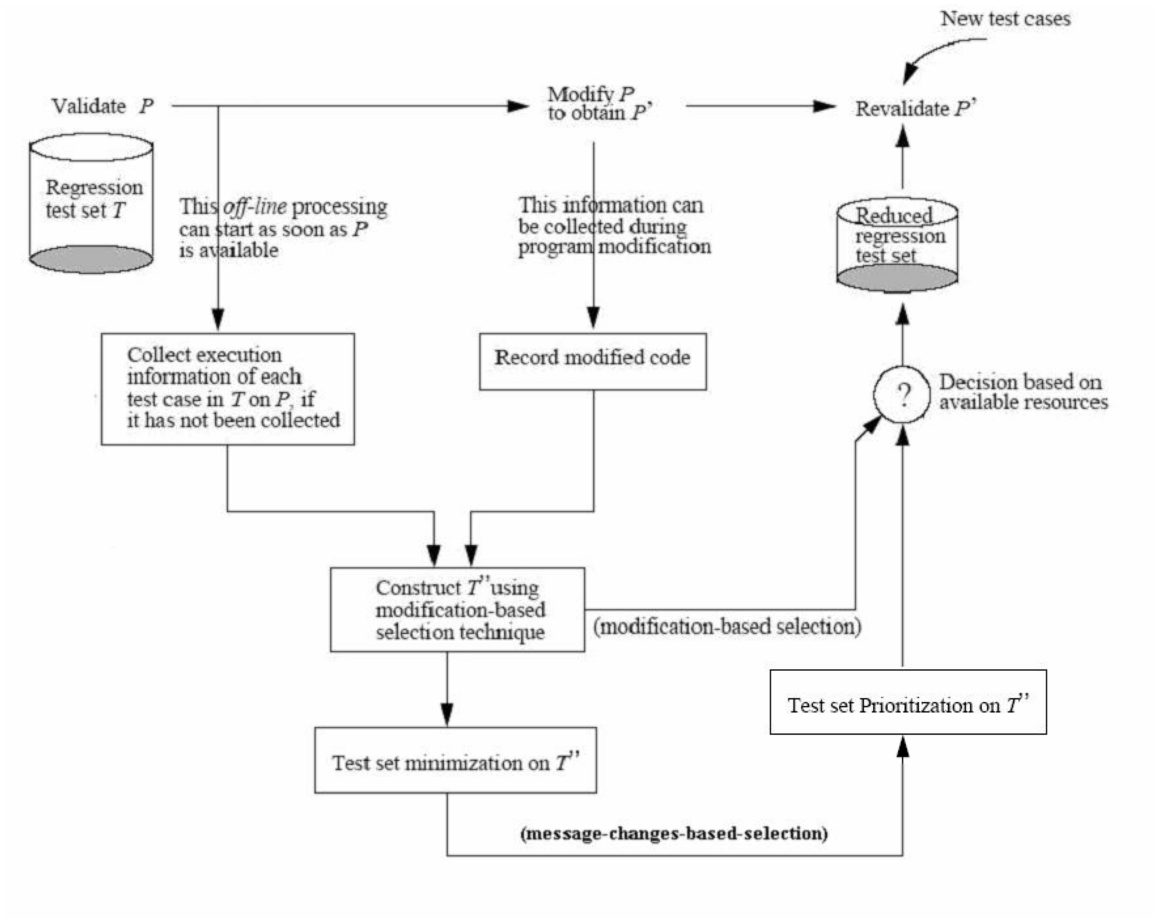
New test cases
Validate P
Modify P to obtain P'
Revalidate P'
Regression test set T
This *off-line* processing can start as soon as P is available
This information can be collected during program modification
Reduced regression test set
Collect execution information of each test case in T on P, if it has not been collected
Record modified code
?
Decision based on available resources
Construct T'' using modification-based selection technique
(modification-based selection)
Test set Prioritization on T''
Test set minimization on T''
(message-changes-based-selection)

Fig 6-1: High-level Architecture of the Technique

6.3 Assumptions of the systems and the definition of being impacted

We assume that (1) every modification of and addition to the software components is properly documented, at least including the information of the changing place as accurate as to the modules or the *method* for Java programs; (2) the messages between system components are all in XML. Any change to the schemas of these messages is properly documented, at least including the information of the changing place as accurate as to the XML *element*; (3) the system integration part is well-documented, at least including the information about the skeleton (on each server side) and the stub (on each client side) of every communication channel, where the skeletons and the stubs are the message handlers on the server and the client sides. The relationship between the messages and the code units is specified, which includes the *generation* *relationship* – Code unit X generates Message Y, and the *consuming relationship* – Code unit U consumes (uses) message V; (4) every test case is well understood based on prior conservative coverable analyses. Therefore, every test case has a known touching module set (the set of units of code); (5) no nondeterministic behavior is involved in generation and utilization of messages. That is, the same code will generate the same message upon the same state, and the same message will drive the same accepting component from the same beginning state to the same resulting state and will make this component to take the same actions.

To ease my description of the proposed RTS technique, I first define the meaning of a unit of code being *impacted*. If a unit (e.g., a module or a method) of code is modified or newly added, this unit is consider impacted. If a unit of code accepts an impacted message (to be defined shortly), this unit of code is impacted. If a unit of code accepts any data from an impacted unit, this unit is impacted.

A message becomes impacted in two ways: (1) if this message is generated by an impacted unit of code; (2) if the schema of the message is modified or newly added. For XML messages, messages consist of elements. An element becomes impacted in two ways: (1) if this element is generated by an impacted unit of code; (2) if the schema regarding to this element is modified.

6.4 Modification based Selection

Recall for any System under Test (P), the regression test set T and the modified program P', we want to find T' $\subseteq$ T such that

$$\forall t \in T,\ t \in T' \iff P'(t) \neq P(t)$$

where P(t) represents the result of running t over P. In message-based communications, running results are the outputs of P or P' because messages do not deliver any state information of P or P'.

In general, computing T' is not decidable for an arbitrary P, P' and T. In practice, one can find T' by executing P' on every regression test case. Avoiding this retest-all way is the goal of RTS.

Our technique selects test cases from T based on impacted code, that is, to find all test cases in T which execute some modified code. Let T’’ be a set consisting of these test cases. Then T’’ = {t | t executes at least one unit of impacted code}. This set is computable based on information provided according to our Assumption (4).

According to our Assumption (5), for a given test case t, if the code executed in P by t is the same as that in P’, t generates no different outputs. This derives that if P(t) ≠ P’(t), t must have executed some code in P’ that was modified or added with respect to P. That is, t∈T’’. On the other hand, since not every execution of the modified code will affect the output for that test case, there may exist some t∈T’’ such that P(t) = P’(t). Therefore, T’’⊇T’. T’’ can be used as a conservative alternative for T’. Since impacted code unit is derived from modified code unit, we refer to the process of forming T’’ as a modification-based selection. T’’ is safe respect to T.

6.5 Message-Comparison based Selection

In the following, I describe a two-phase procedure to compute T’’. This is based on comparing the messages which are exchanged.

6.5.1 Phase I – Recognize impacted units

Earlier in chapter 4, we learned about the Java Interclass Graph (JIG). A JIG accommodates the Java language features and can be used by the graph-traversal algorithm to find dangerous entities by comparing the original and modified programs.

A JIG is constructed on P’ based on the changes observed between P and P’. A JIG has a special way to represent an inter-procedural call. We expand this feature of a JIG to use for inter-subsystem when messages are exchanged.

Step 1 - Identify all the messages that have a modified or added schema. This is known according to our Assumption (2). The result of the first step is to mark every modified or added message *highly-impacted*, and mark every modified or added element in these messages *highly-impacted*.

Identify every modified or newly added code units according to the information available by our Assumption (1). Mark each of these code unit *impacted*.

Step 2 - Build a JIG for each subsystem. A JIG depicts the calling module and the called module using a path edge, and mentions the calling method's name.

Step 3 - Using these JIGs, we can identify every impacted code unit in each subsystem based on two aspects of information: (i) the code units are affected by an impacted unit according to the JIG; (ii) the code-message consuming relationship given in Assumption (3). For each identified code unit, if the upstream impacted entity (a code unit or a message) is *highly-impacted*, mark the unit *highly-impacted*, otherwise mark it *impacted*.

Step 4 - Identify more impacted messages by using the code-message generation relationship provided by our Assumption (3). If at least one new impacted message is identified in any subsystem, mark every message that has a generation relationship with any impacted code unit (if the upstream impacted entity (a code unit or a message) is *highly-impacted*, mark the unit *highly-impacted*, otherwise mark it *impacted*) and go to Step 3; otherwise the task in Phase I is finished.

Step 5 – As an additional step, at the end of this phase, we can consider every element of the message that has changed as affected. Thus, all such fields are marked impacted. This is an additional step to refine and go to a higher granular level. To specify them, we specify the called method and then mention the fields in the message that changed.

6.5.2 Phase II – Select test cases

The final step is obvious. Use the information available according to our Assumption (4); we can identify the test cases in T such that they touch at least one impacted code unit. These test cases are retained, as they may cause different outputs when executed and hence need to be retested. But, if T’’ contains no test case which touches P’, then new test cases are needed, which is a straight forward solution.

6.6 Prioritization based Selection

We observed that modifications to schema of messages in systems typically have a large impact to the system. On the other hand, those changes which are purely business logic changes and are internal to a specific component would often make no big difference. In the PeopleSoft – ArcGIS integration system demonstrated in Chapter 5; let us consider two changes as examples.

1. Let the web service Address Storage change its address reading mechanism by delegating it to a third-party object. Such a change would not really alter the very functionality of the application and hence affected test cases due to this change could be made low priority.

2. On the other hand, let us consider PeopleSoft changes the format in which the data is sent out. Peoplesoft adds some financial information along with the personal data. This change will force a change of the schema of the XML message sent out by the SavePostChange method. Consequently, this will cause changes in both the sending PIB and the web service Address Storage.

From the above two changes, it is obvious the second change will have more impact to the system and should be given a higher priority. Based on this observation, we will recommend to test the test cases that are involved with those code units with a *highly-impacted* mark. If time is allowed, then every test case in T’’ should be tested.

6.7 Web Services Specific

In a service oriented approach, like the one demonstrated in Chapter 6, this would mean, we would know the services which have changed. Hence, these services are marked as infected.

Web services, being potentially platform independent, could have multiple implementations scenarios. Based on the case study in Chapter 5, we realized that this phase can be simplified.

Typically the service and the client-side use a skeleton and a stub as their message handlers respectively. The process of identifying the impacted code units can be manifested onto the paired stub and skeleton. Based on the knowledge available according to our Assumption (3), the client side JIGs can be connected to the service side JIG via the corresponding stubs and the skeleton. Then in Phase II we do not need to consider messages, as long as no message schema is modified.

The final bucket of test cases would be all the test cases which would definitely result in a different output from the previous state of the system and they would NOT be redundant. Of course, for all newly added modules in the system, new test cases need to be added. Validation of those is beyond this work. Eventually the testing team must have enough confidence that the test suites are complete, consistent, and correct.

- Completeness: The test suites cover all the requirements of the system.
- Consistency: For the same requirements, no two test cases contradict each other.
- Correctness: The oracle for the test suites has been correctly specified.

Chapter 7 - CONCLUSION AND FUTURE WORK

This effort was focused at trying to propose a framework for the known problem of RTS for enterprise applications. The attempt was at trying to keep the scope to applications which were message intrinsic. By this we mean, systems which heavily depend on messages for data exchange and inter-component communication.

Research related to this effort is nothing, to our knowledge. This was very motivating and a huge challenge for us. We realized that starting with a grand idea would not work out and so this thesis was divided into parts.

Part I was a survey stage. We spent a lot of time surveying most of the existing techniques for RTS. This gave us a lot of information and was a very good knowledge gaining phase. Also, we realized that there is no one technique which we could easily plug in and use for enterprise systems. We realized that we would have to come up with something which would be like a hybrid, developed on top of many other techniques.

Part II was a case study development stage. At this time, we realized that before we could define our idea and formalize it, we should have a system ready. We needed a fairly complex, non-trivial enterprise application. Also, we needed it to be heavily messages driven. Another project requirement led to our interest in PeopleSoft. Upon subsequent research, we realized that this would be a good candidate to try to develop and set up a system to understand a messaging system better and experiment on an RTS technique. Considering lack of any support on this front, getting the PIB up and running even for this small requirement was an achievement in itself.

Our approach was first to study messaging based systems. In the process, our initial trials at working with web services in an asynchronous manner opened up a road block as far as WSDL is concerned. Its inability to inform about the service implementation style was an impediment. As we thought about working around it, we hit upon this idea of trying to generate another layer over the WSDL which would give the details of the Standard Pattern employed in implementing the service. The technical paper, while not published in a conference, was well acclaimed.

With that aside, we went ahead and eventually crossed all the road blocks to set-up the system as planned. While it took lot of time, the satisfying feature was that this became part of a technical paper which got accepted in an international conference, namely, 9th IEEE - International Conference on Computing and Communications. This was held in July'04 at Egypt. This was another very satisfying milestone in the thesis work.

Part III was then trying to tinker around with the various techniques learnt earlier in phase I and try to come up with a framework which would fit with the application we setup. Upon lot of deliberation the technique described in the chapter 7 was formulated. This is a detailed framework which, while might not be 100% accurate, would be very fast, as it would over-lap with the design cycle of SDLC.

As future work, we have;

- the work of actually implementing this technique on the system. We are confident once this is implemented, it would provide with the required statistical data to further strengthen our claim.
- Combining the technique with another object oriented technique to eliminate test cases which would test unchanged third party components.
- Make this platform independent by testing on non-J2EE applications.

Appendix A - References

[1] Onoma, A.K., Tsai, W., Poonawala, M.H., and H. Suganuma. Regression Testing in an Industrial Environment. Communications oftheACM, Vol. 41, No. 5., May 1998, pp. 81-85. Week Conference, San Francisco, CA., November 1997.

[2] Chen,Y. F., Rosenblum,D. S.0 and K. P Vo. TestTube: A System for Selective Regression Testing. Proc. 16 th International Conference Software Engineering, Sorrento, Italy, May 1994, pp. 211-220.

[3] Rothermel, G. and M. J. Harrold. A Safe, Efficient Regression Test Selection Technique. ACM Transactions on Software Engineering and Methodology. Vol. 6, No. 2, April 1997, pp. 173-210.

[4] Leung, H.K.N. and L. White. A Cost Model to Compare Regression Test Strategies. Proceedings of the Conference on Software Maintenance-91, 1991, pp. 201-208.

[5] Rosenblum, D.S. and E. J. Weyuker. Using Coverage Information to Predict the Cost-Effectiveness of Regression Testing Strategies. IEEE Trans. Software Eng., vol. 23, no. 3, March 1997, pp. 146-156.

[6] D. Callahan, "The program summary graph and flow-sensitive interprocedural data flow analysis," Proceedings of the SIGPLAN'88 Conference on Programming Language Design and Implementation, pp. 47-56, Atlanta, GA, June 1988.

[7] L. A. Clarke, A. Podgurski, D. Richardson, and S. Zeil, "A comparison of data flow path selection criteria," Proceedings 8th International Conference on Software Engineering, pp. 244-251, London, UK, August 1985.

[8] P. G. Frankl and E. J. Weyuker, "An applicable family of data flow testing criteria," IEEE Transactions on Software Engineering, vol. 14, no. 10, pp. 1483-1498, October 1988.

[9] K. Cooper and K. Kennedy, "Interprocedural sideeffect analysis in linear time," Proceedings of the SIGPLAN'88 Conference on Programming Language Design and Implementation, pp. 57-66, Atlanta, GA, June, 1988.

[10] E. W. Myers, "A precise inter-procedural data flow algorithm," Conference Record of the Eighth Annual ACM Symposium on Principles of Programming Languages, pp. 219-230, Williamsburg, VA, January 1981.

[12] M. J. Harrold, "An approach to incremental testing," Technical Report 89-l Department of Computer Science, University of Pittsburgh, January 1989.

[13] Steve Graham, Simeon Simeonov, Toufic Boubez, Doug Davis, Glen Daniels, Yuichi Nakamura, Ryo Neyama, Building Web Services with Java: Making Sense of XML, SOAP, WSDL, and UDDI, 2002. Sams Publishing, Indianapolis.

[14] K. Gottschalk, S. Graham, H. Kreger, and J. Snell, Introduction to Web Services Architecture. http://www.research.ibm.com/journal/sj/412/gottschalk.html.

[15] PeoplSoft Integration Broker based on Web Services.

[16] Francisco Curbera, Matthew Duftler, Rania Khalaf, William Nagy, Nirmal Mukhi, and Sanjiva Weerawarana. Unraveling the Web Services Web: An Introduction to SOAP, WSDL, and UDDI.

[17] Hsia P., Li X., Kung D.C., Hsu C-T, Li L., Toyoshima Y., and Chen C. 1997. A Technique for the Selective Revalidation of OO Software. Software Maintenance: Research and Practice, Vol. 9, 1997, 217-233.

[18] Daou B., Haraty R.A., Mansour N., Regression Testing of Database Applications

[19] ISO/IEC 9075: 1992. Information Technology – Database Languages – SQL.

[20] Leung, H.K.N., White, L., 1992. A Firewall Concept for both Control-Flow and Data-Flow in Regression Integration Testing. Proceeding of International Conference on Software Maintenance, 262-271.

[21] Leung, H.K.N., and White, L. 1990a. A Study of Integration Testing and Software Regression at the Integration Level. Proceeding of International Conference on Software Maintenance, 290-300.

[22] Harrold M.J., Jones J.A., Li T., Liang D., Orso A., Pennings M., Sinha S., Spoon S.A., Gujarathi A., Regression Test Selection for Java Software.

[23] G. Rothermel, M. J. Harrold, and J. Dedhia. Regression test selection for C++ software. Journal of Software Testing, Verification, and Reliability, 10(6):77-109, Jun. 2000.

[24] J. Bible and G. Rothermel. A unifying framework supporting the analysis and development of safe regression test selection techniques. Technical Report 99-60-11, Oregon State University, Dec. 1999.

[25] D. Liang, M. Pennings, and M. J. Harrold. Extending and evaluating flow-insensitive and context-insensitive points-to analyses for java. In Proceedings of the ACM
Workshop on Program Analyses for Software Tools and Engineering, Jun. 2001.

[26] Memon A.M., Soffa M.L., Regression Testing for GUI(s).

[27] PeopleSoft Internet Architecture & Web Services Support

[28] S. Tu, E. Normand, S. Kuchimanchi, V. Bizot, S. Shu, M. Abdelguerfi, J. Ratcliff, and K. Shaw, "Integrating Web Services into Map Image Applications", Proceedings of the Ninth IEEESymposium on Computers and Communications (ISCC 2004), Alexandria, Egypt,pp 44-49, June 29-July 1, 2004

VITA

Sriram Kuchimanchi was born in Hyderabad, India in 1978. He earned a Bachelor of Engineering Degree in Information Science & Engineering during his study at Bangalore University, India from 1995 to 1999. He worked in the software industry as a designer and developer for 3 and a half years working on various web technologies between July 1999 and December 2002.

He was enrolled in the graduate program in Computer Science Department at the University of New Orleans in January 2003. He worked on a research project under the guidance of Dr. Shengru Tu in the Department of Computer Science as a software architect from August 2003 till November 2004. During this time, he worked on a few technical papers and was a co-author for a paper published in an *IEEE symposium*. He was also a teaching assistant in the department, teaching under-graduate students Java from January 2003 till May 2004. He has great interest and rich experience in developing Web Services based enterprise applications. He also has tremendous interest in teaching.