B

MODELING, TESTING, AND REGRESSION TESTING OF WEB APPLICATIONS

by

Hamzeh k. Al Shaar

A thesis submitted in partial fulfillment of the
requirements for the degree of

MS Computer Science

Lebanese American University

2006

LEBANESE AMERICAN UNIVERSITY

School of Arts and Sciences - Beirut Campus

Student Name:Hamzeh K. Al Shaar I.D. #: 200302499

Thesis Title : Modeling, Testing and Regression Testing of Web Applications.

Program : Computer Science

Division/Dept : Computer Science and Mathematics

School : **School of Arts and Sciences**

Approved by: Ramzi A. Haraty

Thesis Advisor: [redacted]

Member : Faisal Abu Khzam

Member : Sanaa Sharafeddine

[redacted]

Date April 26, 2006

I grant to the LEBANESE AMERICAN UNIVERSITY the right to use this work, irrespective of any copyright, for the University's own purpose without cost to the University or to its students, agents and employees. I further agree that the University may reproduce and provide single copies of the work, in any format other than in or from microforms, to the public for the cost of reproduction.

[redacted]

LEBANESE AMERICAN UNIVERSITY

ABSTRACT

MODELING, TESTING, AND REGRESSION TESTING OF WEB APPLICATIONS

by Hamzeh k. Al Shaar

Chairperson of the Supervisory Committee: Professor Ramzi Haraty
Department of Computer Science

Web Applications became at the heart of the business world. All corporate companies and big institutions have very busy e-commerce web sites that host a major part of their businesses. With this great emerge of web applications, techniques for maintaining their high quality attributes should be developed and exercised. More over, the quick evolution of web technology and the high user demands made the web applications subject to rapid maintenance and change which requires the development of efficient regression testing techniques. I researched lots of work done in the field of web application testing and regression testing and I found that each of those papers dealt with a specific part of the web application. While some papers model and test the server side programs of the application, others model and analyze the navigation between pages as seen by the user and yet others dealt with analyzing the architectural environment of the web application. Motivated by the fact that there is no single model to represent the entire web applications and to model it from different perspectives at the same time, I decided to propose a single analysis model which models the three poles of the web application: the client side pages navigated by the user, the server side programs executed at runtime, and the architectural environment hosting the application. Based on this model I am going to propose testing and regression testing techniques for each of the three parts of the model. Having discovered, as well, that there is no automated black box regression testing technique, I am going to propose a methodology and algorithm to create a tool capable of applying black box regression testing automatically.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

This research would not have been possible without the help and assistance of many persons.

First I would like to express my gratitude to my supervisor Dr. Haraty and the Committee members Dr. Abu Khuzam and Dr. Sharaffeddine to whom I am particularly indebted for their substantial aid, advice, and assistance from the beginning to the end of my study.

I extend my gratitude to all the respondents who contributed in providing data to this research, and to the authors of the published papers whose work paved the way for this research and whose ideas constituted the basis for this research.

Finally, special thanks go also to all my friends, colleagues at work and family who valued the benefits of a good education and supported my academic endeavors.

Chapter 1

INTRODUCTION

1.1 Overview

As the web is growing and invading our world, and as using the internet became a normal habit to the new generation, and with the huge number of services available on the web offering business to business (B2B), business to customer (B2C) and customer to customer services (C2C), web applications started to take a major part in the software industry and began to invade the business market playing an important role in facilitating the business flow of most companies. Major companies world wide depend fully on their e-commerce web sites whose availability and correctness are vital to the continuity of their operations. Moreover, the web sites represent the companies' image to their customers especially those over seas, and thus the quality of service offered by the web site reflects the level of professionalism of the company and eventually any problems with the sites will directly affect the overall image of the company and will affect the trust of its customers. For example, a glitch which happened at amazon.com during a non-scheduled maintenance in 1998 brought the web site down for just few hours and it resulted in around $400, 000 lost for the company. In addition to the correctness of the applications we have to take into consideration that web applications will be used by untrained users who might be misusing the system and who are accessing the web site concurrently and around the clock from different geographic locations and whose number is subject to uncontrolled increase at any moment. All this requires our high attention to analyze the environment hosting our application in terms of scalability, compatibility, and availability.

1.2 Need for the Study

Thus with this emerging importance of web applications in the commercial sector, and with the new and challenging quality requirements of web software, techniques to test and to control their quality attributes became a must. However, developing such testing techniques for web applications is much more complicated than that of classical software and this is due to the nature of the web applications. One of those complications is the quick evolution of the web technologies and another is the extreme heterogeneous architectures on the server, the client and the communication levels. In addition to the above it is important to note that the growth in the number of users results in an increase in the user demands and thus leads to an increase in the maintenance and upgrade requirements for the web application. As a result, efficient regression testing techniques should be developed as well in order to accommodate this requirement of fast software maintenance.

Unfortunately there is no well developed and mature model to analyze and test web applications yet. All the work previously made dealt with certain aspects of the web applications while neglecting the rest. Some papers analyzed the server side components of the web applications while neglecting the navigational flow of execution between pages as seen by the end user. Others discussed the navigation between pages but never analyzed the server side programs in depth. Some other papers modeled and analyzed the architectural environment in terms of compatibility and scalability but never tested the application it self. Even worse, very few testing and regression testing techniques have been exploited to be used by web applications. In addition to the above mentioned deficiencies, no effort has been done at all to automate black box regression testing which is very important and time saving since it spares the tester repeating manually all his test cases following a small change. This was the motivation which made me propose one complete analyses model with its testing and regression testing techniques and which represents and analyzes: 1) the architecture of the environment hosting our application, 2) the navigation between pages, and 3) the logic of the server side components and dynamic pages. Along this three poled model, I will be proposing a detailed methodology and algorithm for creating an automated tool for black box regression testing which re-executes the relevant test cases and, automatically, compares the results and presents the tester with the differences to analyze.

1.3 Structure of the Thesis

In this thesis, I will first go in chapter two through background related work; second I will be discussing in chapter three the architectural environment model and its testing techniques. After that I will be discussing in chapter four the client side modeling of web applications and its testing techniques followed by a discussion in chapter five about the server side programs model and its testing techniques. Finally, I will be proposing in chapter six a methodology and an algorithm for developing a tool for automating black box regression testing for web applications. Before concluding my thesis in chapter 8, I will be presenting a case study in chapter seven on a real web application developed by me, where I will be modeling it as per the proposed model and I will be applying to it all the testing techniques discussed.

Chapter 2

LITERATURE REVIEW

In this chapter, I will be looking at some of the previous work done in the field of web application modeling and testing. Moreover I will be looking at some popular work achieved in the field of software testing and regression testing.

Wu an Offut [1] presented us with an analyses model for modeling the server side components and the client server interactions. This technique presented us with a very useful method to partition the server side components and to model the sequence of interactions between those partitions themselves and with the client HTML pages. Even though the model covers the interactions between client pages and between client and server pages, but this was done briefly and without much detail. As a continuation of their previous work, Wu Offut and Duz developed further the model presented in [1]. The extended model [2] covered inter-component connections between different server side components. The new model covers as well the current state representation of web application where the current state of variables is recorded as well. The new model is represented in a graph which makes it easier to track and to see different dependencies.

Ricca and Tonnela [3] modeled the web application as a sequence of web pages and they modeled the interaction between them in a UML diagram. There work purely focuses on the client side HTML pages without any consideration of server side scripts. The proposed model presents different parts of a web application from complete pages, to forms, to frames and to input fields and it presents the interactions between them. The two authors have another paper about web application slicing [4]. In this work, the authors extended the concept of application slicing and applied it to web applications. Application slicing is a well known technique in software testing and adopting it to web pages is a big step in the direction of web applications testing. An application slice is a functional part of a big program which when stripped and executed alone will produce the same functionality as when it was within the program. The model proposed is based on the UML model presented in the previous paper [3] and thus it deals with client side pages and with definition-use variables only without any attention to the server side components.

Sebatien, Karre, and Rotherm [5] presented a technique for automatically testing the web application based on the user session data collected from the user's navigation of the website. The proposed technique simply collects all the URLS requested by the users from the web servers. Then those requests, which are grouped in sessions, are transformed into test cases repeated on the same application in order to test changes o to validate the behavior of the application under different conditions. A similar work has been proposed by Wen [7]. In his paper, he proposes a technique for generating test cases based on URLs to be automatically exercised. This model is a bit similar to that proposed by Sebatien, Karre, and Rotherm [5] but it is not as well developed as that one. This model simply proposes that test cases or URLS are stored in a database to be sequentially executed later while collecting their output for analyses. Suganuma and Nakarume [10] presented a paper which describes an improvement to an existing testing tool they previously developed. This tool is a basic record/replay tool which is used for repeating test cases automatically and in a quick way.

The main idea from those papers [5], [7], and [10] has been adopted by me in proposing a technique for automated black box regression testing. However the ideas presented in those three papers [5] [7] and [10] are immature and are only headlines which I have developed and enhanced remarkably to fit my needs.

Sneed [6] wrote a paper which focuses on the web application architecture and which recommends that this architectural environment should be tested independently of the web application itself. The author highlighted major architectural issues that need to be tested and analyzed. Such issues include inter-server communication (web, application, and database servers), load testing, transaction management, and analyzes of the underlying software platforms.

Many papers have discussed regression testing of classical software such as the work done by Xu, Yan, and Li [8] and the work done by Granja and Jino [9]. Even though those papers do not deal with regression testing of web applications but they contain lots of valuable ideas that I have built upon them in developing regression testing techniques for my models. For example, the authors in [9] discussed the means to select test cases to exercise on possibly modified functionalities in the program. In fact the issue of selecting the re-testable test cases is the most important issue in regression testing.

Xu, Chen, and yang [11] made a paper which discusses regression testing of web applications (using slicing). The authors suggest that web slicing is used and is exploited later for regression testing of web applications. The author goes along the same line of web slicing discussed previously by Ricca and Tonnella [4], but he does not construct the web UML model because it is too expensive. He uses a different technique where he associates with each page some variables describing the number of links in and the number of links out of this page.

Chapter 3

THE ARCITECTURAL ENVIRONEMNT MODEL

3.1 Overview

In this section I am going to explain what exactly I mean by the architectural environment. I will be stating in detail what parts should be modeled and what kind of tests should be conducted and their importance. Then I am going to propose a graphical analytical model and I will be proposing a set of testing and analysis techniques based on this model. Testing of this model is more oriented into analysis and evaluation of the environment rather than checking of the correctness since it is very possible to have different architectures for one application but each with certain advantages and disadvantages.

Modeling the architectural environment of the web application includes representing:

- All physical servers participating in hosting the applications (hardware and operating system)
- All Software Servers running on top of the physical servers (like IIS, Apache, SQL Server…)
- The communication protocols between connected nodes of servers
- The kind of messages exchanged between servers and the underlying programming language
- Redundancy, balancing, and scalability of servers

The model should be able to present a complete overview of the architecture in question and should make it easy for an analyst to determine any information needed regarding the operational environment.

3.2 Quality attributes

Since we do not deal with programs or with lines of code on this level, testing is more into specifying the quality attributes of the underlying system. This is done by evaluating the reliability of the system (as a whole) and checking the level of integrity between its different parts.

The quality attributes of the operational environment are determined by:

3.2.1 Compatibility of Each Server

Since our web application will be running on a set of connected nodes (servers), we should make sure that each server by itself is robust and running compatible operating system and application software because having a single unstable node in our architecture will cause the whole system to be unstable. At this level we should take each server separately and we have to check the compatibility of:

- Hardware with the Operating System
- Operating System with the installed Software servers

The first level of checking should be to determine the compatibility between the Operating system and the Hardware in question. For example, an installation of Microsoft Windows Server on an Intel based server is a typical and a very well known installation. Moreover an IBM AS/400 machine with the IBM OS/400 installed on it is known to be one of the most robust installations. However, it is impossible to install OS/400 on an Intel based server or to install Microsoft Windows Server on an AS/400 machine. Another example would be running Unix OS on Intel based servers. While it is impossible with some versions of UNIX, it works very well with some versions, and it is not recommended with others (even though it runs).

With the big number of hardware manufacturers, the evolution of hardware technologies, and the growing number of operating systems (especially open source) such compatibility check became a must and is required in the early stages of planning and designing the application.

The second level of compatibility checks determines the compatibility of the installed software servers (such as application servers, web servers, database servers …) with the underlying operating system. For example it is typical to Install Microsoft IIS Web server and Microsoft SQL Server on top of Microsoft Windows Server, while it is impossible to install them on UNIX or Linux Operating systems. On the other hand, we can install MySql Database server on both Windows and Linux servers while it is more recommended to run it on Linux. Another example is IBM Websphere application server which runs on Windows, UNIX, Linux and OS/400 very well.

The existence of different types of software servers involved in hosting web applications and the vast number of brands of each type make selecting the application servers one of the most difficult tasks in designing the architecture of the system. Each of the servers has different advantages and drawbacks, but selecting the best of a type does not work all the time due to compatibility problems. Thus this type of compatibility checks is necessary as well.

3.2.2 The Communication Feasibility Between Servers

Since many servers are involved in hosting the web application, those servers should be able to communicate efficiently and smoothly as per the requirements of the applications. The need for efficient communication requires that all communication channels in our system be verified and tested to be feasible and efficient. The communication testing includes testing the feasibility and/or efficiency of the following:

- Communication protocols between servers
- Connection libraries between the applications
- The exchanged messages between servers

- The Verification of the communication protocols between the servers is to verify whether the designated servers are ready to talk with each other on the desired protocol (regardless of what is being sent or received). For example suppose we have a web page (hosted on the web server) which is used to upload attachments. Suppose that the application then moves the uploaded document to a file server via the FTP protocol. In this case we are required to check whether it is possible to establish an FTP connection between the two servers (because we may have firewalls and network restrictions) or whether the file server is capable of accepting FTP request in the first place.

- The verification of the connection libraries checks whether the application servers have the required libraries and extensions to communicate regarding that communication on the lower protocol level is feasible. For example, if we have JSP page on the web server that needs to connect to a Microsoft SQL Server via ODBC connection, then we need to have JDBC_ODBC Bridge (library) installed on the web server. This library enables the Java classes to communicate with the database over TCP/IP protocol.

- The analysis of the exchanged messages between the servers is mainly done for efficiency study and evaluation. Well there is not much to test in this area, but presenting this in our model is necessary for completeness and it is important to analyze the efficiency of our application and to understand the dataflow between different servers.

3.2.3 The Level of Redundancy and Load Balancing

Availability and quick response of the system is one of the most important quality attributes of web applications since users with different time zones access the application around the clock and they need their information instantly or else they will get bored and try some other (competitor) site. Having too many servers involved in the environment hosting the web application increases the single points of failure and eventually increases the risk of having the application non operational if one of the servers fail. As a result it is very important to have server clustering (redundancy) on the major servers in order to achieve high availability for our application. Testing the level of redundancy basically checks how many single points of failure we have in the system and how risky they are.

Another important point is load balancing. Since web users can increase in number remarkably and unexpectedly, we might have a situation where some server (or more) will be unable to handle the entire load by itself and would create a bottle neck for the system. In this case a need for another server to assist the first in handling the load is needed. A typical example of load balancing is to have a farm of web servers receiving the client requests and passing them to backend servers. Another example is to have a grid of database servers hosting a huge database for the application. Testing the level of load balancing is important since it can determine the possibility of having performance bottle necks on the system which results in a slow and inefficient application. We can test the level of load balancing by calculating the number of load balanced nodes versus the total number of nodes we have.

3.2.4 The Level of Scalability

The number of users for web applications increases remarkably soon after the web application goes online and the web site is known to a huge audience. As a result, the ability to scale the architecture with the growth in the number of users is a very important feature that should be taken into consideration. This quality attributes should be tested and analyzed to see how scalable our architecture is. What I mean by scaling a server is adding another physical server that will assist the original one(s) in hosting some application. An example is adding servers running Microsoft IIS to a web server farm of similar server or adding an additional server to a grid of servers hosting an Oracle database. Basically testing this attribute would be by calculating the number of scalable servers as opposed to the total number of involved servers.

3.3 The Model

3.3.1 Description

The model represents the architectural layer in a diagram similar to a UML diagram. The diagram represents nodes of physical servers as rectangles named with the Computer Name of the server. I will be representing the servers by a set of triplets S, where each triplet contains the server name, the operating system installed on it, and the processor type of the machine (like intelx86, SPARC...). The rectangular box includes one or multiple squares, each square representing the software server installed on the machine such as IIS or Oracle Database Server or IBM Websphere.

The set of software servers will be represented by a set SV of triplets where each triplet has the software server name (we name it for ease of reference), the software server installed and the physical server name.

If two software servers are clustered for redundancy then we represent this as 2 parallel dashed lines connecting the 2 boxes (not arrows). If the 2 software servers are load balanced then the two boxes are connected with 2 parallel non dashed lines (not arrows). Normally if two servers are load balanced then they are automatically clustered for redundancy so we present them as load balanced servers only and not as both.

We will have clusters represented as a set C of pairs where each pair contains the names of the software servers being clustered. In case we have more than 2 servers in the cluster, then we have the first and the second in the first pair, the second and the third in the second pair and so on. In other words those pairs are transitive. Similarly we will have a set LB of pairs containing the load balanced servers.

The communication between servers is represented in the diagram by arrows between software servers (squares). If a server sends data to another server then we have an arrow from the first server to the second. If the second server returns data then we will have another arrow in the opposite direction. Note here that low level TCP/IP messages of handshaking and network negotiation are not taken into consideration. But rather we are talking about data relevant to our application such as database queries, data sets, files...

Each of the communication links between the software servers are presented as a set of quadruples Cmi where each quadruple contain the source software server name (member of set SV), the destination software server (as defined by set SV), the underlying protocol being used (such as FTP, Http, or SSL), and the type of messages sent from the source server to the destination servers. If the source or destination servers are members of a grid or a cluster we can mention the name of any of the cluster members instead.

The type of exchanged messages is predefined and can be on of the following:

- http_rq (http request)
- http_rs (http response)
- db_q (database query)
- db_rs (database result set)
- f (file transfer)
- xml (XML file or XML messages)
- SL (packets sent over a direct socket layer opened from the application)

Finally the software libraries and application extensions are represented as set LR of pairs where each pair has the server name (as represented in S) and the name of the communication library, driver, or extension installed.

3.3.2 Example

Let's take an example where we have 2 web servers running Microsoft IIS on Windows server 2003 and those 2 servers are clustered for redundancy. More over the serves run an ASP application that reads and writes data from an Oracle database. The Oracle database is running on a grid of three windows servers load balanced and redundant. Moreover let's assume that the Asp application processes the data read from the application and then formats it in a certain file and sends it to an ftp server. The FTP server is on a remote site (not on the LAN) and it is a normal non clustered server. For sure we have on both web servers the Oracle Drivers, and we have on each of the web servers a library which supports FTP commands and integrated within IIS.

The diagram of the architecture will look as follows (Fig. 3.1):

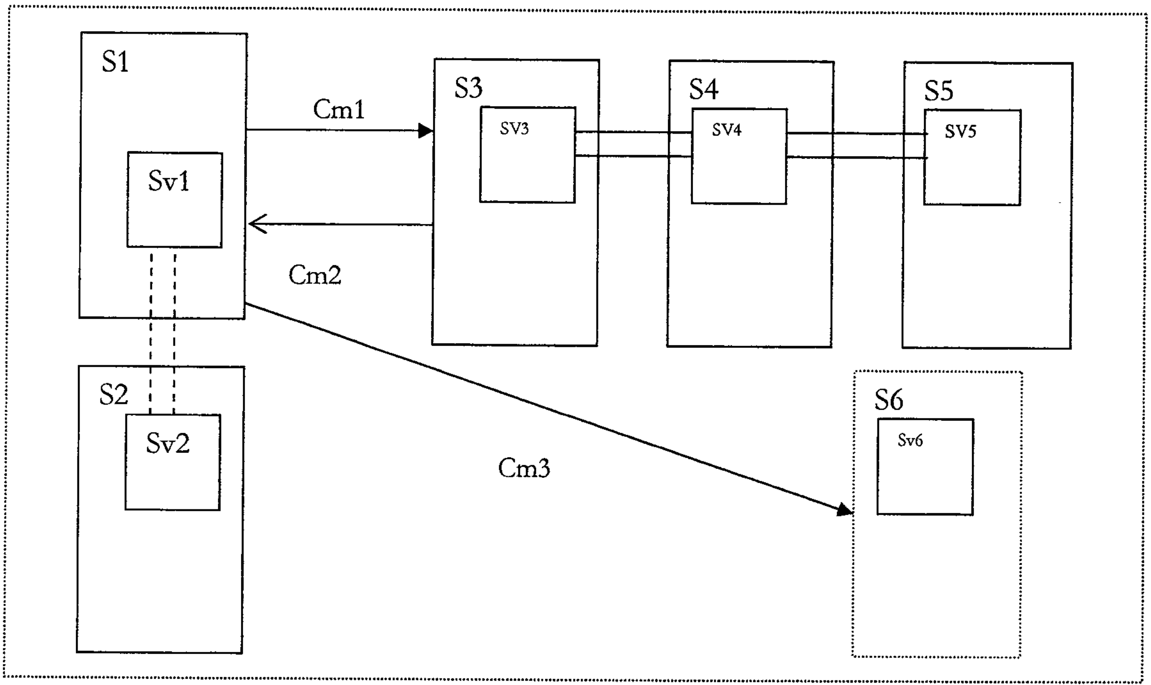


Fig 3.1 - Example of Architectural Environment Model

S= {(S1; Windows 2003, intel_x86), (S2; Windows 2003, intel_x86), (S3; Windows 2003, intel_x86), (S4; Windows 2003, intel_x86), (S5; Windows 2003, intel_x86), (S6; Windows 2003, intel_x86)}

SV= {(sv1; IIS; S1), (sv2; IIS; S2), (sv3; Oracle Db; S3), (sv4; Oracle Db; S4), (sv5; Oracle Db; S5), (sv6; MS FTP; S6)}

C= {(Sv1; Sv2)}

LB= {(Sv3; Sv4), (Sv4; Sv5)}
Cm1= {Sv1, Sv3, TCP/IP, db_q}
Cm2= {Sv3, Sv1, TCP/IP, db_rs}
Cm3= {Sv1, Sv6, FTP, file}

L1= {(S1; Oracle_driver), (S2; Oracle_driver), (S1; FTP_Library), (S2; FTP_Library), (S6; FTP_Listener)

3.4 Testing Techniques

3.4.1 Overview

The tests for the operational environment are as specified above in the explanation and here is a list of the quality attribute tests that should be done:

1- Compatibility of the operating system with the hardware
2- Compatibility of operating system with installed software servers
3- Compatibility of communication protocols
4- Compatibility of application communication libraries
5- Analysis of the messages exchanged
6- Level of redundancy
7- Level of load balancing
8- Level of scalability

Tests 1, 2, 3, and 4 are critical and they should succeed in order to have a running application. Tests 5 through 8 are not critical though important. In other words the application would still work successfully if the tests results didn't score high but it is not very recommended.

3.4.2 OS / HW Compatibility Test

The operating system and hardware compatibility test (OS / HW) is a mandatory test which our architectural environment must pass. The test is represented by the set T1 of pairs.

Each pair has the server name (as in the set S) and the values 0 or 1 paired with it. The value indicates whether the operating system mentioned in the triplet of the server in question from S is compatible with the processor type mentioned in the same triplet. In case the resulting set T has all server names paired with the value 1 then our test has succeeded. In case any server has the other pair value equals to zero then this server is not compatible and should be revised.

In our above example T1= {(S1; 1), (S2; 1), (S3; 1), (S4; 1), (S5; 1), (S6; 1)} so our architecture passes the first test.

3.4.3 OS / SW Compatibility Test

The operating system and software servers compatibility test (OS/SW) is mandatory as well and our designed architecture should pass it or else the whole architecture is rejected and classified as unacceptable. The second test is represented by the set T2 of pairs.

Each pair has the software server name (as mentioned in the set SV) and the result of the test for this triplet who is either zero or one. The test checks if the installed application server (second part of the triplet from the set SV) is compatible with the OS of the server installed on it (the third part of the same triplet). Here we have to refer to the set S to know the OS of the server in question.

All resulting pairs should have the value of 1 paired with the software server. If any pair has the value of zero then the architecture is not acceptable and needs to be revised. In our example above the set T2 looks like:

T2= {(sv1; 1), (sv2; 1), (sv3; 1), (sv4; 1), (sv5; 1), (sv6; 1)}

For the sake of demonstration suppose we have in S the triplet
(S1; Sun Solaris; SPARC)

And in SV the triplet (sv1; IIS; S1)
then T2 will have the pair (sv1;0) which means that the architecture is not acceptable because it is not possible to install IIS on a Unix server even though the Solaris Os is compatible with the SPARC processor.

3.4.4 Communication Protocols Test

The communication protocols test is mandatory to succeed as the previous two. The test is represented as a set T3 of pairs.

The pair contains the name of the connection Ci and the value of the feasibility of this connection on the protocol level mentioned as the 4th part in the quadruple representing Ci. If this connection is possible between the two pairs then the connection name will be paired with the value 1 else it will be paired with the value 0.

In our example the two connections are C1 and C2.

C1 is a TCP/IP connection between the web server and the database server and since both servers support TCP/IP then the connection is feasible.

C2 is an FTP connection between the web server and an FTP server and thus it is a feasible connection since the FTP server has MS FTP server installed on it (assuming that all required extensions and libraries exist on the web servers).

Thus for our example T3= {(C1; 1), (C2; 1)}

If we assume that the server sv6 doesn't support FTP access then we would have the pair (C2; 0) in T3 instead of (C2; 1)

3.4.5 Communication Libraries Test

The communication libraries compatibility test tests the communication feasibility on the application layer level and not on the protocol levels. This test checks for all the needed additional extensions and libraries that do not exist by default and that are required for the communication between servers. Obviously, the success of this test is mandatory as well since even if all the previous tests succeed and this one fails, our application would not be able to run successfully.

This test is represented as a set T4 of the required extensions/libraries and their existence. Thus T4 is a set of triplets where each triplet contains the server name (as in set S), the required library/extension, and the existence value which is one (for exists) or zero (for does not exist). The value of the existence of the libraries is calculated by checking whether the first two parts of the triplet in question exists as a pair in the set LR. If such pair exists then the third part of our triple (value) is 1 else it is zero.
This test succeeds if all triplets have value 1 as their third part.

In the above example T4 would look like:
T4= {(S1; Oracle_driver, 1), (S2; Oracle_driver, 1), (S1; FTP_Library, 1), (S2; FTP_Library, 1), (S6; FTP_Listener, 1)}

for the sake of demonstration, if we do not have oracle driver libraries installed on S2 then the pair (S2;Oracle_driver) does not exist in set LR and thus T4 will have the triplet (S2;Oracle_driver;0) which will result in the failure of this test making our architecture is not feasible and should be revised.

3.4.6 Analysis of the Exchanged Messages

Analysis of the exchanged messages is a non mandatory test which evaluates the efficiency of the communication and the messages exchanged between the servers. In this test we take each communication channel and we evaluate its efficiency taking into consideration the kind of messages, the underlying protocol, and the frequency of use. The value of this evaluation is subjective and it is scaled between 1 and 10 (where 10 is the most optimal). We can represent this test as a set T5 of pairs where each pair has the connection name (Ci) and the value of the evaluation. Roughly speaking it is preferred that all connections be paired with values greater than five.

In our example, if we analyze the frequency of uploading files and the speed of transferring files over FTP to a remote server, then we can assume that the connection C3 has an efficiency of 7 over 10. Sending a query over TCP/IP to an oracle server is very efficient, thus we can assume to have the pair (C1, 9). Similarly retrieving the result set over a TCP/IP connection is relatively efficient (depending on the record set size and frequency of use) so we assume that it has an efficiency of 7. Following our assumed calculation which can be done very thoroughly if need to we will have the set
T5 = {(C1; 9), (C2; 7), (C3; 7)}

Since all connections are paired with values greater than five, then all connections are relatively efficient. If we have any connection with lower values, then they should be revised to avoid performance degradation.

3.4.7 Level of Redundancy

The level of redundancy test provides us with values about the level of redundancy provided by the current architecture. Full redundancy is not necessary for the application to run but it is highly recommended to have all servers clustered to ensure high availability of our application. It is important to keep in mind that all servers that are load-balanced are redundant by default so we should take those into consideration. An easy and straight forward way to calculate redundancy is as follows. First, prepare SV' as the set of all none clustered (and none load-balanced) servers. This set is derived from SV by removing from SV all servers that are part of a pair in the set C or the set LB. Then, we calculate the redundancy by subtracting the ratio of the cardinality of SV' over the cardinality of SV from one. Generally speaking an ideal result equals one.

So *rd= 1 – (|SV'|/|SV|)*

In our example SV' = {sv6} so the value of redundancy = 1 – 1/6 = 5/6 which is relatively good.

3.4.8 Level of Load Balancing

The level of load balancing test, checks the load balancing between the servers. This test is very similar to the previous one. We create the set SV" in a very similar way to SV' but it is derived from SV differently. It eliminates from SV only those servers that exist in any pair in the set LB. We calculate the load balancing by subtracting the ratio of the cardinality of SV'' over the cardinality of SV from one. Generally speaking the optimal result equals one.

In our example SV" = {sv1, sv2, sv6}
So ldb= 1 – (|SV"|/|SV|) = 1 – 3/6 = ½ which is on average good.

3.4.9 Level of Scalability

The level of scalability test measures the level of scalability in our architecture. This test, as well, is not mandatory to succeed for our application to run, but it is preferred to have good scalability on the architectural level to ensure the possibility to handle additional number of users in the future. Normally a typical scalable architecture is where all servers are capable of being scaled and expanded to additional servers. We do not take into consideration the ability to upgrade existing servers in terms of increasing storage or memory since this is out of our scope. We rather consider the underlying technology on each server and the possibility to expand it on more additional servers. The easiest way to perform this test is to measure the ratio of the servers capable of being scaled to the total number of servers. As in the previous tests, a value of one is optimal. So we define a set S' which is a

subset of the set S and which contains the names of all servers that can be scaled. We define scalability as the cardinality of the set S' over the cardinality of the set S;
So Scalability scl=|S'|/|S|.

In our example, IIS servers can be easily scaled by adding them to a web farm that can hold tens of web servers. More over, we can scale the Oracle Database grid easily by adding nodes to the existing DB grid. However the FTP server is not scalable because it is running a simple FTP server application (MS FTP) which is a stand alone application.

So the set S'= {S1, S2, S3, S4, S5} and scl = 5/6 which is a good value since it is near to one.

Chapter 4

THE CLIENT SIDE NAVIGATIONAL MODEL

4.1 Overview

Client side modeling models the web application from the client perspective or as the application is viewed from the client browser. For a normal web surfer browsing the web application at a client browser, the web application consists of a set of web pages residing at the web server and they are navigated by loading them into the browser one after the other in a certain sequence. This sequence is decided by the logic of the web application and is done via HTML hyperlinks.

To model the application from a client side (or a user side), we have to model what does this web user see in his client browser; and this is HTML pages and HTML components. Even when considering dynamic pages and server side scripts, we only deal with their HTML output as seen by the client regardless of the other logic running at the server. From a user point of view, he is reading HTML pages, and interacting with HTML controls, mainly links and forms.

As with all three parts of our web applications model, I am going to use graphs in order to build our analysis model for this part. This model will present the web application in a graph similar to UML diagrams (Unified modeling language) but with major changes. A similar approach was proposed by Ricca and Tonnella [3] who modeled the web application as a UML diagram, but that model does not fit the spirit of our model since it is too general and it mixed the client and server side components into one model. In my model I aim at splitting and differentiating between them. Thus I am going to propose a new model which meets our requirements.

In general, my model will represent web pages (Html pages) by boxes (nodes) and transitions between pages will be presented by edges connecting the boxes. Off course we have many kinds of web pages so we should have different kinds of boxes representing those pages differently. We have different kinds of transition links connecting pages and thus we should have different representations for them as well. The representation of the pages and the transitions should allow us to understand and analyze the structure and the flow of the web application clearly and efficiently.

For the sake of simplicity and in order to make our web application similar to standard graphs and for the sake of simplicity, I will assume that the web application starts at one start page and ends in one end page. If there are more than one start pages, we can create one start page with branches to each one of them. Similarly if we have different exit pages, we can link them all to one exit page.

4.2 The Model

4.2.1 Description

Web pages have different types and behaviors, and since we cannot model each and every case, it is important to differentiate between those three types which cover most cases of HTML pages:

- Static Html Pages
- Dynamic Html pages
- Pages with frames

Static Html pages are those pages that always return the same content regardless of the user input, the state of the server, or any other variables what so ever. Those pages are simply HTML pages located on the web server which are laded as they are into the web browser. I chose to model those pages as squares where each square is tagged by a pair of the page name and the server name where this page resides.

Dynamic Html pages are those pages that are generated by a server side script such as CGI, JSP, or ASP. In this case, what exists on the web server (which is an application server as well) is a server side script or program. When the client requests this server side script by calling it via a normal hyperlink as a static page, this script is executed by the web server and its output is returned to the client browser. Normally, the out put of those scripts are HTML tags which are collected and sent to the client browser which interprets them as a web page and displays them to the user.

In most cases, the server side scripts take input from the browser during the request process and this input can originate from the user input forms or from parameterized hyperlinks. This input is processed at the server together with other dynamic variables such as database values or state variables. Based on the script's logic, a certain HTML output is generated and sent to the browser. Dynamic pages are modeled as rhombus which is tagged by a pair containing the page name and the server name it resides on.

Pages with frames are very popular in many web applications. Having a frame inside a web page is like dividing it into two pages and each one having a different page loaded into it. A simple web page can contain as much frames as we want, and into each frame we can have pages that can be framed as well. Modeling frames is a bit complicated since each frame has a separate page loaded into it and all are contained in the main page with the dividing frameset. Moreover a difficulty arrives in the way pages are loaded into frames and in the links between frames. For while we are still in the same main page, we can navigate different pages in one of the frames, we can load other pages into other frames, or we can as well navigate from the original page to another main page (with or without frames).

Since our aim here is to model the application as seen by the web user, we will be modeling frames as they appear in the client browser, so we will be modeling the page with frames as a box that is divided into sub boxes where each internal box refers to a frame of that page. The holding box should be tagged by the name of the main page holding the frameset. Inside each sub box we should write the page name that is originally loaded into that frame. And since frames can take different shapes and can divide the page in many different ways (vertical or horizontal) and different proportions, we are going to design the sub boxes in the same way by drawing vertical and horizontal dividers and having the proportions the closest to reality as possible.

While in the same main page, invoking a link in one of the frames might load a new page into the same frame or any of the other frames while the rest of the frames remain unchanged. Invoking more links in any of the frames on the page could lead to loading different pages into different frames accordingly.

To generalize this behavior, invoking any link in any of the frames in the main page can lead to changing the content of one or more of the frames of that page. As a result a sequence of links will lead to a sequence of pages loaded into the frames of the main page while the main page is not changed. This sequence of navigation should be modeled by replicating the main page as long as navigation is done within one of its frames until the main page (containing the frames is changed):

1- Each time a new page is loaded into one of the frames, a new box corresponding to the current content of the main page should be added to our diagram and this time the name of the new page should be assigned to the corresponding frame.

2- The new box should be connected to the previous box corresponding to the same main page before its content has changed. The arrow should originate from the frame where the link has originated

3- The sequence of boxes should be connected in the same way that different pages are connected via links or forms Submit and are designated in the same way that normal links are done.

Page Transitions and links: Roughly speaking, pages are called and loaded into the browser using three ways: Hyperlinks, Form Submitting, and Server redirects.

- *Server redirection* happens when a dynamic page is executing at the server and within the executing code there is a command to redirect the execution to another page. The new page now executes and its output (or itself if it is static) is returned to the client browser and not the calling page. Server redirection of pages is very common, and it happens completely at the server and it is possible to have multiple page redirects before the last page is executed and returned to the browser. Hence it is transparent to the user. As a result, server redirection will not be modeled here but we will be looking into it in detail in the section which models the server side programs. So a page with a server redirection will be represented as a normal dynamic page with an arrow arriving to it from the main page and another dotted arrow leaving it to the final output page regardless if it involves other intermediate redirects.

  Moreover, the final page reached after redirection is not always the same. We can have the same dynamic page redirecting to different pages depending on the logic of the application. In the model, I model this by drawing more than one arrow starting from the page with the redirect and ending in each of the possible pages redirected to. This models *conditional page transitions.*

- *Links:* Web pages are connected to each other by hyperlinks or simply links. A link is a reference to a web page which when clicked will result in loading that page from the server into the client browser. If the referenced page is static then the page is transferred as is from the server. If it is a dynamic page, then the page is executed at the server and its output is sent to the client browser. Sometimes, a link can contain parameters called a Query String, and this query string is passed to the dynamic page as arguments before it gets executed on the server (those are ignored by static pages). The concept of links is very easy and it is modeled by drawing a single headed arrow from the page containing the clickable link to the page which the link refers to. If any parameters are passed with the link as a query string, then the arrow is tagged with a list of parameters' names passed with this link. It is very normal to have on the same page many links leading to many pages. We model this by drawing from this pages more than one arrow, each leading to one of the pages linked to this page.

- *Forms:* Forms are the most important components that provide the web user with an interface to enter data into the web page since they contain all the needed sections for data entry such as text boxes, radio buttons... Each form has a target page to which the entered information is submitted and normally this page should be a dynamic page coded in a way to handle the user input. After the user fills the form with the desired information, he submits the form to the server targeting the predefined webpage. The targeted page is then passed the input parameters and then executed at the server with its output returned to the client browser. From a client point of view we are only concerned with the parameters that are passed rather than with what happens with them on the server. We model a form on a page by a small circle tagged with the form name. From this circle the arrow representing the form submission should be drawn. Calling a page by submitting a form to it is modeled by a double headed arrow which is tagged by a list of the parameters that are passed. The arrow starts from inside the circle representing the form (in order to differentiate between many forms on the same page) and ends in the box represented the page we are submitting to. Sometimes, a page submits the form or has a link referencing itself. This is modeled as a circular arrow starting and ending in that page. It can be double or single headed respectively and it can be tagged with parameters. When we have forms within dynamic pages, it is possible to have a single form submitting its input conditionally to more than one page. We model this by branching arrows from this page to all possible pages that it can submit to.

4.2.2 Example:

I am going to give an example of a simple web application which allows a user to logon to his email inbox. This application consists of the following pages:

- Logon page (logon.html): this page contains two forms. The first a form for entering username and password and this form submits its content to the logon validation page (loginvalidate.asp). The validation page can redirect to the email page (mail.asp) if the logon information is correct or it redirects back to the logon page if they are wrong. After three unsuccessful logon attempts, the validation page redirects to an error page (sorry.html). The second form in the logon page is a form that requests the password to be sent by email. It submits its content to the page emailmypass.asp which emails the passwords and then redirects back to the logon screen.

- The inbox page has three frames. The first frame has the page links.html loaded into it. This page has one link to sign out from the email which leads to the sign out page (signout.asp) which redirects in terms to the logon page. The second frame a page listing all the available folders in the email and it has the page folders.asp loaded into it. Clicking on any folder name in this frame will load the folder contents in the third frame. The third frame can have any of the following pages loaded into it: inbox.asp, sentitems.aso, or trash.asp.

In this example, as it appears in Fig 4.1, I didn't represent the page names as a pair of (pagename, servername) in order to keep things simple and because I assume that all pages reside on one web server.

Moreover, I removed some obvious links (arrows) in order to keep our diagram neat and readable especially that is was made for the sake of demonstrating the model. I basically removed three links which are: the links from the page folders.asp to the pages: inbox.asp, trash.asp, and sentitems.asp.

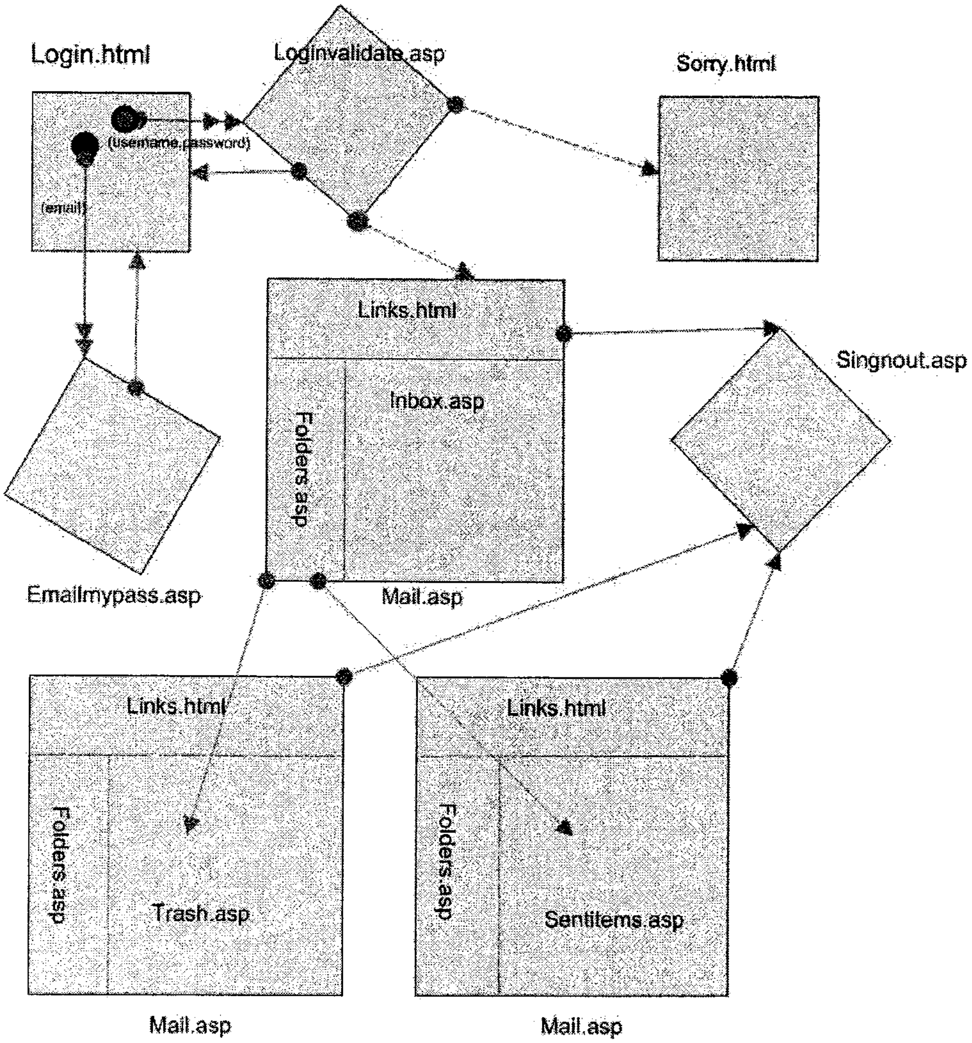


Fig. 4.1 - Example of Client Side Model

4.3 Testing Techniques

4.3.1 Overview

The client Side model presented earlier provides us with an analysis model to represent web applications and to build our analysis and our testing techniques on top of it. In this section I am going to suggest a set of testing techniques based on this model and which enables us to test the correctness of the web application as viewed from the user side or from the client side.

Testing the correctness of the application from the client side differs greatly from testing the correctness of the server side programs. Testing the correctness of the server side programs deal with testing each server side application by itself to ensure it produces the required output as per the required logic of our application. Moreover, testing the server side programs tests the inter connection of different server programs to ensure that they produce output as per the logic of the application. However, testing from a client side does not deal with the logic running at the server and does not need to validate the proper flow and execution between different server components. The client side testing assumes that the server side programs are correct and based on this, the tests should focus on the correctness of the HTML components displayed in the client browser to ensure that they provide the user with the proper HTML pages, proper flow of execution, and proper navigation.

To further elaborate the difference between testing server side programs and client side pages; let's assume that a certain web application contains a logon page which authenticates a user and another page which displays the grades of this user. Using server side testing, we can test the correctness of each of those pages alone. In other words, we can test that the logon page can authenticate the user as per our requirements and that the other page retrieves and selects the grades of the user only if he logged in successfully. But this is not enough, since we need to test from a user point of view that the "login" page always precedes the "grades" page and there are proper navigational components between the two pages. This testing of the order of execution and of navigation is tested in the client side model assuming of course that the server side logic is correct and supports our flow of execution correctly. In other words when we test that there exist transitions between those two pages in both directions, we assume that the server side logic correctly handles the submitted arguments from the logon page and correctly redirects the browser to the login page if the user is not logged in.

Thus, In order to perform Client Side testing, we have to test all pages visible to the user, their main components, and the transitions among them. So we have to test the following:

1- Orphan pages
2- Broken Links
3- Dead End pages
4- Parent Child sequences - parent pages are pages that should precede other child pages

The sequence of performing the tests is important especially between the first and the second test since the second test uses the all-node coverage criterion and assumes that no orphaned pages exist in our application.

4.3.2 Orphan Pages

Orphaned pages are regular web pages in our web applications that cannot be reached from any other page. This may be the result of having wrong links in some pages or after changing the links in other pages. Having an orphaned page is not desirable at all in web applications because it would create a major issue in applying graph coverage criteria especially in the all-nodes criterion where we will be unable to obtain a set of paths covering all nodes. Those cases should be dealt with in one of two scenarios:

1- If the orphaned page is not used anymore in our application, it should be removed from our application, probably saved in a different folder, because having two many pages especially with similar names becomes confusing.

2- If the orphaned page is important and should still be used, then we should analyze our diagram to determine where the wrong or missing link is and then fix it.

To determine the orphaned pages we do the following simple analysis on our graph and nodes:

If N is the set of all nodes, N' is the set of orphaned nodes (initially empty), and E is the set of all edges. Elements of E are designated by the triplet (e1, n1, n2) where n1 and n2 are the two nodes connected by the edge e1 in the direction from n1 to n2.

We start our analysis in determining the orphaned nodes by taking each element n of N (except the start node) and searching the set E for a triplet that has n as its third member. In other words we search for an edge that starts from any node and ends in n. If we cannot find such an edge then n is added to N'. Once we finish this test for all nodes of N, the set N' will have all the pages that are not reached from any where else in the graph.
The pages in N' are then analyzed, they are either removed from N (and N') or corresponding edges for them are added to set E (or existing edges are modified) so that all pages are reachable from other nodes.

4.3.3 Broken Links

Sometimes, we rename or remove some web pages from our web application without updating all the links in all our pages and eventually we will have some links referring to a non existent page. Eventually we will have our web application graph with edges and nodes that do not exist in the real application. When we generate test cases based on our graph model and we try to exercise them on our application, we can detect those broken links and take note of the errors to be fixed later. Based on this scenario, we should generate test cases or test paths in the graph that should visit all nodes in the graph and this is done by the all-nodes graph coverage criterion.

To generate the test cases for the all-node coverage criteria, we create a set N of all the available nodes. We start by the first node and we pick a path from the start page to the end page. Each time we visit a new node, we add this node to the set N' (set of visited nodes). If set N' is not equal to the set N, we add the path that we just traversed to the set T of paths and we start traversing a different path from the first node to the last node. We keep on repeating this procedure until the set N = N'. Now T contains the set of paths that traverse all the nodes in the graph (all-nodes coverage criteria). As we can notice here, in case we have an orphaned page we can never have a set of paths traversing all nodes so it is important to perform the previous test before this one.

Now that we have the set of test cases (test paths in set T), we start exercising them on our real application. Whenever a page in our test case does not exist in reality, we add this page name and the name of the originating page as a pair to the set B.

After we finish all test cases, the set B contains all the broken links (links to non existent pages) that should be fixed. Each link is designated by a pair (source_page, destination_page)

4.3.4 Dead End Pages

When I designed the analysis model, I decided that the application should have one start page and one end in order to make it easier for us to apply graph coverage techniques and general graph theories. In order to generalize this assumption to all web applications, I suggested introducing small changes to applications that do not have a single start or end page by adding one start page leading to all initial start pages and by adding a single end page reached by all initial end pages.

Dead End Pages are pages that do not have any links to other pages and thus they force the user to be locked in them or force him to use the browser's Back button. Using the browser's Back button is not recommended at all and should be avoided in our web application because it produces an unplanned order of execution and it cannot be handled by the logic of the web application and thus might lead to wrong results. For example, if we press back and we go to a dynamic page that updates the database or some global variable, this will result in the update to happen twice and thus having a value different from the intended one. All the pages except the final page should always have links that allow the user to leave the current page.

This test analyzes the web application graph and detects any page that is a Dead End page. Since this test follows the orphaned pages test in order, then we are sure that all pages in N are not orphan pages. The testing technique goes along similar lines of the previous two tests and it is as follows:

Let be N is the set of all available nodes, and E is the set of all edges as defined in the previous section and D is the set of Dead End Pages. D is initially empty. We start investigating each node n of N (except the end page). The node n should have a corresponding edge in E where n is the second item in the triplet. In other words each page in N except for the end page should have an arrow leaving it. If any page does not meet this condition then it is added to the set D.

After we finish investigating all nodes, we start analyzing the nodes we have got in D. Each node can be dealt with based on one of two scenarios:

1- If this page was initially and end page and no further navigation is required from it, then we should add navigational hyperlink from this page to our end page in order to meet our global assumption that each application has one start and one end page

2- If the page is in the middle of an execution path and further navigation is expected after it, this means that we have some missing link from this page. In this case this page should be analyzed as a separate entity in order to determine the missing link and fixing it accordingly.

4.3.5 Parent-Child Sequences

Parent pages are those pages that must be traversed before other pages. A typical example is the logon page in a banking application where the user should logon before he is able to navigate to any other page. In almost all web applications we have parent pages that should be navigated before certain pages. Those parent pages are determined by the analysis and the logic of the web application and mainly are considered as part of the technical requirements of the web application. This kind of information is collected during the requirements gathering phase and the modeling phase which are the early phases in any software engineering cycle.

Determining and testing all parent pages is not an easy task because we can easily end up analyzing the precedence of every 2 pages in the application which is too complicated and an impossible task in some cases.

First we should differentiate between direct parents where a parent page should precede its direct child (connected by an edge) and between grand parents where a parent page should precede another anywhere in the path.

To start our testing technique, we designate the set of direct parent pages by P1 where all elements of P1 are pairs. The first item of the pair is the parent page and the second item is the direct child page. We define the set P2 of indirect parents where each element is a pair of the parent page and the other child page that should follow the parent page later in execution sequence. E as defined earlier is the set of edges. T is the set of paths that satisfies the all-node coverage criterion as defined earlier. For the sake of completeness we define F1 as the set of not satisfied parent child requirement and this set contains all direct parent child pairs that did not succeed the test. We define F2 to hold the indirect parent child pairs that failed the test as well. F1 and F2 are initially empty.

We start by testing the direct parents. For each pair in P1, we search for a corresponding pair in E where the first and second items from P1 match the second and third items in E correspondingly. If such a match could not be found, then we add the pair of P1 to F1 to deal with it later. Similarly, we have to make sure that the child page is not reached from pages other than the parent so we should check the set E for edges ending in the child and originating from any node other than the parent (specified in the pair from P1). If such a match is found, then the pair from P1 is added to F1 as well. When we finish, we analyze all the pairs that are in F1, and we fix the pages (by adding the needed hyperlinks, and removing the wrong ones) and the diagram accordingly in order to satisfy the required and missing transitions.

As for indirect parent child relationships, we have to admit that we can never test this completely simply because it requires checking all possible paths passing in the child and to verify that all those paths had traversed the parent before that. This requires that we check all possible paths in the graph (the paths returned in the all-path coverage criterion) which is simply impossible with any graph with cycles. Since the full solution is impossible, I am going to propose a technique that tests an important subset of the paths and gets us close to the full test. This methodology of partial testing is very popular in software engineering practices and it is known as the optimistic methodology since it assumes that we can discover most of the errors through testing an important subset of the graph.

What we should verify here is that: for each parent child requirement, all paths traversing the child should have traversed the parent earlier. Since validating all the paths is impossible we start by using the paths defined in the set T of the all-nodes coverage criterion. So we start by taking each pair in P2 which is in the form of (n1, n2) where n1 is the parent and n2 is the child node. Next we check all paths in T that contain the node n2. For each one of those paths, we have to verify that n1 precedes n2 in sequence. We are guaranteed to have at least one path traversing the node n2 since we are looking at a set satisfying the all-nodes coverage criterion.

After we finish the above step, most probably we will be left with a set of pairs in F2 that does not satisfy the conditions, or in other words a set of pairs having the parent child requirements violated. After that we have to analyze each of those pairs and to fix the web application and its corresponding graph accordingly. Since the graph will be changed, it is very possible that we have messed some other requirements that have been working in previous tests. So we regenerate the sets, E and T and we repeat the tests on P2 and so on.

The above test provides us with result on the available paths covered by the all-nodes criterion. This criterion provides the minimum set of paths guaranteeing that every node is exercised at least once. However, this set is not enough to guarantee complete testing because for a child node that is accessible by many paths, we can have a path that does not satisfy our requirement but is not in the set T and thus hasn't been tested yet. In order to reduce the level of errors resulting from this scenario, we try to increase the set T by adding additional paths that are candidates to contain violations. The more we increase the set T, the more our application is validated against violations but the more difficult testing becomes.

A simple rule to follow while adding paths to set T is as follows. We define level-K coverage criterion on a set of nodes L, as the set of paths covering each node in L at least K times. The previous all-nodes coverage criterion is simply level-1 criterion on set N. So we add to our new set L (which is subset of N) the child nodes that are reached by more than one path. We can use the level-k criterion to satisfy cycle coverage requirements. To cover the traversal of a cycle at least once, we add the starting node (which is the same as the end node) to the set L, and we apply level-2 coverage on set L. This way we can guarantee that each starting node of our cycle is visited at least twice which means that the cycle is traversed at least once. To simplify things, we will limit set L to:

- Child nodes that have more than one arrow arriving directly to them in the graph
- Child nodes whose path from the start node has a cycle

After identifying the set L, we can start adding to T, the paths satisfying level-k on L, where K can range between 1 and 100 depending on how deep we intend to go in testing.

4.4 Regression Testing Techniques

4.4.1 Overview

Now that I presented different testing techniques for the client side model I have to complete the picture by presenting regression testing techniques. As we all know, regression testing is an essential part in software testing because it deals with testing operational software after modifying it as a result of maintenance or upgrade.

Since software testing is very expensive and time consuming, we should test efficiently only the changed part and not the whole software again. The challenge with regression testing is to find a methodology to determine the test cases to test efficiently the changed sections only. In the following section, I will be proposing some regression testing techniques based on my model and on the testing strategies presented above.

We should start by identifying all the possible changes that can be applied to a web application due to maintenance or upgrade. We are not concerned here with all possible changes that would happen especially at the server side or with the architectural components. We are mainly concerned with the change that affects our client model as presented earlier. Those changes can be safely summarized as follows:

1. Adding a web page
2. Removing a web page
3. Changing the links and the forms on a web page
4. Renaming a web page

For each of theses cases I am going to propose the most optimized technique to do regression testing for each of the four tests presented earlier.

4.4.2 Adding a Web Page

Adding a web page is basically adding a new node to the set of nodes N, and adding two or more elements to the set of edges E. Adding a page requires that we should repeat all four tests on a part of the Application:

- Orphan Pages: Adding a page means that we are inserting it in the graph and connecting it with edges arriving to it from existent nodes and edges leaving it to existent nodes. Thus if we are to have Orphan pages as a result of those pages, it is going to be limited to the added node itself and to the nodes that are supposed to receive arrows (edges) from this new node. The nodes that have arrows leaving them and arriving to the new node are not at risk of being orphaned since the links arriving to them are not changed. The same applies to all other nodes of the graph. So regression testing of orphan pages in this case can be done as follows:

Let A be the set of pages that are added and the pages that are supposed to receive links from those added pages. For each node n in A check that a corresponding edge e in E exist such that the third element in the edge triplet is equal to n (the edge e arrives to node n)

- Broken Links: after adding a page, we have to change the links in the pages that should have edges leaving them and arriving to that page. Moreover, we should add edges from the added page to the other nodes. So the possibility that we might have broken links as a result of this operation can be limited to the links leaving the added page itself and to the links leaving all the pages that have links arriving to the added page (or in other words the pages preceding the added page in the path from the start page). We add those nodes to the set A of affected nodes. Now that we identified the nodes that should be tested for broken links we select from E all the triplets (edges) that have any of those nodes as the second element. Finally, after identifying the set of edges that need to be tested, we can generate a set of test cases that traverses each of those edges at least once. Normally this set should be very small and if for any reason it became big and not doable manually, the all-node criterion can be applied on the set A of affected nodes.

- Dead End Pages: The problem of having pages with no exit paths can only appear with the added page itself, because we assume that all other pages have been tested previously. So easily we can check if the added page has links leaving it to other nodes or to the end page.

- Parent child relationships: Precedence requirements between pages can be changed upon adding a new page. Whether the added page needs to be a parent or a child and whether the relationship is direct or indirect. Even though the initial testing techniques and especially the indirect one are very expensive and take time, we can apply regression testing techniques efficiently when adding a page. Assuming that the set of requirements are changed correctly due to the change in the application, we simply pick from the set P1 and P2 (sets of direct and indirect parent requirements) all the pairs that have the added page as the first element or the second element. Next we can apply the testing techniques specified above on just those pairs (that should be few). The indirect parent relationship testing is easier now since the set of paths that we have to verify are only those traversing the new page and not the all-node criterion as before.

4.4.3 Removing a Web Page

Removing a page is basically removing a node from the set of nodes N and removing all its corresponding edges in E and off course adding new edges to link the other nodes that were connected to the removed node. Regression testing of our four tests is basically easy and similar to the methodology we used in the case of adding a page and we are going to look at each one separately.

- Orphan Pages: Removing a page from the graph means that all edges leaving it to other pages are removed too. This might lead to having some of those pages that used to have links arriving to them from the removed page to be orphaned if that was the only link arriving to them and if no other links were added or changed to point to those pages. So in this case, if R is the set of all pages that used to receive edges from the removed page, then we should verify that for each node n of R, there exist a triplet in E (edge) whose third item is equal to n (there exist and edge arriving to n)

- Broken Links: After a page is removed, all links arriving to it should be removed, and new ones pointing somewhere else should be added. So the only risk of having broken links will be with the pages that were previously pointing to the removed page where they remain to point to it after the page was removed. Regression testing for broken links in this case is straight forward and easy. It is similar to that of adding a page. Let R be the set of all pages that used to point to the removed page. We have to check that all edges leaving every node in R are working in the real application and thus note broken. For each node n in R we select all triplets from E having n as the second item in the triplet (in other words all edges starting at n). Then we generate a set of test cases that traverses all those edges.

- Dead end pages: the only problem with dead end pages might happen with pages that were previously pointing to the removed page, especially if the link to the removed page was the only exit of any of those pages. If R is the set of nodes that were previously pointing to the removed page, we apply on R exactly the same regression test strategy that we discussed for adding a page in the previous section

- Parent Child relationships: Removing a page, affects the links on the pages that used to have links to that page and on the pages that used to receive links from the removed page. Regression testing of this part basically consists of two parts. First we have to make sure that no more parent child requirements in P1 and P2 use this removed node. So we have to check the sets P1 and P2 as described above and to verify that none of them has reference to the removed node in any pair. If any such pair exists, then requirements and the graph should be reviewed and fixed accordingly. The second step is to verify that all existing parent child obligations containing any of the nodes that used to be connected to the removed pair in either direction still hold. So we follow the usual testing technique as described in the previous sections on those nodes only.

4.4.4 Modifying Links on a Page

Changing links and form targets on certain pages have direct effects on our web application diagram and those changes should be verified. In fact, the process of regression testing for the modified links is very similar for testing the added and removed pages since the same methodology is used but the involved pages in the tests change. Here we can safely say that always the involved pages are the page where the link originates, the page where the link used to point and the new page where the modified link points to. We will assume that those three pages are placed in a set C. I will briefly go through the procedure through the four regression tests:

- Orphan Pages: Using the same technique discussed through this model, all elements of set C are checked if they are orphan pages.
- Broken Links: Elements of set C are verified for not having broken links as per the methodology used with the cases of removing and adding a page.
- Dead End pages: All pages that are elements of set C are tested if they are dead end pages by the same methodology used before
- Parent Child Requirements: those are checked on all elements in set C and using the same technique for testing the removed and added pages proposed before

4.4.5 Renaming a Web Page

Renaming a web page can be simply considered as removing this webpage and then adding another one with a new name. So we use exactly the regression testing techniques specified earlier for removing a page and then for adding a page. This procedure though it is a bit heavy, but it guarantees for sure that all the above four validations (tests) are true for the page with the old name and for the page with the new name.

Having presented in detail, testing techniques and regression testing techniques for the client side model, I can conclude this chapter by confirming that we have a complete analysis model for representing the web application from a user's point of view. Moreover, we have a complete set of tests based on this model in order to verify the correctness of our application.

Chapter 5

THE SERVER SIDE PROGRAMS MODEL

5.1 Overview

Web applications as we all know are certain kind of software applications that can be executed using a web browser. The way in which the web applications execute is as follows:

1- Web pages are hosted on a certain web server. Those pages can be static pages which mean that they contain fixed data and they always return the same information. Or web pages can be dynamic which means that they execute at runtime and they return different results to the client browser based on the execution at run time. This execution flow and output can be affected by so many factors such as Database values, session variables, submitted arguments …

2- The client browser requests the pages from the web server which returns the pages in the form of HTML code. The browser parses the HTML code and displays it in the browser in a readable format. Pages displayed by the browser can be requested by typing the URL in the browser's address bar, by clicking hyperlinks on the page, or by submitting forms of input data.

3- The browser can submit data to the web server as input using Forms or by sending arguments in the URL as Query Strings.

Now that I briefly stated how web applications work, I can simply summarize the flow of execution of a web application as a sequence of interactions between the client browser and the web server. The web server might interact with other backend servers while executing but this is transparent to the client browser where the only response it expects is an HTML file regardless of the backend execution at the server(s). So from a client (browser) side, the application is a set of HTML pages that are returned based on some HTTP requests.

In the previous model representing the client side components, we looked at the web application as a set of static and dynamic HTML pages connected by hyperlinks. In that model we didn't analyze the different HTML sections and the way they were generated but rather we considered the page as a whole and analyzed the flow of execution between pages, assuming that the server side code is correct and supports the generation of pages and the flow of execution as per the requirements of the application. In this section we are going to look into the details of the server side code that is simply the engine that generates our HTML output. In this model I am going to present an analysis model that allows us to analyze the way in which dynamic HTML pages generation, dynamic links and dynamic server redirection. So in short, the client browser normally submits data to the server components via forms and links and it receives back HTML pages.

It is important to note that in this model we will not be testing the logic of the server side code itself, but rather the dynamic behavior in which this code creates HTML dynamically while executing and sends it to the browser. Testing the logic of the server side program itself and testing server components that do not generate any HTML to the browser can be done using classical software testing techniques and is outside the scope of this thesis and thus will not be covered.

I decided to adopt an existing model which models web applications with emphasis on the server side programs (proposed by Wu and Offut [2]). However I will be using part of the modeling techniques proposed by Wu and Offut [2] and I will be modifying it in order to come up with an analysis model which satisfies our requirements (as stated above) and completes the other two models presented before.

The original model considers the whole web application from the server side perspective and it represents the whole web application as a set of server components connected with each other. It even represented the hyperlinks connecting the pages with each other. However, this modeling is not powerful enough (as our client side model is) to model all the aspects of page interactions especially when it comes to static links and frames. So I am going to use from Wu and Offut's[2] model only the methodology to analyze and model single server component, server redirects, and intra-server component communication (referred to later as composite transitions). Moreover, I will be adding additional properties to the model in order to link it to the other two models represented earlier.

Derived from the original model [2], I will follow the below steps in order to create my model:

1- Atomic Sections (ATS) :We first Define something called the Atomic Sections
2- Composite sections (CS): From the ATS defined above we will derive the Composite sections
3- Transitions: We define transitions and interactions between different composite and atomic sections
4- Transition rules: We define transitions rules
5- From ATS, CS, and Transitions we model the web component

5.2 Definitions and Concepts

Before proceeding in describing the model, we have to define and explain some concepts and definitions that we will be using to construct this model. Some of the definitions are based on the work presented by Offut in [1] and [2], but I customized and enhanced them so they fit the spirit of my model. In this section I will be discussing the concepts of atomic sections, composite sections, composition rules, and transition rules.

5.2.1 Atomic Sections:

An Atomic section as defined in [1] and [2] is a section of HTML code, and it has the property of everything or nothing. It means that only the complete atomic section can be sent to the browser but not part of it. The simplest example of the atomic section is a static HTML page with no server side code.

If we have a dynamic page, then the dynamically generated HTML code should be divided into many atomic sections such that each atomic section is a consecutive set of HTML code and not interrupted with any server side code. If we have server side scripts interrupting the HTML lines, then this server side script would be able to control the rendering of the HTML lines and might stop the lines below it from being sent to the browser based on the logic of execution. In this case the HTML lines should be divided into two atomic sections. If the set of HTML lines does not contain any server side code within it, then the executing program does not have control to interrupt sending those lines to the browser at any point.

Atomic sections are similar to the basic blocks of a standard program, but they deal only with HTML responses while ignoring the internal processing of the software.
The below example (Table 5.1) is a jsp page that outputs html code based on the program's logic. The atomic sections are tagged with p1, p2 …

```
ID = request.getParameter ("ID");

passWord = request.getParameter ("PASSWD");

retry = request.getParameter ("RETRY");
```

```
         PrintWriter out = response.getWriter();
p1   =   out.println ("<HTML>");
         out.println ("<HEAD><TITLE>" + title + "</TITLE></HEAD>");
         out.println ("<BODY>");
         if (Validate (ID, passWord) && retry < 3)
         {
p2   =   out.println ("<B> Grade Report </B>");
         for (int I=0; I < numberOfCourse; I++)
p3   =                  out.println ("<P><B>" + CourseName(I) + "</B>" + CourseGrade(I) + "</P>");
         }
         else
         {
p4   =   retry++;
         out.println ("Wrong ID or wrong password");
         out.println ("<FORM Method=\"GET\" Action=\"GradeServlet\">);
         out.println ("<INPUT Type=\"TEXT\" Name=\"ID\" Size=\"10\">");
         out.println ("<INPUT Type=\"PASSWORD\" Name=\"PASSWD\" Width=20>");
                    out.println ("<INPUT Type=\"HIDDEN\" Name=\"RETRY\" Value=" + (retry) + ">");
         out.println ("<INPUT Type=\"SUBMIT\" Name=\"SUBMIT\" Value=\"SUBMIT\">");
         out.println ("<A HREF="Call sendMailfunct()">Send Mail to the Instructor</A>");
         out.println ("<INPUT Type=\"RESET\" Value=\"RESET\"></FORM>");
         }
p5   =   out.println ("</BODY></HTML>");
         out.close();
```

Table 5.1 - Atomic Sections in Sample JSP page

5.2.2 Composite Sections:

I will adopt the definition proposed by Offut [2] of composite sections. Atomic sections are combined together to form composite sections. Composite sections are created dynamically at run time and are affected by the control flow of the server code. Different users will have different HTML responses based on the logic of execution of the server component. Possible compositions of the atomic sections into composite sections are: Sequence, Iteration, and Aggregation.

1. Basis: p is an atomic section.
2. Sequence: ($p \rightarrow p_1.p_2$): p1 and p2 are atomic (or composite) sections, and p is composed of p1 followed by p2. In this case the output of p is simply, the html output of p1 concatenated with the html output of p2.
3. Selection ($p \rightarrow p_1 \mid p_2$): Suppose that p1 and p2 are atomic (or composite) section. A composite section p that selects either p1 or p2 (and not both) is a kind of composite section that combines both atomic sections.
4. Iteration ($p \rightarrow p_1^*$): If p1 is a composite section, then p is composed of repeated copies of p1.
5. Aggregation ($p \rightarrow p_1\{p_2\}$): p1 and p2 are composite sections, p2 is included as part of p1 when p1 is transmitted to the client. For example, a function call in p1 or an include command will include p2 in p1. Here p is a composite section composed of two atomic (composite sections) aggregated in each other.

The above stated composition rules can be extended by using regular expressions or BNF notations.

For example the + sign can be used as $\mathbf{P}^+$ to represent **one** or more composite sections concatenated together to form one composite section. $\mathbf{P}^*$ can be used to represent **zero** or more concatenated sections.

Those notations are good to model *while loops* in the server component that contain an atomic section within it. Similarly $\mathbf{P^n}$ can be used to represent **exactly n** concatenations of P, this is a good way to model *for loops*. Moreover, we use the "dot" operator, so *S.p*1 indicates the composite section *p*1 is produced by program component *S*. this is useful when we have aggregate components within each other and it is useful to identify which component produced the HTML atomic section.

5.2.3 Transitions

Web applications, use HTML and Web links in order to combine different Web components (modeled above). The web application uses four different kinds of interactions to combine web components: HTML Links, Composite Transitions, Operational Transitions, and Server Side redirects,

1- Link transitions from component p to component q: those are normal HTML hyperlinks between pages and components. Html links are modeled in the client side model because it is the client's method to navigate pages. Html links will not be modeled in the server side model because what we focus on here is to model the internal structure of a single page or component and not the links between different pages.

2- Composite transitions modeled as s->q: The execution of component S generates p and sends it to the client browser in HTML format. Normally a single component S can generate many components p1,p2,p3... so we can represent the composite transition in the composition rules as: s|-> p1 | p2 | ... |pq

An example of a composite transition is calling and executing a java bean or a servlet from within a JSP page. Here the HTML output (atomic section) produced by the bean to the page is considered as a composite transition from the component (bean) to the atomic section.

3- Operational Transitions: the user can create a new transition in the software which is not designed within the logic of the application such as pressing the Back, Stop, or Refresh buttons in the browser. Those kinds of transitions are in general not preferred in web applications, but in all cases this has to do with the transition between pages (similar to hyperlinks) and thus will not be modeled here.

4- Server Redirect: or what we call forward transition are modeled in the composition rule as p|=> q. This is a server side transition that is not controlled by the user but rather it is an automatic redirect or transition from one web component to the other. For example when a user log on successfully, the user gets redirected to his home page automatically

5.2.4 The Composition Rule

Based on the above definition of composite sections and their possible compositions, we can model the internal structure of a Web component (which is a big composite section of smaller composite sections) by a *composition rule* which represents **all** possible generated output for a certain web component. For example for the above stated example, the composition rule would look like $(p2.(p3|p4)^{*}|p5).p6$

Up till now, we defined a way to divide the web component into atomic and composite sections, and based on those definitions, we suggested a way to model the internal structure of any web component.

5.3 The Model

5.3.1 Description

Based on the definitions of the atomic sections, composite sections, the composition rules, and the transition rules defined above we are going to define a way to model sever components and inter server communication as follows:

A web component can be modeled as a quintuple:
WC={S, C, T , ATS, CS} where S is the start page, C is a set of composition rules, T is a set of transition rules (Composite transitions and aggregate transitions), ATS is the set of atomic sections and Cs is the set of Composite sections.

After defining the above sets and the web component, we can model the component in a graph, where each node is an atomic section, composite section or another web component, and the edges are composition and transition rules as explained above. Modeling the web component as a graph was not proposed in the original model by Offut [2] who considered the whole component as a singe node in the Web Application Diagram.

Nodes: The nodes of the graph can be atomic sections, composite section, or web components. In order to differentiate, we model each one of those in a different way.

1. The atomic sections (which are the basic block) are modeled as a normal circle.
2. The composite sections (which are a composition of two or more atomic sections) are modeled as a square.
3. Web components which are complete working entities are represented as a rhombus. In our model, we might just need to represent a reference to another component

without worrying about its internal details. We do this by representing this whole component as one entity (rhombus).

Edges: To start with composition rules, we will be using the following terminology to represent composition rules. Off course composition rules are used to combine different atomic sections together to form composite sections. So here the nodes are only limited to being atomic sections.

1- Sequence of two atomic sections is represented by a regular arrow starting at the first component and ending in the second component.

2- Selection between two atomic sections is done by drawing two arrows leaving the same atomic sections to two different sections. This means that either the first section or the second can be selected following the previous section. If we do not have a selection between two composite sections but we have an optional selection of one composite section, then we represent this by a selection between the atomic section and an empty atomic section (which contains nothing) we will call it ***e***.

3- Iteration of a certain atomic section is represented by drawing a circular arrow around the node corresponding to that atomic section.

4- Aggregation of atomic sections in composite sections is modeled in drawing a double arrows (two parallel arrows) from the included section towards the including section.

Transition rules are the second kind of edges that will be used in our model. We will be modeling the server redirects and the composite transitions as explained above.

1. Server redirects will be modeled as double headed arrow. Since server redirect command is not an html section and thus is not in an atomic section, so we draw the arrow to start from the last atomic section before doing the redirect and which ends in the rhombus representing the component redirecting to.

2. Composite transitions are modeled as double headed dashed arrows starting from the component that generates the atomic section and ends in the atomic section composed by the component.

5.3.2 Example

For the sake of illustration above, suppose we have the web component "Grading" in the above example. If there is a start login page index.html that submits the values to this web component called "Grading". The component checks if the username and the password are valid. If they are valid then the student is presented with his courses and grades by looping on all courses and printing the output (this is atomic section p3). If the username and password are wrong, he is denied access and he is given the choice to re-enter the login information and he is presented by a link to call the server side component SendMailFunc() to send an email to the administrator (this is atomic p4). Note here that the send mail link is not a normal link to another web page, but it is a link that calls and executes the server side component S. As we can see here, the links between pages and the forms are not modeled but rather are considered as normal HTML output and a subpart of an atomic section and is not dealt with.

This model can be represented in a graph as well where different atomic sections are presented as boxes and they are linked by arrows representing how atomic sections are being generated from Atomic Sections.

So in our example we have the following:

ATS={p1,p2,p3,p4,p5}

CS=={p1.p2.p3*.p5, p1.p2.p4.p5}

Grading=p1.((p2.p3*)|p4).p5

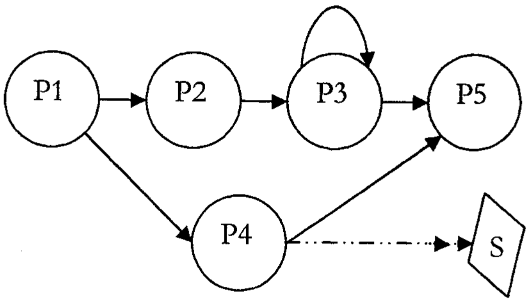


Fig 5.1 - Example of Server Component Model

The above example's simple diagram (Fig 5.1), illustrates how to model server components as per the technique we proposed. As we can see, The Composition Rule (as defined earlier) for this component can be derived completely from our diagram. The arrows from P1 to P2 and P4 represent Selection. The arrow from P2 to P3 represents sequence, the arrow from P3 to itself represents iteration, and the arrow from P4 to S represents Composite transitions where S is executed and its output is returned to the browser.

Moreover we can see that even though the interconnection of this component with other server components is modeled (from P4 to S), the external component itself is modeled as one entity without its implementation details. Detailed structure of component S is modeled in a separate graph similar to this one and specific to S.

5.4 Testing Techniques

5.4.1 Overview

The web application is modeled as a graph, where all nodes are atomic or composite sections and edges are transitions or composition rules. Thus any sequence of interactions will be resulting in a certain traversal of the graph and as a result a certain sequence of atomic and composite sections; or in other words a certain HTML output that will be rendered in the browser. As a result any traversal of our web component graph WCG from the start point to the end point is a test case that corresponds to a certain sequence of transitions and composition between the component's atomic sections resulting in the end with a certain output. If we are able to exercise all the possible paths in the WCG, then we will be able to simulate all possible behaviors and output of the web component.

Any undesired output can be traced back to the test case that generated it and as a result any bugs can be fixed. So for each test path we derive from our WCG, a corresponding test case should be prepared and tested for it.

The different traversal paths of the Graph correspond to a certain flow of logic and is driven by different input variables and other factors such as user interaction or environment variables. In our testing we are not concerned with what values or input data that would cause this execution or flow of logic, but we are rather concerned with the feasibility of this execution and with the risk of having possible undesired executions. That is why we try to cover all possible paths or executions in order to make sure that every possible track that this program would take falls within the program's intended requirements. Moreover, pieces of dead code (unreachable statements) or unfeasible branches, or impossible but required output combinations among others can be easily discovered by analyzing the test paths.

Off course for a relatively medium application, the total number of test paths can be extremely huge or simply infinite especially with the existence of cycles in the graph. As a result, different graph coverage criteria have been proposed in order to define the minimum required test paths to cover the graph. I am going to define briefly some of the classical known graph coverage criteria:

a. Node Coverage Criteria requires that all the nodes in the graph are covered in at least one path from the set of paths (test cases). In our case it can be an atomic or composite section or simply a web component. This criterion only requires that all nodes are covered by a set of paths.

b. Edge Coverage Criteria means that all the edges in the graph are covered by a set of Paths.

c. Path Coverage Criteria, a set P of execution paths satisfies the path coverage criteria if it contains all execution paths from the start point to the end point. However this is not feasible with the existence of cycles because the complete set of test paths is infinite

d. Elementary paths are those that navigate the graph from the start point to the end point with no cycles (no loops). This coverage criterion is not very practical since we rarely have any program with no cycles and it is not very efficient to test any program without testing its cycle iterations.

e. The cycle count criteria can be used to test cycles and it allows cycles to be exercised only a limited number of times. The cycle count-K criteria allows any cycle to be executed zero, one, or K times.

f. Prime Criterion is a criterion developed by Amman, Offut and Ling [2] and it was used in the original model which we built upon while modeling the server side programs. This criterion allows coverage of the graph with cycles by using the notion of sidetrips. I am going to look briefly at this criterion below because I will be adopting it in the server side model.

5.4.2 Prime Paths Criterion

We will be using the prime paths definition as defined by Amman, Offut and Ling [2]. They proposed that the prime path coverage criteria to be used on all the Web Application Graph, but here we will be using it on just the internal section of a web component. Following is the overview of the prime path coverage criterion.

A Prime path is a path between two nodes such that it contains no cycles. In other words any node is found at most once in this path except for the first and last node which can be the same. A path (test case) is a sequence of nodes from the start node to the end node and in our model it represents a test case.

A path p tours a sub path q if and only if q is a sub path of p. If the sub path q is a prime path then cycles are not allowed. In order to allow cycles coverage in the touring prime paths we introduce the notion of sidetrips. A path p tours a prime path q with side strip if p tours q and at the same time allows adding other nodes outside q and in the middle of q and hence allowing traversing cycles.

Formally speaking a path p is said to tour a sub path q with sidetrips if all nodes of q appear in p in the same order.

The prime coverage criterion requires identifying the paths that tour all the prime sub paths with sidetrips. We can start by identifying all the prime paths and then we can add the required sidetrips to tour the cycles. The easiest way to identify the prime paths is by using the all-node coverage criterion. This means that we start to explore prime paths in our graph and we add them to the set of paths T until all the nodes in the graph are covered at least once by one path.

In the below Example in Fig. 3.4, the Start node is S_0 and the end node is S_f the path [a, b, d] is a prime path since all its nodes appear once. The path that that tours abd with sidetrips is a path that passes in c as well like the path [S_0, a, b, c, b, d, S_f]. Other prime paths in the graph are [a, b], [b, d], and [b, c]. So the set of path that tour all prime paths with sidetrips are:

[S_0, a, b, c, b, d, S_f], [S_0, a, b, c, b, c, d, S_f]. [S_0, a, b, c, d, S_f].

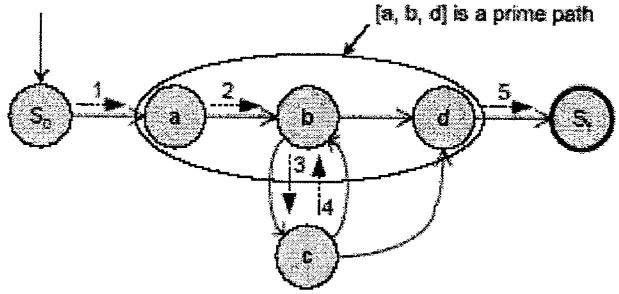


Fig.5.2 Touring a graph with sidetrips.

The prime paths provides a coverage criteria that supports cycles and as a result can be used safely in our model to generate most of the required paths or test cases to test our modeled application.

1- Define all Atomic sections in our application (this can be done automatically)

2- Define all composite sections

3- Draw the intra component graph or the WCG (this can be done automatically)

4- Determine the prime paths for the WCG or for the intra component graph too (this can be done automatically)

5- Determine the paths to tour all prime paths with sidetrips (can be done automatically)

6- Now that we have all test paths and test cases, prepare the input data needed for the test cases

7- Run the test cases and record results

8- Analyze the tests versus test results to check for unexpected behavior of the application or wrong output.

9- Double check to make sure that all Atomic sections were covered by the set of tests executed (since some atomic sections need special environment variables or special state variables)

10- Generate manual test cases if needed to cover any missed atomic sections.

5.5 Regression Testing

5.5.1 Overview

Regression testing, as previously defined, should be applied following maintenance or upgrade of an already tested and operational web component. Changing the server side component means changing its behavior and its internal structure and thus changing its Html output and eventually the atomic and composite sections. More over the intercommunication of the component with other components may be changed as a result of the change that we have done. In brief, the changes applied can affect our WCG in all its aspects and as a result the WCG should be revised accordingly and the affected sections should be tested.

As in all regression testing techniques, we should minimize the number of test cases while at the same time maintaining high quality testing. Following I will analyze the possible changes that might happen to the diagram and I will propose an efficient regression testing technique for that.

The possible changes for a web component might be one of the following:

1- Atomic section's content is changed

2- Adding new atomic sections

3- Removing atomic sections

4- Change in composition and transition rules

5- Adding and removing inter communication with external server components

5.5.2 Changing Atomic Section Content

As for the change in the atomic section's content, we don't really care much about that in our model. In other words we will be testing the changes that affect our Web Component Graph (WCG) only, and since the content of a single atomic section does not change the graph we will not test this. Anyways, the content verification comes as an independent procedure and it contains many details such as spell checking among other stuff and this is out of the scope of the models discussed in this thesis.

5.5.3 Adding and Removing Atomic Sections and Transition Rules

The other four changes cannot be taken separately and each one alone but all as one package. Because a simple change can at the same time add sections remove others and change the composition rules between them. As a result, I will be proposing the following technique to derive the most efficient test cases for our regression testing.

Let's assume that the set of test cases containing the prime paths is T. We designate the set of added atomic sections by A and the set of removed atomic section by R.

As for the change of composition rules (or edges), we need to identify the set of nodes that are involved with the changed edges, and we add those nodes to set C. The method we use to populate the set C is as follows:

For each edge that has been changed, identify the two nodes that were previously linked to this edge, and the two nodes that are newly linked by this node. We add those nodes to set C unless if any of those nodes belong to set R (has been removed).

- First, we remove from set T all the prime paths that path in any of the nodes in sets R and C (the nodes that have been removed or changed).

- Second, we add to set T prime paths that cover all the nodes in set A and C, such that each node is traversed at least once by a certain path. In case any of the added nodes is not covered by any prime path, this means that our all node coverage criterion is violated. In this case, we have to analyze those nodes manually and determine the reason for not being covered by any prime path. The reason mainly is one of two:

    - The node might be rarely reached in the real application and thus reaching it by a normal prime path might be difficult. In this case we have to generate a manual test case (path) to cover this node. Examples of such nodes are error handling nodes that depends on specific environment and global exceptions

    - The added node is un reachable at all in our program and in this case the composition and transition rules for this node should be revised and fixed before continuing our regression testing.

Now set T has a set of paths that satisfies the all-node coverage criterion for the existing and for the added nodes.

- Third, we select from the set T all paths that traverse any of the nodes in sets A, or in set C (added nodes and the nodes involved in change of edges) and we mark those test paths (test cases) for retesting them.

- Fourth, we run the test cases, record the results and analyze the output in order to determine any errors.

In this section, we have successfully defined a technique for testing and regression testing of the server side programs of the web application. Those techniques are based on the graph model we defined earlier.

Chapter 6

AN AUTOMATED BLACK BOX REGRESSION TESTING TECHNIQUE

6.1 Black Box Testing

The three models presented earlier provided us with a detailed analysis model for modeling and testing web applications. The three models complete each other and they are tightly connected that we can assume that the three of them are one big model where each sub part models the application from its own perspective.

White box testing in software engineering deals with testing and validating the internal structure of a software component. In white box testing the internal paths and the implementation details of the component are tested. So far all three models presented earlier provided us with analysis models for white box testing since they represent the web application with its small details. Eventually, all the testing techniques presented so far are white box testing techniques.

The architectural environment model presents the details of the implementation platform by representing all nodes and analyzing their connectivity, compatibility, and scalability among other features. The client side model, divides the application into a set of pages and it analyzes the connectivity and the transitions between them. The server side programs model, takes us one level deeper by analyzing the internal structure of a single web page (web component)

However, white box testing is not enough to validate the correctness of any application, because it is very common to have an application where each section is correct and works fine by itself but they fail to comply with the application's requirements when integrated and put together. For example we can have a set of software components where each one of them functions properly as per our requirements, but when we link them together in one application, they do not perform correctly as per our application's requirements.

Here comes the importance of black box testing techniques. It deals with the whole application as one entity and generates test cases and it compares output values versus input values.

6.2 Black Box Testing Challenges

Web applications differ from classical software applications in their architecture, and as a result we should take into consideration the following:

1. Black box testing for web applications can be done on two levels. It can be done on the level of an individual web component (web page) or on the level of the application in whole. If we do black box testing on the component level, then each component is treated as a closed box, and test cases are generated to test the input and output of this component. If we are to apply black box testing techniques on the level of the application as a whole, then we should look at the entire web application as a closed entity and we should generate test cases that would test global behavior of the application.

2. The web application's overall behavior cannot be tested by validating the output versus the input as in classical applications. This is because the web application does not have a specific output. For a certain test case, web application's behavior should

be validated based on the sequence of pages it traverses, the output of the pages traversed, and the final state of global (or database) values.

Similar to white box testing, black box testing should have regression testing techniques in order to validate the correctness of changes and upgrades quickly and efficiently.

6.3 The Proposed Technique

In this section I am going to propose a method for testing and automated regression testing of web applications. The level of black box testing we are going to use is on the application level and not on the component level. This means that we are going to run test cases that traverse the whole application and not just a single component; this is because I beleive that the white box testing techniques proposed in the server and client side models are enough to detect any errors on the component level.

Moreover, it is important to mention that this regression testing technique I am going to propose is an automated one, which means that we run the test cases manually one time. Later regression testing would re-run and validate those tests automatically (maybe after slight changes in the test cases). Besides being an automated technique, I am going to specify exactly the implementation algorithm and the detailed steps we follow to implement this technique. After detailing the algorithm, the implementation in any programming language would be straight forward and is then just a matter of coding.

The basic steps which we will be using in our technique are as follows:

1- Create Test Cases for the application and specify input data.

2- Use the developed tool with the embedded browser to run the test cases recording the visited URLs and the submitted arguments and form values.

3- While running the test force the developed tool to save certain HTML output for later comparison and validation in the regression testing stage.

4- In the regression testing step, the tool will execute the sequence of saved URLs automatically and it will collect the output values specified in step 3. After executing the test cases, the tool will compare the output values collected in this step with those collected in step 2 and it will provide the user with the sections that produced different output. The tester will have to analyze those differences manually.

As we can notice from the steps listed above, the technique is simply divided in two parts. The first part consists of selecting the test cases and this is done manually by the tester (step1). The second part is using the developed tool to execute the cases, collect information and re-execute the cases later on (steps 2, 3, and 4). Since our technique can be summarized in those two parts, we will be discussing each one of them in depth in the following sections. First I will be proposing an efficient algorithm for test case selection, and then I will be explaining in detail the architecture of the tool and the algorithm of its functionality.

6.3.1 Test Case Selection

The test cases that we should create for black box testing should traverse the application based on a certain set of input values to be defined by us. So the test cases should consist of all input criteria at each transition. Since almost each page of the web application requires different user input, then combination of all input on all pages might be extremely large or even impossible. In order to make test case generation efficient and manageable, I am going to propose a technique for black box test case selection. The below technique may not provide us with a complete coverage of all scenarios but will cover the majority of scenarios. We might need to generate manually few additional cases in order to cover all possible scenarios of the web application behavior.

Since defining test cases for all possible inputs is not feasible I am going to define input ranges instead of choosing input values from the entire domain and select the test cases based on them.

I am going to categorize the possible input types are as follows:

1- Text values entered via input field: Since input values of type text can have infinite options, the pattern of input values that should be covered by our test cases are as follows:

    a. Username and password field should be tested for the following combinations:

        i. wrong username /correct password
        ii. wrong username / wrong password
        iii. correct username / wrong password
        iv. correct username / correct password

    b. Regular text input should be tested for the following values:

        i. Empty Field
        ii. Extremely long field
        iii. Normal expected value
        iv. Text with special characters (especially HTML text such as <BR> or special wild characters such as: ,/&%#@)

2- Numeric input via an input field: the numeric values that can be entered in a numeric input field are simply indefinite. Thus we should define a set of input patterns (for each input field) that would cover the entire range. First we should define the

domain of input values that are allowed to be entered in those fields. This domain can simply be a *range of values* or a *set of values*. Moreover, we should identify the type of numeric values allowed in this field for example integer, decimal, even numbers, odd numbers ...

After defining the range, we should construct test cases that have numeric input values with the following patterns:

a. Valid values from a valid numeric type and within the domain that is a number within the range or the set defining the domain.
b. Non Numeric values such as text, alphanumeric, an input that contains arithmetic operations, or HTML characters
c. Values of a valid type but outside the domain; that is values that are out side the valid range or outside the set of allowed values.
d. Values of invalid type but of a valid domain.
e. Values of both an invalid type and invalid domains.
f. Empty value
g. Zero values
h. Boundary values for domains that are defined by a range.
i. Negative values

3- Text Options and numeric input via a drop down menu field: input selected from drop down menus has lower probability to generate errors than manual input entered by the user. The best practice is to test each and every option of this selection list. But sometimes, the list can be too long for testing each option of it, and sometimes choosing different options would lead to the same result (such as selecting a country name in a registration form). Thus, we have to manually analyze the available options and come up with a set of selection patterns that would result in all possible outputs. So we have to differentiate the following:

a. If the section values are numeric, then we have to define the domain of values and thus choose values to cover the three patterns: within the domain, outside the domain, and on the boundaries
b. If the selection is non numeric, then we group together the options that will yield the same result upon submitting and then choose a value representing each pattern (group) to be used in a test case.

4- Check box selection: Testing the values for check boxes is as easy as testing whether the checkbox is checked or not. For a group of check boxes, we have to identify the different selection patterns that would result in different output. For each of those patterns, we have to select a value to be used in a test case.

5- Radio buttons: radio button selection is a selection that forces the user to select one value or option from a predefined set. Normally, a radio button group consists of a limited number of options which makes it feasible to test each and every option of it. If the options of a certain selection grow in number, then drop down lists can be used instead of radio buttons group. So the best selection for radio buttons is testing each and every option of the available. In case the options are numerous (which is inefficient for the end user), we can still identify the patterns of input and generate a test case for each of those values.

Now that we have defined an efficient way to select the input of our test cases, we have to use those input selections to select black box test cases. We can write those tests in a list, where each list item is a set containing the page name and the entered values on that page. There is no formal algorithm or fixed technique to be used in order to select the test cases since the number of cases and the details of the tests are specific to the web application itself. Black box testing is much more subjective and depends on the personal skills and knowledge of the tester, and thus it would not be wise to outline a formal testing technique to be followed by the tester as is the case with white box testing. But roughly speaking the optimal test case coverage is a set of test cases that covers all possible combinations of the input values specified in the previous step. An easier and more flexible coverage technique is selecting a set of test cases such that each of the input values defined in the previous step are covered at least once in a certain test case.

6.3.2 The Tool

6.3.2.1 Overview

As outlined above, we should apply the selected test cases on our application using a custom developed tool, which makes it much easier to do regression testing later and in an automated way. This tool which we will be using contains an embedded web browser in it besides other features which makes it feasible for the tester to do the black box testing while navigating the application as a regular web user do and at the same time storing some additional information that would enable him to do quick and automated regression testing later on.

The key idea behind the approach for doing automatic regression testing is that the execution of any web application consists of calling a sequence of web pages from the server to the client and specifying an input for each one. What determines the behavior of the web application is the sequence in which the pages are called and the input values passed to each page. Based on this simple analysis, we can conclude that each time we call the same pages in the same sequence and passing the same parameters, we should receive the same output. While we navigate the application using the tool, this tool should save the sequence of web pages, their input values, and specific output values (specified b the tester). Later on when we want to do regression testing, after modifying the application, the tool should re-execute a subset of the initial set of URLs with the same set of input values and we compare the old and the new output to validate the correctness of the changes.

The reason why I decided to save and compare specific sections of HTML output and not the entire output is because saving and comparing the complete HTML page is inefficient and in many cases turns out to be useless for the following reasons:

- As clarified above, we generate our test cases based on input patterns and not on specific input values, so what we care about here are key html sections that act as indicators and can differentiate between different input patterns regardless of the specific input values themselves. Moreover, it is so common to have a dynamic web page returning different content for two different input values even though those values belong to the same input pattern. Obviously, the entire html page is not a good indicator for this kind of pages. Anyways, even if the entire page works as an indicator of the input in some cases, then it is not efficient to save the whole page and compare it.

- After modifying the existing application, we will definitely have many pages changed and thus their html output modified. If we are to compare the entire html pages, then we will end up with loads of html output to compare and validate even though they are of little importance to our test cases. In this case it would be more efficient to repeat the test cases manually. On the other hand, if we have specific html indicators to compare, then this can be done automatically and the output can be validated manually by the tester.

Even though, specifying html indicators is an efficient technique, yet sometimes it is not enough especially when the modified page includes new html sections that should be validated in the regression testing and thus re-running the tests automatically will not take record of those sections. As a result, we will make it feasible for the tester to force the tool to collect specific entire pages (that will contain the missing indicators) prior to re-running the test cases. After all, having very few html pages to validate is still much better than having a huge set of pages to validate.

6.3.2.2 The Tool's Architecture

This thesis will not be discussing the programming and implementation details of the tool but it will be presenting the architecture, flow of execution, and features of this tool. It is important to mention that all features included in the architecture are feasible and can be implemented and programmed for I will not be including any theoretical procedures that can not be implemented by the existing programming languages.

The tool will be divided into three parts: The browser, the capturing tool and the analyzer and I will be looking at each one alone. It is important to say that the browser and the capture tool have to co-exist within the same screen of the tool's interface because we need to capture data as we navigate the test cases manually the first time. The analyzer which re-executes test and compares the collected output and validates is preferred to be in a separate screen since its functionality is not used at navigation time.

It is important to mention as well that for each test case, all files generated by the tool are stored in a subfolder in order to avoid overlapping of files with same names.

6.3.2.2.1 The Embedded Browser

The embedded browser is used to do the black box testing manually and to collect the needed information automatically for later use. What we need this browser to do is to collect a list of all visited URLs, in addition to all parameters passed to this URL. Parameters passed using the GET command gets concatenated automatically in the end of the URL, and thus they are captured with the URL itself. The other method for passing parameters to a web page is by POSTING the parameters from a form. This information is passed in the header of the request sent by the browser and they are not seen by the end user. Capturing this kind of information is very easy if we use special APIs in advanced programming languages to implement the browser since those APIs have special functions to capture the request headers. A very known API that can be used to implement a browser exist in Microsoft Visual Basic which allows us to open an instance of Microsoft Internet Explorer inside our application and it provides us with a set of functions that makes it feasible to capture all HTTP protocol details sent and received by the browser..

Now that the information is available, we have to save it in a flat text file in a predefined format so that the analyzer can later read the file, reconstruct the request headers and re-execute them automatically. The suggested way to store the collected information is by storing each request on a separate line (the first line the oldest) and dividing each line into three columns as per the following architecture

1- First Column contains the line number or the sequence number, specifying the order in which the respective URL was executed. This number starts at one in the first line of the file and increments by one for each line. The importance of this line number is not to know the sequence of the URL in execution (since this is already known from the order they are listed in the file) but rather for easier reference in the validation process.
2- Second Column: The URL itself as it is sent to the server. This URL could contain extra parameters (query string) attached to its end.
3- The set of parameters sent in the request Header via the form post (the post command in the HTTP protocol). This information will be saved in the same format in which a sequence of parameters is appended in the end of a URL. The parameters are separated by & and each parameter is saved in the following format: *parameter_name=Value*

6.3.2.2.2 The Collector Tool

The collector tool is used while browsing to collect specific HTML sections or indicators to be used later for comparison and analysis in the regression testing phase. The collector's interface coexists with the browser's interface on the same screen. For each new page displayed in the browser, all possible HTML indicators whose values can be captured are displayed in a side screen. The tester can select the indicators he wants to capture.

The challenge in this tool is to be able to find a systematic and smart algorithm which is capable of presenting the tester with the possible indicators to choose from. To solve this issue, I decided to consider each HTML component or HTML tag that can be named in HTML as a candidate for being an indicator. In HTML code, the sections that can be associated with a name are:

1. Form elements which include:
   a. Text input
   b. Radio buttons group
   c. Checkboxes
   d. Drop down lists
2. HTML Tables, Table Rows, Table cells
3. <Div> tags that can enclose text or other HTML sections
4. <Layer> Tags

If a certain HTML or text output which is needed to be used as an indicator, and which is not from any one of the above types, then the web page should be edited prior to testing and this indicator should be enclosed within a <div> or <layer> tag and given a name. It is important to mention that adding a DIV tag will not change the visible output for the end user and thus this change is transparent to the application and can be applied safely.

Now that our collector tool knows which indicators to display, the tester should make sure that all his important indicators are included in named HTML tags prior to testing. Once the tester starts applying the test cases and navigating the pages, the collector interface should list on the screen all available indicators of the current visited page. The interface should allow multiple selections of indicators. For the sake of completeness, sometimes the tester needs to capture the entire HTML page as an indicator, thus the tool should give him the option to do that as well.

After discussing on what indicators the tool should present and how to provide the user with the option to capture them, we have to discuss how this information should be saved. The captured indicators are normally big html sections and thus embedding them all in a single page would not be efficient and might cause confusion and difficulty in analyzing the results later on. The proposed way is to have one file containing the index of all indicators and pointers to other files containing the content of those indicators. The structure of those two files is as follows:

1. Index file: the index file contains a list of all the HTML indicators captured and additional information associated with them, each on a line. The format of the file is as follows:

    a. The first column has the line number of the URL from which the indicator has been captured. This line number is the same number that the browser tool associated with the URL of this page in the URLs file. We use this number to link the two files.
    b. Second column contains the indicators name. If the entire page is captured then the page's name is written in the name of the indicator.

c. Third column contains the indicator's type. This type can be one of the possible types specified above (textfield, div tag, layer…). The type ALL indicates that the entire page is captured.

d. Fourth column contains the filename which contains the content of the html indicator. The naming convention of this file is as follows: pagename_indicatorname_linenumber.txt. For example a sample value for the file name would be "index.asp_logintxt_1.txt".

2. The content file contains the HTML code of the selected indicator as is. No special formatting is required here.

6.3.2.2.3 The Analyzer Tool

The analyzer tool uses the set of files generated by the embedded browser and by the collector tool in order to re-execute the test cases automatically, generate output files, and compare the results presenting us with the difference in indicator values. Those values should be manually analyzed by the tester and any modifications should be done accordingly. However, the tool should not re-execute all the test cases during regression testing but we select a subset of those test cases which are relevant to the changes which we have made. Selecting which test cases to re-execute is purely a subjective process and is done manually by the tester.

Once we have selected a set of test cases for regression testing, the detailed steps of the analyzer tool for each of those cases are as follows:

1- Read the first line in the URL file, and construct an HTTP request and its header using the URL and the post parameters associated with it. This http request should be constructed as per the standard HTTP protocol standards.

2- Submit this http request to the server and receive the response.

3- Check the index file of indicators and identify the list of indicators corresponding to this page that should be captured. The set of indicators can be easily identified because they have the same line number as that of the URL used in constructing the request.

4- Capture the indicator values from this page and store their content in files in the same directory. The main index file is not created again because it is used to point for the original content files as well as for the new content files. The new content files have a naming convention similar to that of the original ones, but they are prefixed with the letters "reg_" which correspond for regression testing. So the content retrieved in regression testing for a certain indicator is saved in the file named reg_pagename_indicatorname_linenumber.txt. For example a sample value for the file name would be "reg_index.asp_logintxt_1.txt".

5- Read the next URL in the URLs file and repeat the same process.

6- After all the URLs have been read and executed, scan all the indicator files in the folder, and compare for each indicator the original file with its counterpart generated by the regression testing. If there is any difference between the content of the files then they are copied to a sub folder named "differences" for later inspection.

Sometimes, while running the black box test case we select some indicator sections from a page which is going to be modified later. However, after modifying this page and in particular adding sections to it, the previously selected HTML indicators became insufficient to determine the correctness of the application, and thus if we let the analyzer tool to run on the previously selected indicators we will have missing indicators and thus we wont be able to validate the exact correctness of our application. To solve this issue, the tester can edit the saved index file, by adding or editing existing records and thus he can modify or add HTML indicators. Off course, the added and changed indicators will appear in the difference folder because they will yield a different result and by this the tester will be obliged to inspect and analyze their results. We can add to our analyzer tool a user interface that allows the tester to add and edit indicators easily instead of editing raw text files.

After running the analyzer tool we will have a set of indicator files that are not matching, which is something normal to have after any change in the application. The role of the tester is to check all the pairs of files in that folder and to compare their content. Sometimes, the difference in content is a way to validate that the application has been changed correctly and in this case the tester should just inspect the files to make sure that all differences are expected. In other cases, the difference in the indicators points to a problem in the changes applied to the application. In both cases those files should be reviewed and inspected by the tester who should locate any problems and act accordingly.

Chapter 7

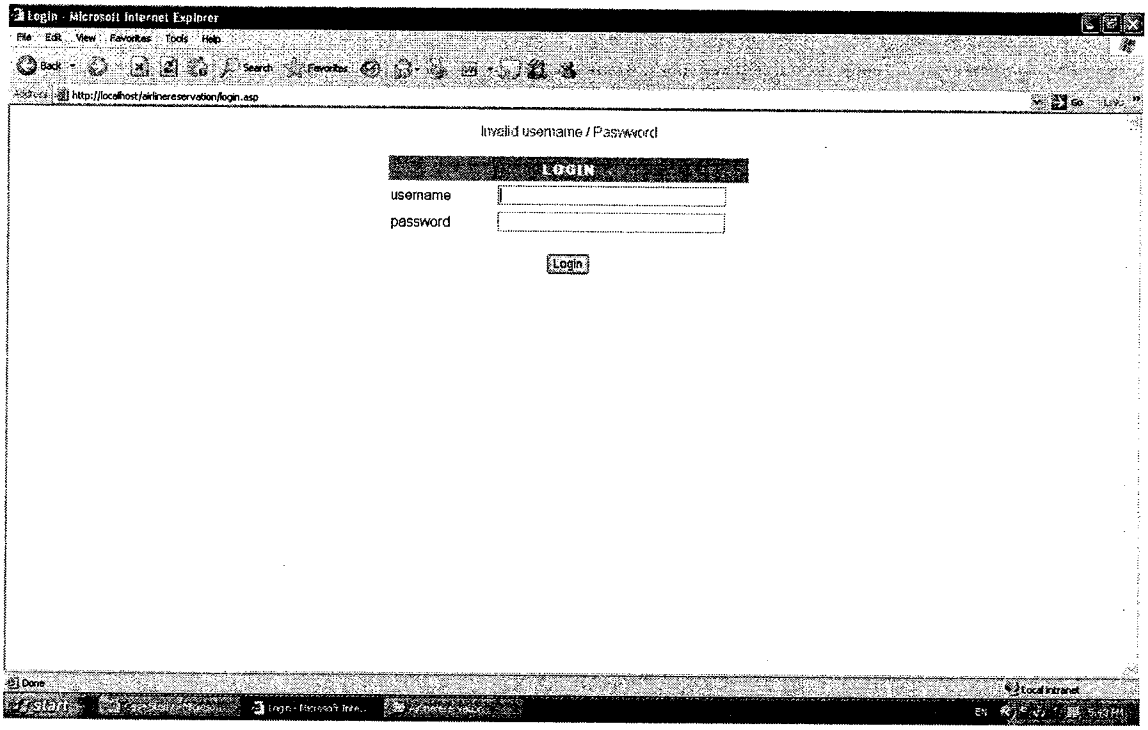
CASE STUDY

In order to further elaborate and support my presented ideas, I will be presenting in this chapter a case study on an example web application. I will be presenting an example web application and I will be applying to it the proposed models and their testing techniques.

7.1 Application Description

The web application I will be using is a simplified version of an application used in airline reservation systems. This custom made application which has only the basic features, allows us to model and test all of the features presented in this thesis in a simplified way. I avoided making a huge and complicated application since that would result in complicated models and long test cases while my aim in this section is to materialize and apply my ideas in the simplest way possible. This web application operates as follows:

The user starts by entering his username and password in the logon page. If the combination of the username and the password are wrong, then the user gets an error message. If the login is successful, the user is redirected to a page providing information about the available flights. (Fig.7.1.1)


Login - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites
http://localhost/airlinereservation/login.asp
Go
Done
Local intranet
start

Invalid username / Password

LOGIN

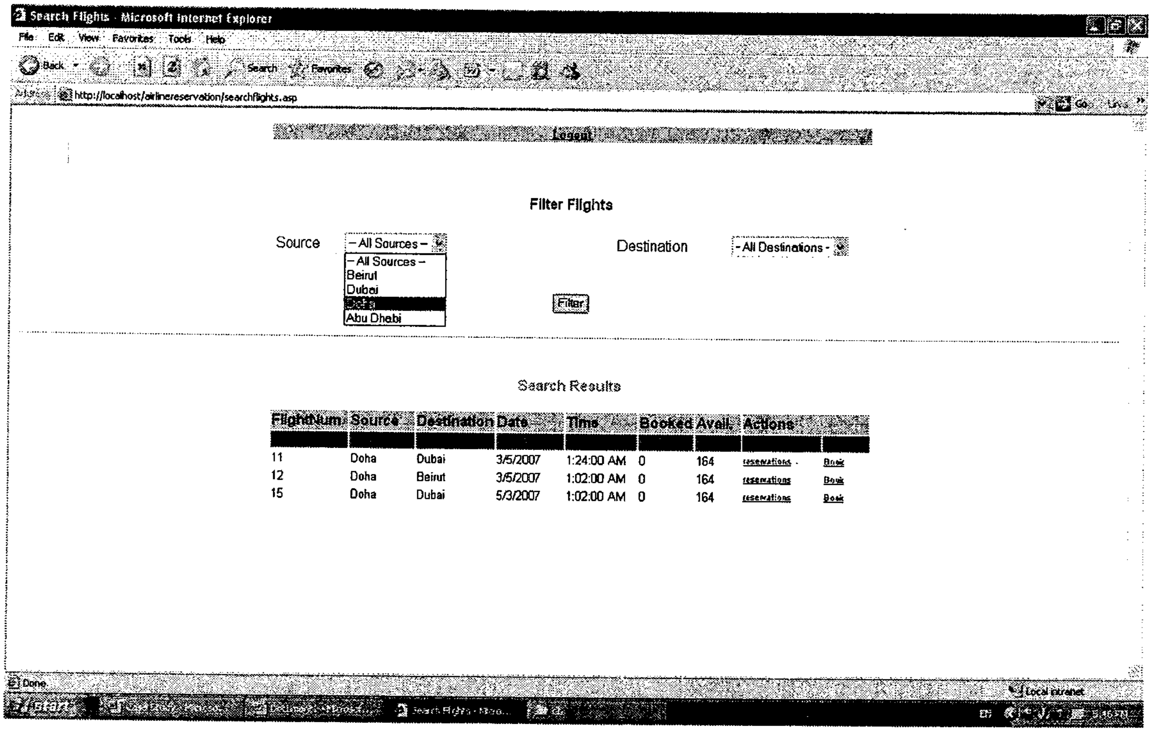
username

password

Login

Fig. 7.1.1 - Snapshot of Login.asp

By default, the page opens with all the available flights listed. This page has a search feature to filter the output. The search criterion filters the output by filtering on the source and the destination of the flights. The resulting search consists of a list of available flights each on a line. Each line contains a brief of the flight information such as source, destination, flight number, departure time. In addition to the listed flight information, for each flight the user is provided with two options: the first is to view the booked reservations, and the other to reserve a place on that flight. The second option to reserve is only enabled when there are still available seats on the flight. Moreover, a fully booked flight is highlighted in red color which makes it easier for the user to identify. This page has a link to the logout page which is the last page to be visited in the application and which causes the user to log out and it deletes his session variable and redirects him to the login page. (Fig. 7.1.2)



Search Flights - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites
Go
Logout
Filter
Done
Local intranet

http://localhost/airlinereservation/searchflights.asp

Filter Flights

Source – All Sources –
– All Sources –
Beirut
Dubai
Doha
Abu Dhabi

Destination - All Destinations -

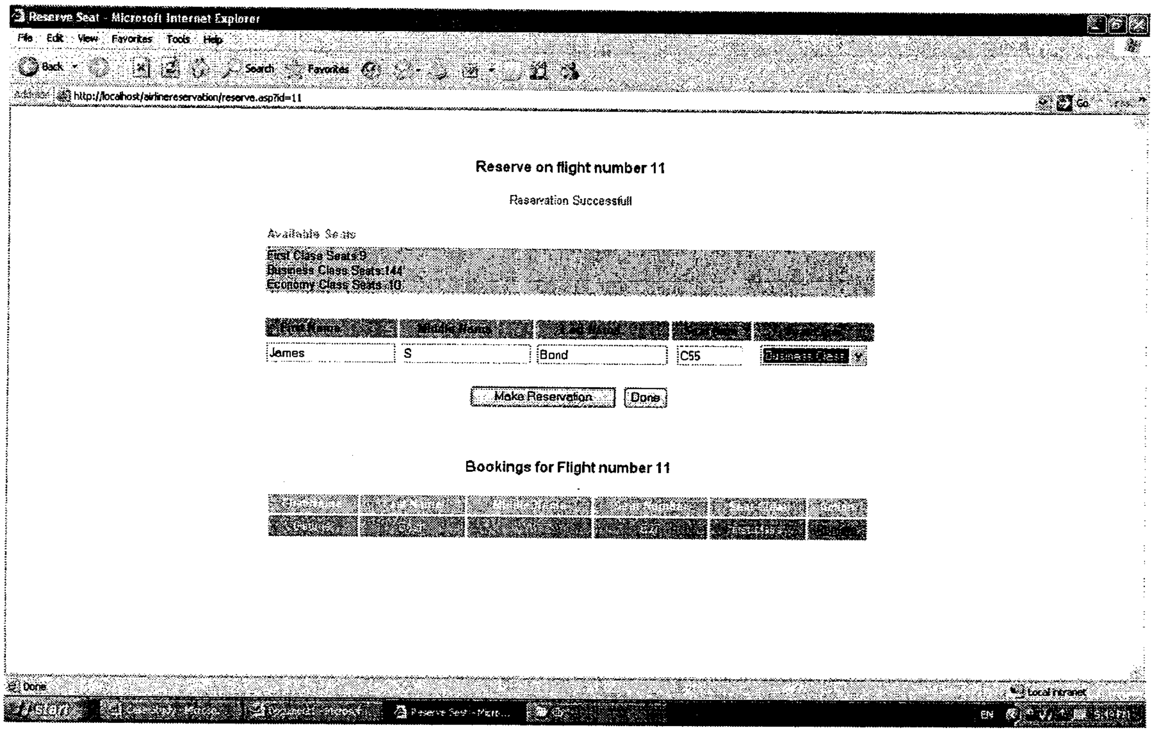
Search Results

FlightNum	Source	Destination	Date	Time	Booked	Avail.	Actions	
11	Doha	Dubai	3/5/2007	1:24:00 AM	0	164	reservations	Book
12	Doha	Beirut	3/5/2007	1:02:00 AM	0	164	reservations	Book
15	Doha	Dubai	5/3/2007	1:02:00 AM	0	164	reservations	Book

Fig. 7.1.2 - Snapshot of searchflights.asp

Clicking the "Book" link transfers the user to a page to create a reservation. On that page, the user has to enter the passenger's information, beside the reserved seat number and the reserved seat class. That page presents information about the detailed available number of seats per class. And it presents a list of the previously booked passengers with the option to cancel the reservations for any of those. Clicking the remove link causes the page to call it self with special URL arguments causing the respective record to be deleted. The user can leave this page by clicking the "Done" button and thus redirected to the main flight search page. (Fig. 7.1.3)

Clicking the "reservations" link transfers the user to a page summarizing the current bookings for this flight. The user has the option to remove any of those bookings by clicking the remove link. Clicking the remove link causes the page to call it self with special URL arguments causing the respective record to be deleted. The user can leave this page by clicking the back button which takes him back to the main flight search page. Each page of the pages mentioned before uses an include file that contains the database connection initialization. This file initiates and opens the connection to the database on each page. (Fig. 7.1.4)


Reserve Seat - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back
Search
Favorites
Address
http://localhost/airlinereservation/reserve.asp?id=11
Go
Make Reservation
Done
Done
Local intranet
start
EN

Reserve on flight number 11

Raservation Successfull

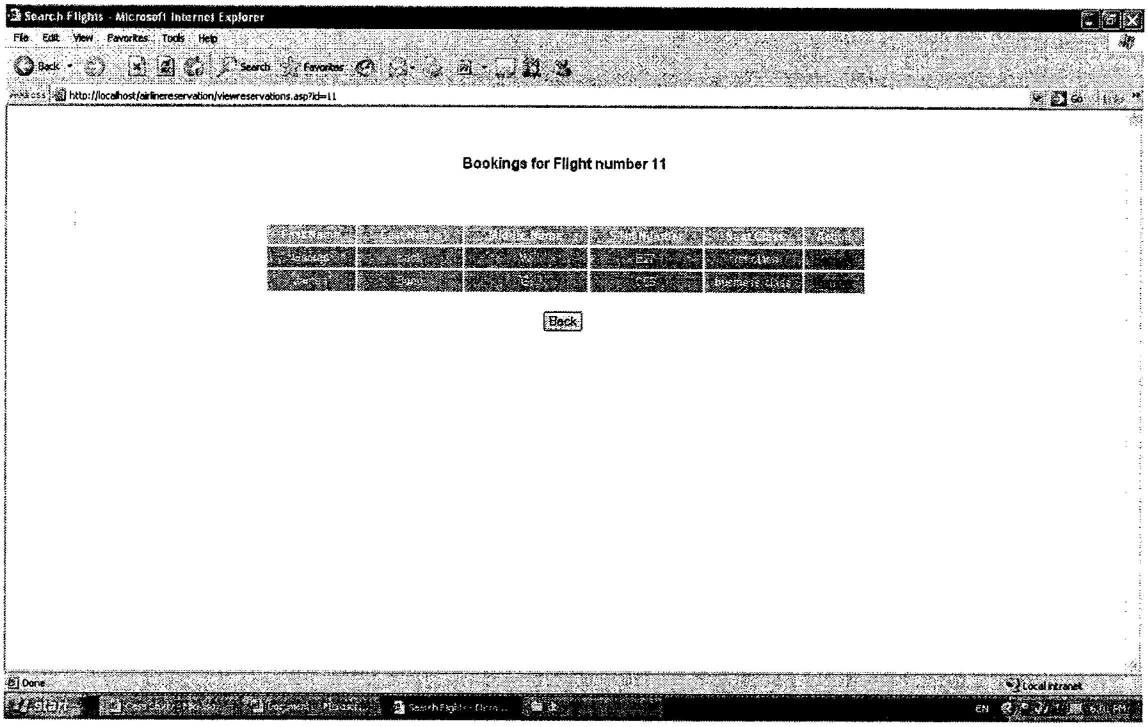
Available Seats

First Class Seats:9
Business Class Seats:144
Economy Class Seats:10

First Name	Middle Name	Last Name	[illegible]	[illegible]
James	S	Bond	C55	Business Class

Bookings for Flight number 11

Fig 7.1.3 - Snapshot of reserve.asp



http://localhost/airlinereservation/viewreservations.asp?id=11

Bookings for Flight number 11

[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
[illegible]	[illegible]	[illegible]	[illegible]	First Class	[illegible]
[illegible]	[illegible]	[illegible]	[illegible]	Business Class	[illegible]

Back

Fig. 7.1.4 - Snapshot of viewreservations.asp

Now that we know how the application behaves, we will briefly describe its architecture and technical information. To start with the programming language, the application is written is ASP 3.0 which is the most popular web development language so far. I decided to run the application on a cluster of redundant Intel servers running Microsoft Windows 2000 with Microsoft IIS as a Web Application Server. The windows cluster is used for redundancy and not for load balancing.

The database used can be either an SQL Server database or a Microsoft access database and this varies with the expected size of the database and with the required response time. Off course for huge databases with quick response time SQL Server is used, while for desktop editions of the application Microsoft Access is more than enough. Changing the database is as easy as changing the configuration in the connection file, included in all the other pages, without changing anything in our pages. If this connection file initializes and opens an SQL database, then our application reads and writes to this database. Similarly, if this file initializes an MS Access database then our application reads and writes to this database. In both cases, the database can reside on the same machine as the application or on a separate machine. I decided to have an SQL Server database running on a separate windows machine.

7.2 The Architectural Environment Model

7.2.1 The Model

As we mentioned earlier, the application runs on a cluster of redundant cluster of two servers, and the database resides on a separate server. We will call the two servers running IIS S1, and S2 respectively. We will call the IIS application server (software sever) running on both machines SV1 and SV2 respectively. We will call the machine running the SQL database S3 and we will denote the Microsoft SQL Server Installation (software server) by SV2. The IIS servers send database requests to the database and receive responses from it over the TCP/IP protocol. We will denote those communication channels by Cm1 and Cm2 respectively. The Software libraries that are required to be installed on the servers are limited to the SQL ODBC driver that should be installed on both application servers to allow communication with the database.

Based on the above information, we will formally define the following sets to represent our architecture:

- Set S contains all the hardware servers. S= {(S1; Windows 2003, intel_x86), (S2; Windows 2003, intel_x86), (S3; Windows 2003, intel_x86)}

- Set SV contains all the software servers. **SV**= {(sv1; IIS; S1), (sv2; IIS; S2), (sv3; Microsoft SQL Server; S3)}
- Sets Cm1 and Cm2 represent the communication channels from and to the database.
  - **Cm1**= {Sv1, Sv3, TCP/IP, db_q}
  - **Cm2**= {Sv3, Sv1, TCP/IP, db_rs}

- Set C to represent the clusters. **C**= {(Sv1; Sv2)}
- Set LI to represent the required software libraries. **L1**={(S1;SQL_ODBC_DRIVER), (S2;SQL_ODBC_DRIVER)}

Following the drawing convention we discussed in the section for modeling architectural environment, the diagram of our architecture will look the diagram in Fig. 4.2 below:

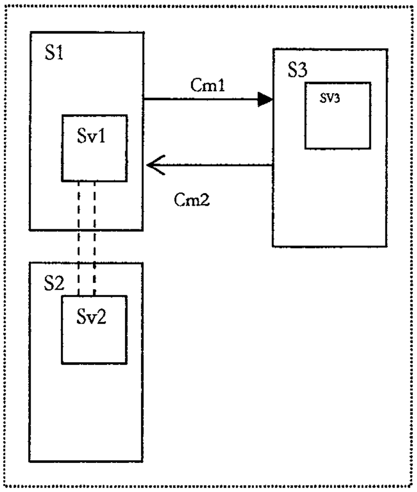


Fig 7.2 - Case Study – Architectural Model

7.2.2 Testing Techniques

In order to test the quality of our architecture, we have to apply the eight test techniques defined earlier for this model. The architecture should pass the four tests for it to work. The remaining four test are used to evaluate other quality attributes of our architecture

1- Compatibility of the operating system with the hardware
2- Compatibility of operating System with installed software servers
3- Compatibility of communication protocols
4- Compatibility of application communication libraries
5- Analysis of the messages exchanged
6- Level of redundancy
7- Level of load balancing
8- Level of scalability

In the following sections, we will apply each of the mentioned tests alone.

- For the first test, the set of OS / HW compatibility is denoted by T1. Since Windows is compatible with Intel processors, then T1= {(S1; 1), (S2; 1), (S3; 1)}. Since all servers in T1 are paired with the value 1, then our architecture passed the first test.

- The Second test analyzes the Operating systems with the installed software and represents them in the set T2. Since all application servers sv1, sv2, and sv3 are compatible with Windows servers then T2={(sv1;1), (sv2;1), (sv3;1)}. Again since all set elements are paired with the value 1, then we can safely say that our architecture passed the second test.

- The third test verifies the feasibility of the communication channels Cm1 and cm2. Since both channels are based on TCP/IP and since all servers are running windows which supports TCP/IP by default, then both connections are feasible and this test can be represented by T3={(Cm1,1);(cm2,1)}. Similarly, our architecture passed the third test.

- The fourth test verifies the existence of the required software extensions on the servers and this is represented by a set of Triplets T4. Again, for our architecture to pass this test, all triplets should be paired with the value one. Since both IIS servers have the SQL ODBC drivers installed on them then the set is: T4= {(S1; SQL_ODBC_DRIVER,1), (S2, SQL_ODBC_DRIVER,1)} which implies that our architecture passed the fourth test.

- The fifth test evaluates the efficiency of the messages exchanged per each communication channels. Since both communication channels are based on TCP/IP and they are on the LAN, then both communication channels have optimal efficiency and we can safely give them a high score of 9/10. So the fifth test can be represented by T5 = {(C1; 9), (C2; 9)}. Obviously the test indicates high efficiency for our architecture.

- The sixth test checks for the redundancy of the servers. The redundancy value uses the following formula as stated earlier: So rd= 1 – (|SV'|/|SV|) where SV' is the set of software servers non clustered and non load balanced, where SV is the set of all software servers. So for our architecture, rd= 1- 1/3=2/3 =0.66 which is a sign of good redundancy since it is above 0.5.

- The seventh test checks the quality of the load balancing attribute of our architecture. It uses the following formula: ldb= 1 – (|SV"|/|SV|) where Sv' is the set of non load balanced servers and SV is the set of all servers. In our architecture ldb=1-1=0.

Obviously, the value indicates that we have no load balancing in our architecture which requires our attention.

- The eighth test checks the level of scalability of our architecture. We test salability based on the following formula So Scalability scl=|S'|/|S| where S' is the set of servers that can be scaled and s is the set of all servers. In our example, IIS servers can be scaled as much as we want. On the other hand, SQL server can not be expanded on different machines. So scl=2/3=0.66 which is a good value since it is above 1.5 but we should be aware that the database cannot be scaled which might create a bottle neck in the future.

7.3 The Client Side Model

7.3.1 The Model

This model considers all the pages included in the application a seen for the end user. I will start by listing all the web pages in our application which are as follows:

1. Login.asp: This page has a form allowing the user to enter his login information. The page submits the arguments to itself and it redirects to the page "searchflights.asp" if the login is successful. The page displays an error message if the username and password are wrong and it does not redirect to any other page.

2. Searchflights.asp: This page displays the available flights, and it has a form to filter on those flights. This form submits the page to itself to display the filtered values. On this page, each flight has 2 links corresponding with it. One link points to the page

"reserve.asp" which allows the user to make a new reservation. The second link points to the page "viewreservations.asp" that displays all the available reservations for a certain flight. Each of those two links have the id of the corresponding flight passed as an argument to the other two pages. This page has a link to the page "logout.asp" which allows the user to exit the application.

3. Reserve.asp: This page allows the user to make a new reservation. It has a form which allows the user to enter the reservation information. This form submits to the same page and saves the entered information. This page lists all the available reservations as well. The user has the option to delete a reservation by clicking on a corresponding link for the desired reservation. This link calls the same page with special arguments, to instruct the page to delete the desired booking. The user can leave this page by clicking the "Done" button which takes him to the page "Searchflightspage.asp"

4. Viewreservation.asp: This page lists all the reservations for the selected flight. The user has the option to cancel any reservation by clicking on a link that calls the page itself with specific arguments attached to the URL. The user can leave this page by pressing the back button which takes him to the page "searchflights.asp"

5. Logoutpage: This page is the exit page of our application and it can only be called from the page "searchflights.asp". When called, this pages deletes the session variables f the logged in user and it redirects to the login page.

Based on the above information, the graph of the client side model will look like Fig 7.3: I have named the pages by P1, P2, P3, P4, and P5 for easier reference in the test cases.

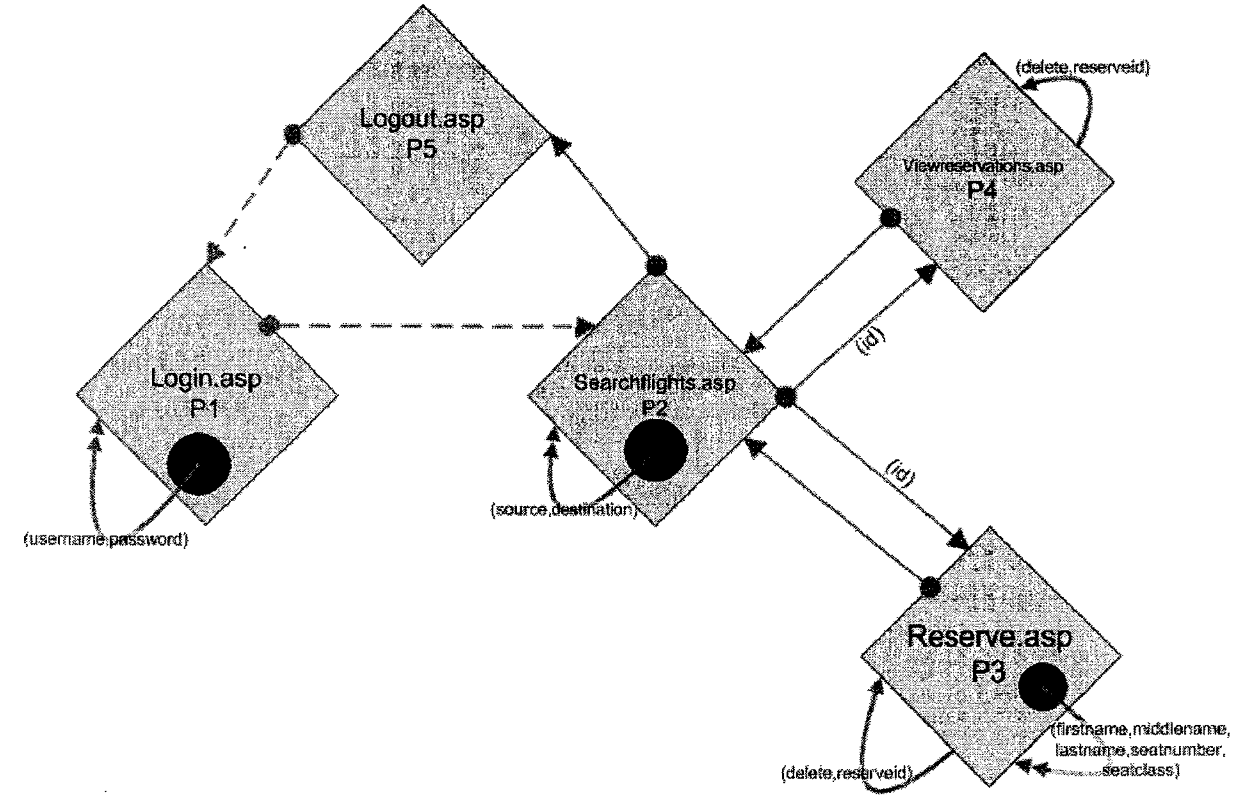


Fig 7.3 - Case Study – Client Side Model

7.3.2 Testing Techniques

After constructing our web application graph, we have to follow the testing techniques discussed earlier in order to validate the correctness of our application and to make sure that it maintains high quality attributes. The tests to be conducted are as follows:

1- Tests for orphaned pages
2- Tests for broken Links
3- Tests for dead end pages
4- Tests for Parent child sequencing

Before starting our tests, we will define the sets N of nodes and E of edges (as per discussed earlier) because they will be used in all the tests.

N= {p1, p2, p3, p4, p5} and

E={(e1,p1,p1), (e2,p1,p2), (e3,p2,p2), (e4,p2,p3), (e5,p3,p3), (e6 ,p3,p3), (e7,p3,p2), (e8,p2,p4), (e9,p4,p4), (e10,p4,p2), (e11,p2,p5), (e12,p5,p1)}

The test for orphaned pages is conducted by checking all nodes in N (except for the start page p1) and verifying that each node has at least one corresponding edge in E where n is the third item in the triplet. The pages p2, p3, p4, and p5 has the edges e2, e4, e8, and e11 satisfying this condition correspondingly. Thus the set N' of orphaned pages is empty and we do not have any orphaned pages

The second test checks for broken links. This is done by trying to create a set of test cases that traverses all nodes such that each node is visited at least once by a certain test case. Once we obtain this set of test paths, then we have no broken links. In our example we can have the following 2 paths that satisfy the all node coverage criterion which verify that our application has no broken links:

- p1, p2, p3, p2, p5, p1
- p1, p2, p4, p2, p5, p1

Hence, our application passed the second test.

The third test checks for dead end pages or in other words pages that do not give the user an option to leave them. This test is conducted by inspecting all nodes in N (except the exit page p5) and verifying that each node n has a corresponding edge (triplet) in E, such that n is the second item in the triplet. The nodes p1, p2, p3, and p4 has the edges e2, e3, e7, and e10 satisfying this condition. Thus the set D of dead end pages is empty and our application does not have any page that leads the user to a navigational dead end where he finds him self forced to use the browser's back button.

The fourth test checks for the parent child sequence requirements in our application. As per the testing analysis discussed earlier, we need to identify two sets of parent child requirements one direct corresponding to direct sequence of two pages and one indirect corresponding to the order in which two pages are visited but not necessarily directly after each other.

The direct set of requirements is that p3 and p4 should follow p2, since p2 passes arguments containing the reservation id to those two pages, and this value is mandatory for the operation of the two pages.
So the set of direct requirements P1= {(p2, p3), (p2, p4)}

To verify this we should inspect E to make sure that there is an Edge from p2 to p3 and from p2 to p4. This is true since we have the edges e4 and e8. Moreover we should make sure that p3 and p4 are not reached from any page other than p2.this is true as well since we cannot find any edge in E arriving to p3 and p4 except from p2 and from the pages themselves. So our application satisfies all the direct parent child relationships.

The indirect set of requirements can be limited in our example to requiring the login page p1 to precede any other page in our application for the user is required to log in before using the application. So the set of indirect requirements P2= {(p1, p2), (p1, p3), (p1, p4), (p1, p5)}.

The way to conduct this test is to validate that for every requirement in P2, each path traversing the child should have traversed its parent at some point earlier. Since checking all available paths is impossible we have chosen to derive a set of test cases that satisfies the

level-2 coverage criterion as defined earlier. This coverage criterion guarantees that all nodes have been visited at least once and that all cycles have been traversed at least once. The set of derived test cases from our example that satisfies the level-2 coverage are as follows:
p1,p2,p2,p3,p3,p2,p5
p1,p2,p4,p4,p2,p5

Analyzing the above test cases verifies that our indirect parent child sequences are all satisfied and we can safely say that our application passed this test. Sure, if we still have doubts we can keep on generating additional paths and analyzing them until we find a path violating our requirements or until we are satisfied by the test results.

7.4 The Server Side Model

7.4.1 The Model

In the server programs model, I will be considering all dynamic pages and server components that generate HTML dynamically. Thus I will be considering the pages: "login.asp", "searchflights.asp", "viewreservations.asp" and "reserve.asp". The pages "logout.asp" and "connection.inc.asp" will not be analyzed because they do not generate any HTML output and thus do not have any atomic sections. The logic of the server side code can be tested using classical software techniques and thus will not be considered here. In this section, I will identify the atomic and composite sections of each of the four pages, I will derive the composition rule for each of them, and I will draw the corresponding graph. After constructing the graphs, I will be applying the testing techniques suggested earlier in this thesis to test the server side programs.

In the following sections I will be presenting the source code of each of the four pages in the form of tables and I will be specifying where the atomic sections begin and end. Since we need sometimes to have an atomic section to represent empty output, we will be using the atomic section named "e" to represent an empty atomic section (that gives no output). For example "e" is used in representing an "If statement" without an "else" which is an optional output; this is done by creating a selection between the atomic section of the "if statement" and the empty selection. The empty atomic section can be used in many similar scenarios so we are going to include it by default in all pages.

Moreover, I am going to represent the external components that are included in our webpage by Si. Since this is an inclusion of an external component and since we are not interested in analyzing its details, we refer to the whole component as one entity regardless of what is inside it. For example the include file "connection.inc.asp" is represented by S1 in the pages below.

7.4.1.1 Login.asp

In this section I will analyze the atomic sections of the page login.asp and I will be constructing the corresponding WCG diagram. The atomic sections are as follows (Table 7.4.1)

	E

<%
if request.form("submit")<>"" then
if request.form("username")="admin" and request.form("password")="adminpass" then
session("user")="admin"
response.redirect "searchflights.asp"
 	
else
err_msg="Invalid username / Paswword"
end if
end if
 	
%>	

<html> <head>	P1

<meta http-equiv="Content-Language" content="en-us"> <meta http-equiv="Content-Type" content="text/html; charset=windows-1252"> <title>Login</title> </head> <body> <div align="center"> <form method="POST" action="login.asp">	
<div align="center"><font color="#FF0000" face="Arial"><%=err_msg%>	P2

</font><table border="0" width="413" cellpadding="2" id="table1">
        <tr bgcolor="#808080">
            <td colspan="2">
            <p align="center"><b>
            <font face="Arial Black" color="#FFFFFF">LOGIN</font></b></td>
        </tr>
        <tr>
            <td width="116"><font face="Arial">username</font></td>
            <td><font face="Arial">
            <input type="text" name="username" size="39"></font></td>
        </tr>
        <tr>
            <td width="116"><font face="Arial">password</font></td>
            <td><font face="Arial">
            <input type="password" name="password" size="39"></font></td>
        </tr>
      </table>
    </div>
    <p><font face="Arial">
    <input type="submit" value="Login" name="submit"></font></p>
  </form>
</div>
</body>
</html>
	P3

Table 7.4.1 - Case Study - Atomic sections of Login.asp

The composition rule C1 for the page “login.asp” is C1=P1.P2.P3 |=> S1 | e

The login page redirects conditionally to the page "searchflights.asp" only if the login is successful. If the login fails, the page does not redirect to anywhere. This is represented in our graph below by drawing a selection from P3, between an empty atomic section (it does nothing) and a server redirect.

The graph of this page looks like the picture in Fig 7.4.1:

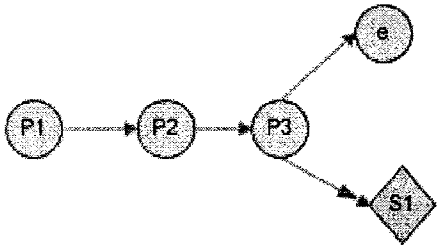


Fig. 7.4.1 - Case Study – Graph for server component login.asp

7.4.1.2 Searchflights.asp

In this section I will analyze the atomic sections of the page searchflights.asp and I will be constructing the corresponding WCG diagram. The atomic sections for this page are as follows (Table 7.4.2)

	e
<!--#include file="connection.inc.asp"-->	S2
<html> <head> <meta http-equiv="Content-Language" content="en-us"> <meta http-equiv="Content-Type" content="text/html; charset=windows-1252"> <title>Search Flights</title>	P1

```
</head>

<body>

<center>

<table width="683">
<tr bgcolor="#99CCFF">
        <td bgcolor="#C0C0C0">
        <center><a href="logout.asp"><b><font face="Arial" size="2">Logout</font></b></a></center></td>
         </tr>

</table>
</center>

<form name="form1" method="POST" action="" onsubmit="javascript:return validate_form()">

        <p align="center">&nbsp;</p>
        <p align="center"><b><font face="Arial">Filter Flights</font></b></p>
        <center>
        <table border="0" width="683" height="29">

                <tr>
                        <td height="29" width="57">
                        <p align="center"><font face="Arial">Source </font> </td>
                        <td height="29" width="148">
                        <p align="center"><font face="Arial"><select size="1" name="source">
                        <option value=""> -- All Sources --</option>
                        <option>Beirut</option>
                        <option>Dubai</option>
                        <option>Doha</option>
                        <option>Abu Dhabi</option>
                        </select></font></td>
                        <td height="29" width="112">
                        <p align="center">&nbsp;</td>
                        <td height="29" width="146">
                        <p align="center">
                        <font face="Arial">Destination</font></td>
                        <td height="29" width="146" colspan="2">
                        <font face="Arial">
                        <select size="1" name="destination">
```

```
                    <option value=""> - All Destinations - </option>
                    <option>Beirut</option>
                    <option>Dubai</option>
                    <option>Doha</option>
                    <option>Abu Dhabi</option>
                    </select></font></td>
            </tr>
        </table>
        </center>
        <p align="center"><font face="Arial">
        <br>
        <input type="submit" value="Filter" name="submit" ></font></p>
</form>
<hr>
<p align="center"><font face="Arial" color="#996633">
<%
source=request.form("source")
destination=request.form("destination")
sql_get_all="select * from flights where destination like '%" &destination& "%' and source like '%" &source& "%'"
set rs_get_all=conn.execute(sql_get_all)
%>


&nbsp;&nbsp;&nbsp;&nbsp; <b><br>
Search Results</b></font><div align="center">
&nbsp;<table width="683">
<tr bgcolor="#99CCFF"><td width="90"><b><font face="Arial">FlightNum</font></b></td>
        <td width="82">
        <b><font face="Arial">Source</font></b></td> <td width="87"> <b><font
face="Arial">Destination</font></b></td>
        <td width="87">
        <b><font face="Arial">Date</font></b></td> <td width="97"> <b><font face="Arial">Time</font></b></td>
<td>
        <b><font face="Arial">Booked</font></b></td> <td width="53">
        <b><font face="Arial">Avail.</font></b></td> <td width="180" colspan="2">
        <b><font face="Arial">Actions</font></b></td> </tr>
```

```
<%do while not rs_get_all.eof
        booked=0
        available=0
        is_full="false"
        sql_booked="select count(id) as thesum from reservations where flightnumber=" & rs_get_all("flightnumber")
```

set rs_booked= conn.execute(sql_booked) if not rs_booked.eof then booked=rs_booked("thesum") else: booked=0 sql_available="select (firstclass_seats + economyclass_seats + businessclass_seats) as av from flights where flightnumber=" & rs_get_all("flightnumber") set rs_av = conn.execute(sql_available) if not rs_av.eof then available= rs_av("av") else: available=0 if booked - available= 0 then is_full="true" %>	
<tr	P2
<%if is_full="true" then response.write "bgcolor='#FF0000'"	P3
:else response.write "bgcolor='#FFFFFF'"%>	P4
><td width="90"> <font face="Arial" size="2">	P5
<%=rs_get_all("flightnumber")%>	P6
</font></td><td width="82"><font face="Arial" size="2">	P7
<%=rs_get_all("source")%>	P8
</font></td> <td width="87"> <font face="Arial" size="2">	P9
<%=rs_get_all("destination")%>	P10
</font></td> <td width="87"> <font face="Arial" size="2">	P11
<%=rs_get_all("flightdate")%>	P12
</font></td> <td width="97"> <font face="Arial" size="2">	P13
<%=rs_get_all("flighttime")%>	P14
</font></td> <td> <font face="Arial" size="2">	P15
<%=booked%>	P16
</font></td> <td width="53"> <font face="Arial" size="2">	P17
<%=available%>	P18
</font></td> <td width="89"> <font face="Arial" size="1"> <a href="viewreservations.asp?id=	P19
<%=rs_get_all("flightnumber")%>">	P20
reservations</a> </font></td> <td width="56"> <font face="Arial" size="1">	P21
<%if is_full<>"true" then%>	
<a href="reserve.asp?id=	P22
<%=rs_get_all("flightnumber")%>">	P23
Book</a>	P24
<%else%>	
Full	P25
<%end if%>	

</font></td></tr>	P26
<%rs_get_all.movenext Loop Conn.close()%>	
</table> <p align="center"><font face="Arial"> </font></div> <font face="Arial"> </font> </body> </html>	P27

Table 7.4.2 - Case Study - Atomic sections of searchflights.asp

The composition rule C2 for "searchflights.asp" is

C2=S2|->p1.(p2.(p3|p4).p5.p6.p7.p8.p9.10.p11.p12.p13.p14.p15.p16.p17.p18. p19.p20.p21.(p22.p23.p24)|p25).p26)*.p27

The graph of this page looks as follows (Fig. 7.4.2)

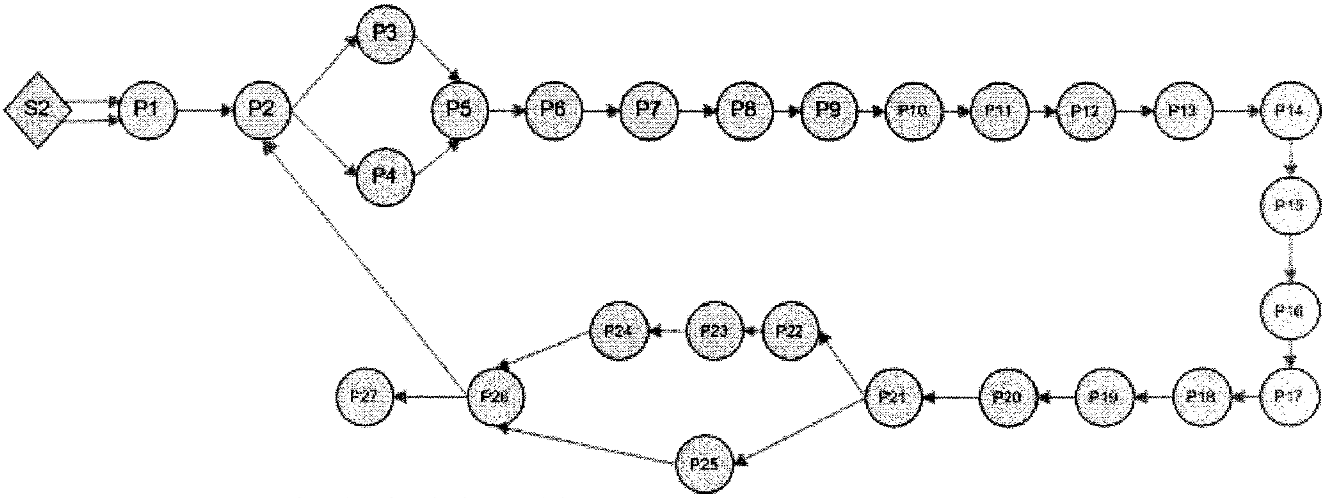


Fig. 7.4.2 - Case Study – Graph for server component searchflights.asp

7.4.1.3 Viewreservations.asp

In this section I will analyze the atomic sections of the page viewreservations.asp and I will be constructing the corresponding WCG diagram. The atomic sections are as follows (Table 7.4.3)

	e
<!--#include file="connection.inc.asp"-->	S2
<html> <head> <meta http-equiv="Content-Language" content="en-us"> <meta http-equiv="Content-Type" content="text/html; charset=windows-1252"> <title>Search Flights</title> </head> <body>	P1
<% ' delete a reservation if request.querystring("delete") = "true" then sql_del="delete from reservations where id=" & request.querystring("reservid") conn.execute(sql_del) end if sql_bookings="select * from reservations where flightnumber=" & request.querystring("id") set rs_bookings= conn.execute(sql_bookings) %>	
<form name="form1" method="POST" action="" onsubmit="javascript:return validate_form()"> <p align="center">&nbsp;</p> <p align="center"><b><font face="Arial">Bookings for Flight number	P2

<%=request.querystring("id")%>	P3
</font> </b></p> <p align="center">&nbsp;</p> <center> <table border="0" width="683" cellspacing="3" cellpadding="2"> <tr> <td height="21" width="101" align="center" bgcolor="#FF9966" valign="top"> <p align="center"><b><font color="#FFFFFF" face="Arial" size="2">First Name</font></b></td> <td height="21" width="121" align="center" bgcolor="#FF9966" valign="top"> <p align="center"><b><font color="#FFFFFF" face="Arial" size="2">Last Name</font></b></td> <td height="21" width="142" align="center" bgcolor="#FF9966" valign="top"> <p align="center"><b><font face="Arial" color="#FFFFFF" size="2">Middle Name</font></b></td> <td height="21" width="128" align="center" bgcolor="#FF9966" valign="top"> <p align="center"> <b><font color="#FFFFFF" face="Arial" size="2">Seat Number</font></b></td> <td height="21" width="110" align="center" bgcolor="#FF9966" valign="top"> <b><font color="#FFFFFF" face="Arial" size="2">Seat Class</font></b></td> <td height="21" width="64" align="center" bgcolor="#FF9966" valign="top"> <b><font face="Arial" color="#FFFFFF" size="2">Action</font></b></td> </tr>	P4
<%do while not rs_bookings.eof%>	
<tr> <td height="23" width="101" align="center" bgcolor="#808080"> <font color="#FFFFFF" size="2" face="Arial">	P5
<%=rs_bookings("firstname")%>	P6
</font></td> <td height="23" width="121" align="center" bgcolor="#808080"> <font color="#FFFFFF" size="2" face="Arial">	P7
<%=rs_bookings("lastname")%>	P8
</font></td><td height="23" width="142" align="center" bgcolor="#808080"><font color="#FFFFFF" size="2" face="Arial">	P9
<%=rs_bookings("middlename")%>	P10

</font></td><td height="23" width="128" align="center" bgcolor="#808080"><font color="#FFFFFF" size="2" face="Arial">	P11
<%=rs_bookings("seatnumber")%>	P12
</font></td><td height="23" width="110" align="center" bgcolor="#808080"><font color="#FFFFFF" size="2" face="Arial">	P13
<%=rs_bookings("seatclass")%>	P14
</font></td><td height="23" width="64" align="center" bgcolor="#808080"><font face="Arial" size="2"><a href="viewreservations.asp?delete=true&reservid=	P15
<%=rs_bookings("id")%>	P16
&id=	P17
<%=request.querystring("id")%>">	P18
Remove</a></font></td></tr>	P19
<%rs_bookings.movenext loop conn.close%>	
</table> </center> <palign="center"><input type="button" value="Back" name="B1" onclick="javascript:location.replace('searchflights.asp')"><font face="Arial"> </font></p> <p aiign="center">&nbsp;</p> </form> </body> </html>	P20

Table 7.4.3 - Case Study - Atomic sections of viewreservations.asp

The Composition rule C3 for the page "viewreservations.asp" is as follows:

C3= S2|->p1.p2.p3.p4.(p5.p6.p7.p8.p9.p10.p11.p12.p13.p14.p15.p16.p17.p18.p19)*.p20

The graph for this page looks as follows (Fig. 7.4.3)

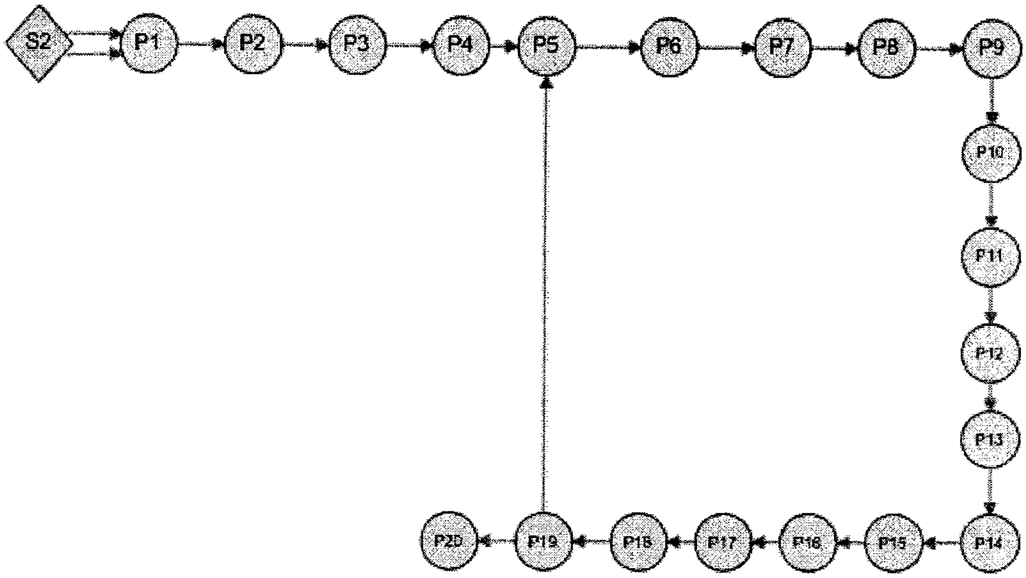


Fig 7.4.3 - Case Study – Graph for server component viewreservations.asp

7.4.1.4 Reserve.asp

In this section I will analyze the atomic sections of the page reserve.asp and I will be constructing the corresponding WCG diagram. The atomic sections of this page are as follows (Table 4.4.4)

	E
<!--#include file="connection.inc.asp"-->	S2
<html> <head> <meta http-equiv="Content-Language" content="en-us"> <meta http-equiv="Content-Type" content="text/html; charset=windows-1252"> <title>Reserve Seat</title> <script language="javascript"> function validate_form() { if ((document.form1.firstname.value=="") ||(document.form1.lastname.value=="") ||(document.form1.middlename.value=="") ||(document.form1.seatnumber.value=="") ||(document.form1.seatclass.value=="none")) {	P1

```
alert("All Feilds are mandatory");
return false;
}

}
</script>
</head>
<body>
```

```
<%

' delete a reservation
if request.querystring("delete") = "true" then
sql_del="delete from reservations where id=" & request.querystring("reservid")
conn.execute(sql_del)
end if




if request.form("submit") <>"" then

sql_chk="select * from reservations where flightnumber=" &request.querystring("id")& " and seatnumber='" &
replace(request.form("seatnumber"),"'","''")& "' and seatclass='" & request.form("seatclass")& "'"
set rs_chk = conn.execute (sql_chk)

if rs_chk.eof then

sql_reserve="insert into reservations(firstname,middlename,lastname,seatnumber,seatclass,flightnumber) values('"&
replace(request.form("firstname"),"'","''")&"','"&                 replace(request.form("middlename"),"'","''")&"','"&
replace(request.form("lastname"),"'","''")&"','"&                  replace(request.form("seatnumber"),"'","''")&"','"&
request.form("seatclass")&"'," & request.querystring("id")&")"
'response.write (sql_reserve)
response.flush
'response.write (sql_reserve)
            conn.execute (sql_reserve)
            err_msg="Reservation Successfull"

else

err_msg="Seat Already booked to someone Else. Please try again later"
```

```
' if chk.eof
end if

end if ' if submitted

' select available
sql_total="select firstclass_seats , economyclass_seats , businessclass_seats from flights where flightnumber=" &
request.querystring("id")
set rs_tot = conn.execute(sql_total)

tot_first=0
tot_bus=0
tot_econ=0

if not rs_tot.eof then

tot_first=rs_tot("firstclass_seats")
tot_bus=rs_tot("businessclass_seats")
tot_econ=rs_tot("economyclass_seats")

end if



sql_bookings_first="select count(*) as cnt from reservations where seatclass='first class' and flightnumber=" &
request.querystring("id")
set rs_bookings_first= conn.execute(sql_bookings_first)

sql_bookings_bus="select count(*)as cnt from reservations where seatclass='business class' and flightnumber=" &
request.querystring("id")
set rs_bookings_bus= conn.execute(sql_bookings_bus)

sql_bookings_econ="select count(*) as cnt from reservations where seatclass='economy class' and flightnumber=" &
request.querystring("id")
set rs_bookings_econ= conn.execute(sql_bookings_econ)

if not rs_bookings_first.eof then booked_first=rs_bookings_first("cnt") else: booked_first=0
if not rs_bookings_bus.eof then booked_bus=rs_bookings_bus("cnt") else: booked_bus=0
if not rs_bookings_econ.eof then booked_econ=rs_bookings_econ("cnt") else: booked_econ=0

ava_first=tot_first - booked_first
```

ava_bus= tot_bus - booked_bus
ava_econ = tot_econ - booked_econ
%>	

<form name="form1" method="POST" action="" onsubmit="javascript:return validate_form()"> <p align="center">&nbsp;</p> <p align="center"><b><font face="Arial">Reserve on flight number	P2
<%=request.querystring("id")%>	P3
</font></b></p> <p align="center"><font color="#FF0000" size="2" face="Arial">	P4
<%=err_msg%>	P5
</font></p><center> <table border="0" width="695" cellspacing="3" cellpadding="2"> <tr><td height="21" width="685" align="center" valign="top" colspan="5" > <p align="left"><b><font face="Arial" size="2" color="#FF6600">Available Seats</font></b></td></tr> <tr bgcolor="#C0C0C0"><td height="21" width="685" align="center" valign="top" colspan="5"> <p align="left"><font face="Arial" size="2">First Class Seats:	P6
<%=ava_first%>	P7
  Business Class Seats:	P8
<%=ava_bus%>	P9
  Economy Class Seats:	P10
<%=ava_econ%>	P11
</font></td> </tr> <tr> <td height="21" width="101" align="center" valign="top"> </td> <td height="21" width="121" align="center" valign="top"> </td> <td height="21" width="121" align="center" valign="top"> </td> <td height="21" width="85" align="center" valign="top"> </td> <td height="21" width="133" align="center" valign="top"> </td> </tr> <tr>	P12

<td height="21" width="101" align="center" bgcolor="#808080" valign="top"> <p align="center"><b><font color="#0000FF" face="Arial" size="2">First Name</font></b></td> <td height="21" width="121" align="center" bgcolor="#808080" valign="top"> <b><font face="Arial" color="#0000FF" size="2">Middle Name</font></b></td> <td height="21" width="121" align="center" bgcolor="#808080" valign="top"> <p align="center"><b><font color="#0000FF" face="Arial" size="2">Last Name</font></b></td> <td height="21" width="85" align="center" bgcolor="#808080" valign="top"> <p align="center"> <b><font color="#0000FF" face="Arial" size="2">Seat Num</font></b></td> <td height="21" width="133" align="center" bgcolor="#808080" valign="top"> <b><font color="#0000FF" face="Arial" size="2">Seat Class</font></b></td> </tr> <tr> <td height="21" width="101" align="center" valign="top"> <font face="Arial"> <input type="text" name="firstname" size="20"></font></td> <td height="21" width="121" align="center" valign="top"> <font face="Arial"> <input type="text" name="middlename" size="20"></font></td> <td height="21" width="121" align="center" valign="top"> <font face="Arial"> <input type="text" name="lastname" size="20"></font></td> <td height="21" width="85" align="center" valign="top"> <font face="Arial"> <input type="text" name="seatnumber" size="8"></font></td> <td height="21" width="133" align="center" valign="top"> <font face="Arial"> <select size="1" name="seatclass"> <option value="none"> -Select Class- </option>	
<%if int(ava_first)>0 then%>	
<option value="first class">First Class</option>	P13
<%end if%>	
<%if int(ava_bus)>0 then%>	
<option value="business class">Business Class</option>	P14
<%end if%>	
<%if int(ava_econ)>0 then%>	
<option value="economy class">Economy Class</option>	P15
<%end if%>	
</select></font></td></tr></table></center><p align="center">	P16

<%if int(ava_first)=0 and int(ava_bus)=0 and int(avaecon)=0 then%>	
<font face="Arial" size="2" color="#FF0000">No Seats Available You cannot Reserve</font><font face="Arial" size="2" color="#FF6600">	P17
<%else%>	
</font> <input type="submit" value="Make Reservation" name="submit">	P18
<%end if%>	
<input type="button" value="Done" name="B1" onclick="javascript:location.replace('searchflights.asp')"></p> </form> <p align="center"><b><font face="Arial"> Bookings for Flight number	P19
<%=request.querystring("id")%>	P20
</font> </b></p> <center> <table border="0" width="683" cellspacing="3" cellpadding="2" id="table1"> <tr> <td height="21" width="101" align="center" bgcolor="#FF9966" valign="top"> <p align="center"><b><font color="#FFFFFF" face="Arial" size="2">First Name</font></b></td> <td height="21" width="121" align="center" bgcolor="#FF9966" valign="top"> <p align="center"><b><font color="#FFFFFF" face="Arial" size="2">Last Name</font></b></td> <td height="21" width="142" align="center" bgcolor="#FF9966" valign="top"> <p align="center"><b><font face="Arial" color="#FFFFFF" size="2">Middle Name</font></b></td> <td height="21" width="128" align="center" bgcolor="#FF9966" valign="top"> <p align="center"> <b><font color="#FFFFFF" face="Arial" size="2">Seat Number</font></b></td> <td height="21" width="110" align="center" bgcolor="#FF9966" valign="top"> <b><font color="#FFFFFF" face="Arial" size="2">Seat Class</font></b></td> <td height="21" width="64" align="center" bgcolor="#FF9966" valign="top"> <b><font face="Arial" color="#FFFFFF" size="2">Action</font></b></td> </tr>	P21
<% sql_bookings="select * from reservations where flightnumber=" & request.querystring("id") set rs_bookings= conn.execute(sql_bookings) do while not rs_bookings.eof%>	
<tr><td height="23" width="101" align="center" bgcolor="#808080"><font color="#FFFFFF" size="2" face="Arial">	P22

<%=rs_bookings("firstname")%>	P23
</font></td><td height="23" width="121" align="center" bgcolor="#808080"> <font color="#FFFFFF" size="2" face="Arial">	P24
<%=rs_bookings("lastname")%>	P25
</font></td> <td height="23" width="142" align="center" bgcolor="#808080"> <font color="#FFFFFF" size="2" face="Arial">	P26
<%=rs_bookings("middlename")%>	P27
</font></td><td height="23" width="128" align="center" bgcolor="#808080"> <font color="#FFFFFF" size="2" face="Arial">	P28
<%=rs_bookings("seatnumber")%>	P29
</font></td> <td height="23" width="110" align="center" bgcolor="#808080"> <font color="#FFFFFF" size="2" face="Arial">	P30
<%=rs_bookings("seatclass")%>	P31
</font></td> <td height="23" width="64" align="center" bgcolor="#808080"> <font face="Arial" size="2"> <a href="reserve.asp?delete=true&reservid=	P32
<%=rs_bookings("id")%>	P33
&id=	P34
<%=request.querystring("id")%>">	P35
Remove</a></font></td></tr>	P36
<%rs_bookings.movenext loop %>	
</table></center> </body> </html>	P37

Table 7.4.4 - Case Study - Atomic sections of reserve.asp

The composition rule C4 of the page "reserve.asp" is as follows:

C4=S2|->

p1.p2.p3.p4.p5.p6.p7.p8.p9.p10.p11.p12.(p13|e).(p14|e).(p15|e).p16.(p17|p18).p19.p20.p21 .(p22.p23.p24.p25.p26.p27.p28.p29.p30.p31.p32.p33.p34.p35.p36)*.p37

The graph of this page looks as follows (Fig. 7.4.4)

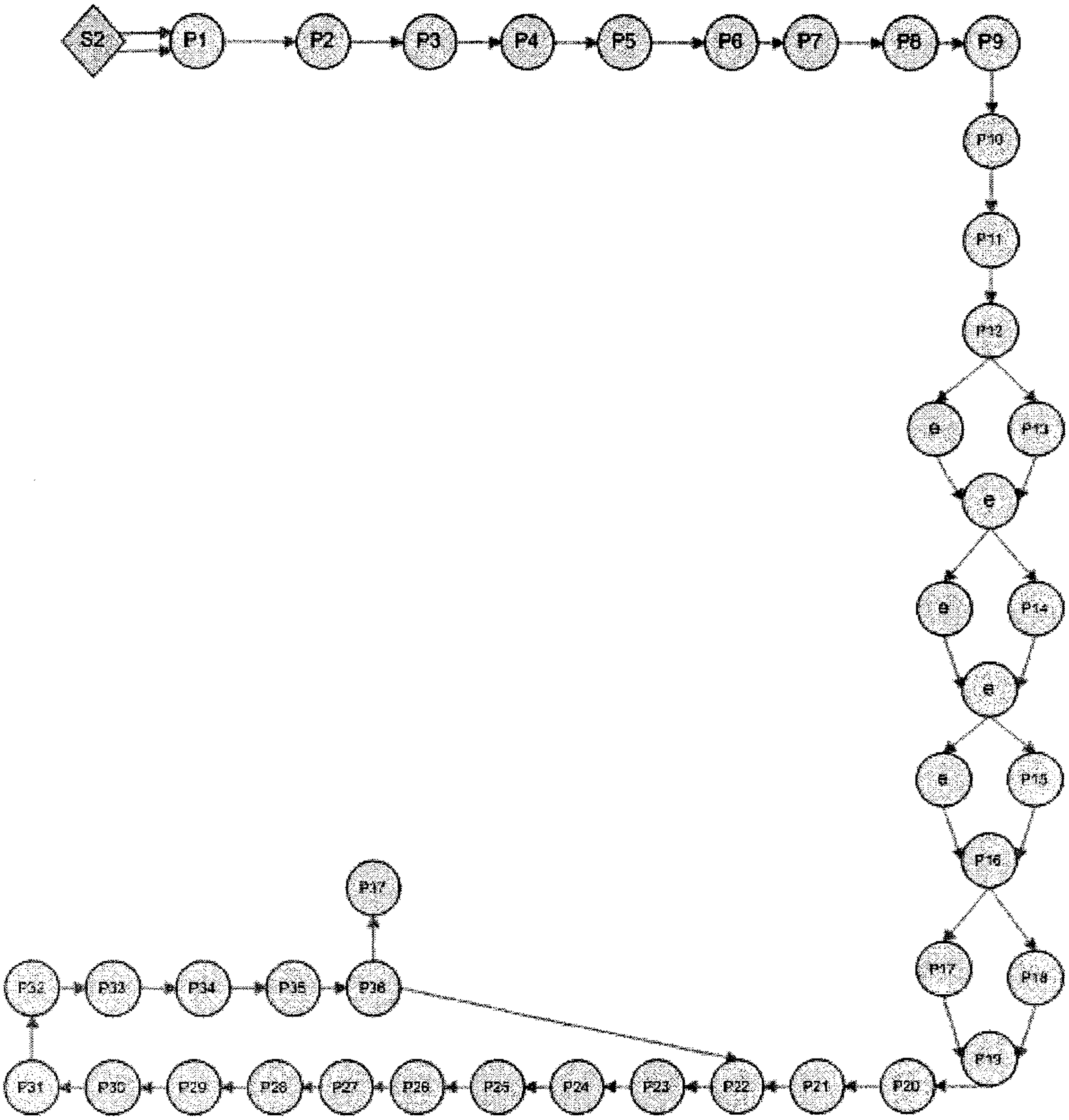


Fig. 7.4.4 - Case Study – Graph for server component reserve.asp

7.4.2 Testing Techniques

After identifying the atomic sections, deriving the composition rules, and drawing the graphs for each of the dynamic pages in the application, I will be testing the web components by creating test cases or test paths based on the prime criterion as proposed in this thesis earlier. Once the test paths are identified, they have to be applied to each of the corresponding components to verify correct HTML output. Applying those test cases to an individual component is an easy task but it might require establishing special environment and input conditions for the running application and thus they cannot be simulated by describing it on paper but they should be applied directly on the real application. For this reason, in this section I will generate all the needed test cases or test paths to test the components but I will not be simulating how those will be applied to the system.

7.4.2.1 Login.asp

The composition rule of this page is "C1=P1.P2.P3 |=> S2 | e" and since this graph has no cycles, then we can have two prime paths that gives us full coverage without the need to find any touring paths with sidetrips.. The two paths are:
[P1, P2, P3, S]} And [P1, P2, P3, e]

7.4.2.2 Searchflights.asp

The composition rule of this page is
"C2=S1|->p1.(p2.(p3|p4).p5.p6.p7.p8.p9.10.p11.p12.p13.p14.p15.p16.p17.p18.
p19.p20.p21.(p22.p23.p24)|p25).p26)*.p27"

To make deriving the touring paths easier I will abbreviate the sequential atomic sections into composite sections and thus I will be representing each of those sequences as one entity. Representing those sequences in this way will not lead to a different result since they are sequential in execution but this will make generating the paths easier. We will be using the following composite sections:

cs1= p6.p7.p8.p9.10.p11.p12.p13.p14.p15.p16.p17.p18.p19.p20.p21
cs2= p22.p23.p24

So our composition rule C2= S1|-> p1.(p2.(p3|p4).cs1.(cs2)|p25).p26)*.p27
And its graph looks as follows (Fig. 7.4.5)

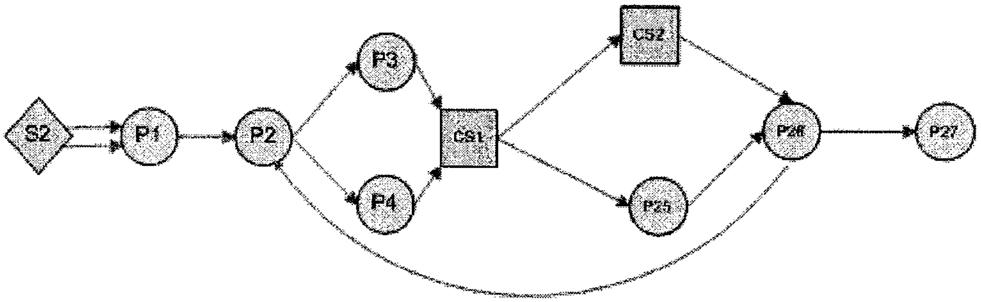


Fig.7.4.5 - Case Study – Reduced Graph of searchflights.asp

Based on the new composition rule, we will derive the prime paths (with no cycles) between the start atomic section P1 and the last atomic section P26 and those are:

[p1, p2, p3, cs1, cs2, p26, p27]
[p1, p2, p3, cs1, p25, p26, p27]
[p1, p2, p4, cs1, cs2, p26, p27]
[p1, p2, p4, cs1, p25, p26, p27]

Next we have to add to the above prime paths sidetrips that allow cycles to be traversed. We have one cycle between p2 and p26. This cycle should be added to the four prime paths we have above since all of them pass in p2 or p26. This cycle has four possible paths but it is not necessary to apply each of the four paths to each of the above prime paths. We can generate as many touring paths as we want but the minimum is to generate at least one touring path with side strip for each cycle for each prime path and we can select *any* path of this cycle to be added as side strip to *any* of the prime paths above. So the paths that tour the above prime paths with sidetrips will be:

[p1, p2, p3, cs1, cs2, p26, p2, p3, cs1, cs2, p26, p27]
[p1, p2, p3, cs1, p25, p26, p2, p4, cs1, cs2, p26, p27]
[p1, p2, p4, cs1, cs2, p26, p2, p3, cs1, cs2, p26, p27]
[p1, p2, p4, cs1, p25, p26, p2, p3, cs1, p25, p26, p27]

Now that we have the paths (or test cases) to test this component, we just have to find the required conditions in the live environment (database values, user input …) to test each of those cases on our page and this will not be simulated in this thesis (as explained earlier).

7.4.2.3 Viewreservations.asp

The composition rule of this page is as follows:
C3= S2|->p1.p2.p3.p4.(p5.p6.p7.p8.p9.p10.p11.p12.p13.p14.p15.p16.p17.p18.p19)*.p20

Again we combine the sequential atomic sections, each into one composite section. We can derive two composite sections as follows:

Cs1= p1.p2.p3.p4
Cs2= .p6.p7.p8.p9.p10.p11.p12.p13.p14.p15.p16.p17.p18.p19

So our composition rule will look like this S2|->cs1.(cs2)*.p20 and its graph looks as follows (Fig. 7.4.6)

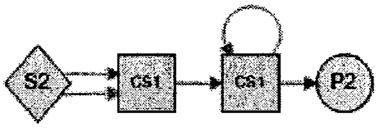


Fig. 7.4.6 - Case Study – Reduced Graph of viewreservations.asp

The only prime path starting at the first node p1 and ending in the last node p2 is
[cs1, cs2, p2]

The only cycle in this graph is from cs2 to itself, so the path that tours our prime path with side strip is [cs1, cs2, cs2, p2] which is our test path or test case for this component.

7.4.2.4 Reserve.asp

The composition rule for this component is:
C4=S2|->
p1.p2.p3.p4.p5.p6.p7.p8.p9.p10.p11.p12.(p13|e).(p14|e).(p15|e).p16.(p17|p18).p19.p20.p21
.(p22.p23.p24.p25.p26.p27.p28.p29.p30.p31.p32.p33.p34.p35.p36)*.p37

Again I am going to combine sequential atomic sections in composite sections as follows:

Cs1= p1.p2.p3.p4.p5.p6.p7.p8.p9.p10.p11.p12
Cs2= p19.p20.p21
Cs3= p22.p23.p24.p25.p26.p27.p28.p29.p30.p31.p32.p33.p34.p35.p36

So the new composition rule will be as follows:
C4= S2|->cs1. (p13|e).(p14|e).(p15|e).p16.(p17|p18).cs2.(cs3)*.p37

And the graph looks as follows (Fig 7.4.7.)

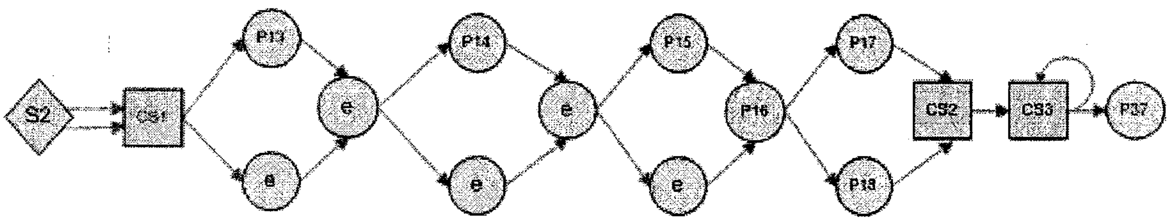


Fig. 7.4.7 - Case Study – Reduced Graph of reserve.asp

Since the graph of this component is big and has many conditional branches, listing all the prime paths between the first and the last node and later adding touring sidetrips to them will not be easy. Thus we follow the technique proposed in the model earlier, by selecting prime paths that satisfy the all-node coverage. In other words, we select a set of prime paths that covers all the nodes in the graph. I choose to use the following two paths to cover all the nodes in the graph:

[cs1, p13, p14, p15, p16, p17, cs2, cs3, p37]
[cs1, p13, p14, p16, p18, cs2, cs3, p37]

The only cycle that we have in this graph is from cs3 to itself and repeats the composite section cs3 many times. Simply we can add cs3 as a side strip to our prime paths listed above in order to have test paths with cycle coverage. Thus our test cases for this component will be:

[cs1, p13, p14, p15, p16, p17, cs2, cs3, cs3, p37]
[cs1, p13, p14, p15, p16, p17, cs2, cs3, cs3, p37]

7.5 Conclusion

By applying my modeling techniques to the sample web application on all three levels: the operational environment, the client side pages, and the server side programs, and by applying to those models the testing methodology discussed earlier, I can conclude this section by verifying that my proposed model proved to work on real applications and it provides satisfactory results in an efficient manner. I didn't go through the regression testing steps on my models simply because applying the regression testing techniques is very similar to the tests we applied with the differences in selecting the test cases themselves. The test case selection for regression testing, which is discussed in depth through out the thesis, are very straight forward and I believe that there is no need to illustrate them in this example.

Chapter 8

CONCLUSION AND FUTURE WORK

This thesis discussed a variety of ideas in the field of software modeling and testing and in particular web applications testing. The final conclusion of this thesis can be summarized by three main results.

First, the thesis presented a complete theoretical analysis model for modeling web applications and this was divided into three sub models; the architectural environment model, the model representing the web pages as seen by the user and the navigation between them, and the server side programs models that present the server programs which execute at run time and that produce dynamic HTML to the user. For each of those sub-models, the thesis presented a complete set of efficient testing and regression testing techniques that test the correctness and quality attributes of the web application based on the respective sub-model.

Second, the thesis presented a procedure for automatic black box regression testing of web applications. Along this procedure the thesis presented us with an efficient methodology for black box test cases selection based on user input. Moreover, the thesis described and in detail the structure, algorithm, and functionality of an automated tool that will be used with the procedure.

Third, the thesis presented a case study by applying the modeling and testing techniques on a sample web application. The case study showed that all the proposed ideas and techniques are feasible on real applications and they produce satisfactory results.

8.1 Limitations of the Research

Despite the major advantages and innovativeness of the research done, it still have some limitations mainly due to the heterogeneous nature of components involved in web architectures in addition to the continuous evolution and change of the web technology. Some of the limitations of this analysis model is that it applies to the classical web architectures such as ASP, JSP, and PHP while it might not cover completely the new architectures such as .NET (which is event driven) and new JAVA based web applications.

Another limitation is that the server side model tested the correctness of server side programs based on their HTML output regardless of the correctness of the server side logic. This type of modeling and testing is very similar to classical software testing and mainly that's why it was not covered in this research. But for the sake of completeness, the logic of server programs should be explicitly addressed by a separate model.

Yet another limitation with this model appears with web applications whose entire content is completely dynamic and retrieved at run time from external sources like databases, XML files, or content servers. This limitation needs to be researched by itself and a separate sub-model should be developed for it.

8.2 Future Work

Future research in this area should focus on eliminating some of the limitations mentioned in the previous section. New versions of the model should cover new architectures like .Net and it should contain a model for representing and testing server side logic and external content sources. Moreover, future versions should contain an implementation of the custom made tool in one of the known programming languages.

BIBLIOGRAPHY

[1] Ye Wu and Jeff Offutt. Modeling and Testing Web-based Applications. *GMU ISE Technical ISE-TR-02-08*, November 2002.

[2] Ye Wu, Jeff Offutt, Member, IEEE Computer Society, and Xiaochen Duz. Modeling and Testing of Dynamic Aspects of Web Applications. *GMU ISE Technical ISE-TR-04-01*, March 2004

[3] Filippo Ricca and Paolo Tonella. Analysis and Testing of Web Applications. In *Proc. of the 23rd International Conference on Software Engineering (ICSE'01)*, 2001

[4] Filippo Ricca and Paolo Tonella. Web Application Slicing. In *Proc. of International Conference on Software Maintenance (ICSM'2001)*, 2001

[5] Sebastian Elbaum, Srikanth Karre, and GreggRotherme. Improving Web Application Testing with User Session Data. In *Proc. of the 25th International Conference on Software Engineering (ICSE'03)*, 2003

[6] Harry M. Sneed. Testing a Web Application. In *Proc. of the Sixth IEEE International Workshop on Web Site Evolution (WSE'04)*, 2004

[7] Robert B. Wen. URL-Driven Automated Testing. In *Proc. of the Second Asia-Pacific Conference On Quality Software (APAQS'01)*, 2001

[8] Hui Xu, Jianhua Yan, Bo Huang, Liqun Li, and Zhen Tan. Regression Testing Techniques and Applications. *Technical paper from Concordia University, Canada, department of computer Science*, March 2003

[9] Ivan Granja and Mario Jino. Techniques for Regression Testing: Selecting Test Case Sets Taylored to Possibly Modified Functionalities. *The 3rd European Conference on Software Maintenance and Reengineering CSMR'99*, March 1999

[10] Hiroshi Suganuma, Kinya Nakamura, and Tsutomu Syomura. Test operation-driven approach on building regression testing environment. In *Proc. of the 25th Annual International Computer Software and Applications Conference (COMPSAC'01)*, 2001

[11] Lei Xu, Baowen Xu, Zhenqiang Chen, Jixiang Jiang,, and Huowang Chen. Regression Testing for Web Applications Based on Slicing. In *Proc. of the 27th Annual International Computer Software and Applications Conference (COMPSAC'03)*, 2003

[12] Simeon c. Ntafos. A Comparison of Some Structural Testing Strategies. *868 IEEE Transactions on Software Engineering, VOL 14, NO 6*, June 1988

[13] Hong Zhu. Software Unit Test Coverage and Adequacy. *ACM Computing Surveys, Vol. 29, No. 4*, December 1997

[14] Martina Marre′ and Antonia Bertolino. Using Spanning Sets for Coverage Testing. *IEEE Transactions on Software Engineering, VOL. 29, NO. 11*, November 2003