Journal of Theoretical and Applied Information Technology

www.jatit.org

MODELING AND AUTOMATED BLACKBOX REGRESSION TESTING OF WEB APPLICATIONS

Hamzeh Al Shaar and Ramzi Haraty
Department of Computer Science and Mathematics
Lebanese American University
Beirut, Lebanon
Email: h_alshaar@yahoo.com, rharaty@lau.edu.lb

ABSTRACT

Web applications are nowadays 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 emergence of web applications, techniques for maintaining their high quality attributes should be developed and exercised. Moreover, the quick evolution of web technology and the high user demands made web applications subject to rapid maintenance and change, which require the development of efficient regression testing techniques. The current testing efforts documented in research deal with a specific part of a 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 deal with analyzing the architectural environment of the web application. Motivated by the fact that there is no single model to represent the entire web application, and to model it from different perspectives at the same time, we propose a single analysis model with it testing techniques which models and tests 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. Having discovered, as well, that there is no automated black box regression testing technique, we also propose a methodology and algorithm to create a tool capable of applying black box regression testing automatically.

Keywords: *Modeling, Testing And Regression Testing, And Web Applications.*

1. INTRODUCTION

As the web is growing and invading our world, and as using the Internet became a normal habit to the new generation, 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.

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 mainly due to the nature of the web applications.

Unfortunately, there is no well-developed and mature model to analyze and test web applications yet. All the previous work dealt with certain aspects

of the web applications while neglecting the rest. Even worse, very few testing and regression testing techniques have been exploited to be used by web applications.

Motivated by the fact that there is no single model to represent the entire web application, and to model it from different perspectives at the same time, we 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, we also propose testing and regression testing techniques for each of the three parts of our model. The rest of the paper is organized as follows: section 2 presents related work. Section 3 presents our three poled model and the testing techniques of each pole. Section 4 presents our automated black box regression testing technique. Section 5 presents a case study. And Section 6 concludes the paper.

2. RELATED WORK

In this section, we present some of the previous work done in the field of web application modeling and testing and in the field of regression testing.

Wu and Offutt [14] presented an analysis model for modeling the server side components and the client server interactions. Wu et al furthered their work by extending their model to cover inter-component connections between different server side components [15]. The new model also covers the current state representation of a web application and the current state of variables. Ricca and Tonnela [1] modeled a web application as a sequence of web pages and they modeled the interaction between them in a UML diagram. Ricca and Tonnela also tackled web application slicing [2]. In their work, Ricc and Tonnela extended the concept of application slicing and applied it to web applications.

Sebatien, Karre, and Rotherm [12] presented a technique for automatically testing a web application based on the user session data collected from the user's navigation of the website. Similar work was proposed by Wen [11]. In his paper, he proposes a technique for generating test cases that are based on URLs to be automatically exercised. Sneed [4] wrote a paper that focuses on the web application architecture and recommends that this architectural environment be tested independently of the web application itself.

There are numerous papers that discussed regression testing of classical software [5][6][10][13]. These papers include work done by Xu, Yan, and Li [7], and the work done by Granja and Jino [8] and the work by Xu, Chen, and Yang [9], which also discusses regression testing of web applications using slicing.

3. THE MODEL

Our work models a web application from three different perspectives: The architectural environment, the client side navigation and flow of execution, and the server side programs and their dynamic output. Each of those parts has its own sub model and its own testing techniques.

3.1 The Architectural Environment Model

In this section, we propose a graphical analytical model to describe the architectural environment and also propose a set of testing and analysis techniques based on this model. Testing of this model is more oriented towards analysis and evaluation of the environment rather than checking for correctness, since it is 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,
- All software installed on physical servers (e.g., IIS, Apache, SQL Server, etc.),
- The communication protocols between connected nodes of servers,
- The type of messages exchanged, and
- The clustering and redundancy of servers.

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. We represent 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 (e.g., Intelx86, SPARC, etc.). The rectangular box includes one or multiple squares, with each square representing the software server installed on the machine such as IIS, Oracle Database Server or IBM Websphere.

The set of software servers are represented by a set SV of triplet where each triplet contains the software server name (named 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 two parallel dashed lines connecting the two boxes (not arrows). If the two software servers are load balanced, then the two boxes are connected with two parallel non-dashed lines (not arrows). Normally, if two servers are load balanced then they are automatically clustered for redundancy, so they are presented as load balanced servers only and not as both.

Clusters are 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 two 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 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.

The communication links between the software servers are presented as a set of quadruples Cmi where each quadruple contains 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 name any of the cluster members instead.

The type of exchanged messages is predefined and can be one 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), and
- 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.

3.1.1 The Architectural Model Testing Techniques

The tests for the operational environment are specified as a list of the quality attribute tests that should be done:

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

Tests 1, 2, 3, and 4 are critical and all tests should succeed in order to have a running application. Tests 5 through 8 are not critical, although important.

OS/HW Compatibility Test

The OS 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 has all server names paired with the value 1, then our test has succeeded.

OS/SW Compatibility Test

The OS and software servers compatibility test (OS/SW) is also a mandatory test, which our architectural environment must pass. 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 which is either zero or one, indicating if the installed application server (second part of the triplet from the set SV) is compatible with the operating system of the server installed on it. All resulting pairs should have the value of 1 paired with the software servers.

Communication Protocols Test

The communication protocol 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 Cmi and the value of the feasibility of this connection on the protocol level mentioned as the fourth 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 “one” else it will be paired with the value “zero”.

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. 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). This test succeeds if all triplets have value 1 as their third part.

Analysis of the Exchanged Messages

Analysis of the exchanged messages is a non-mandatory test, which evaluates the efficiency of the communication and the message exchange between the servers. In this test, we take each communication channel and we evaluate its efficiency taking into

consideration the kind of messaging, 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). This test can be represented as a set T5 of pairs where each pair has the connection name (Ci) and the value of the evaluation.

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 the 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: we first prepare the set SV' which is the set of all non-clustered and non-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 as the one minus the ratio of the cardinality of SV' over the cardinality of SV. Generally speaking, an ideal result equals one.

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 but it is derived from SV in a slightly different way. It eliminates from SV only those servers that exist in any pair in the set LB. We calculate the level of load-balancing as 1 minus the cardinality of SV" over the cardinality of SV. The optimal value is one.

Level of Scalability

The level of scalability test measures the level of scalability in our architecture. This test is not mandatory to succeed for the 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. 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|. scl = 1 is optimal.

3.2 The Client Side Navigational Model

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 applications consists of a set of web pages residing at the web server and are navigated by loading them into the browser one after the other by 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, we have to model what this web user sees in the client browser. 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's point of view, s/he is reading HTML pages, and interacting with HTML controls, mainly links and forms.

As with all three parts of our model, we use graphs in order to build our analysis model. The model presents the web application in a graph similar to UML.

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

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

Static HTML pages: we 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. Dynamic pages are modeled as rhombuses tagged by a pair containing the page name and the server name they reside on.

Pages with frames: our aim is to model the application as seen by the web user, and we model frames as they appear in the client browser, so we model 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 write the page name that is originally loaded into that frame. Frame behavior is 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.

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

- A *page* with a server redirection is 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.

- *Links:* Web pages are connected to each other by hyperlinks or simply links. A link is modeled by drawing a single headed arrow from the page containing the clickable link to the page, which the link refers to. The arrow is tagged by the list of parameters passed by the link.

- *Forms:* Forms are the most important components that provide the web user with an interface to submit data into the web page. We model a form on a page by a small circle tagged with the form name. The form submission is modeled by a double headed arrow, which is tagged by a list of the parameters that are passed.

3.2.1 Client Side Model Testing Techniques

In this section we suggest a set of testing techniques based on the suggested model and which enables the testing of 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 does not deal with the logic running at the server and it 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 and a proper flow of navigation between pages.

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 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 the application.

Orphan Pages

Orphaned pages are regular web pages in a web application that cannot be reached from any other page. Having an orphaned page is not desirable because it causes problems with graph coverage criterion. Orphan pages should be either:

1. removed from the application, or
2. analyzed and proper edges are adjusted accordingly.

To determine the orphaned pages we do the following analysis on the 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 the 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. If we cannot find such an edge, then n is added to N'. N' will contain all orphan nodes that should be dealt with accordingly.

Broken Links

Sometimes we may have some links referring to a non-existent page. When we generate test cases based on our graph model and we try to exercise them on the application, we can detect those broken links and fix them.

To generate test cases based on the all-node coverage criterion, we create a set N of all the available nodes. We start 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). We add the path that we just traversed to the set T of paths and we start traversing a different path until N = N'. Now the set T contains the set of paths that traverses all the nodes in the graph (all-nodes coverage criteria). We apply all test cases in T to the application to detect (and later fix) any broken links.

Dead 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 be forced to use the browser's back button. 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 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. Any page not satisfying this condition is added to the set D. After we finish investigating all nodes, we start analyzing the nodes obtained in D. Each node in D can be dealt with based on one of two scenarios:

1- N navigational hyperlinks are added from this page to the end if no additional navigation is required from this page.
2- Corresponding edges to the graph are added in case navigation is needed from this page to some other page in the application.

Parent-Child Sequences

Parent pages are those pages that must be traversed before other pages. We should differentiate between direct parents where a parent page and a child should have an edge between them and between grand parents where a parent page and a child should have a path connecting them.

To start the testing technique, we designate the set of direct parent pages by P1 where all elements of P1 are pairs where 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 the execution sequence. E as defined earlier is the set of edges. T is the set of paths that satisfies the all-node coverage criterion. For the sake of completeness, we define F1 as the set of non-satisfied direct parent child requirement. We define F2 to hold the indirect parent child pairs that failed the test. F1 and F2 are initially empty.

We start by testing the direct parents. For each pair in P1, search for a corresponding pair in E where the first and second items from P1 match the second and third items in E, correspondingly. Similarly, we make sure that the child page is not reached from pages other than the parent so we 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).

What is verified for indirect parent-child sequences is: for each parent child requirement, all paths traversing the child should have traversed the parent earlier. If the paths satisfying the all-node criterion are not enough, select the test paths satisfying the level-k coverage criterion. The level-k coverage criterion on a set of nodes L, is 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. To simplify things, we limit set L to child nodes:

- having more than one arrow arriving directly to them in the graph, and
- whose path from the start node has a loop.

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 the testing effect to be.

3.3 The Server Side Programs Model

In modeling the client side components, we considered the web application as a set of static and dynamic HTML pages connected by hyperlinks. We did not analyze the different HTML sections and the way they were generated, but rather 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 look into the server side code that is simply the engine that generates the HTML output. We decided to adopt Wu and Offut's model which models web applications with emphasis on the server side programs [15]. While Wu and Offut modeled the entire web application in one graph, we adopted his technique to represent the internal structure of an individual web component. We follow the steps below to create our server side model:

1- Atomic sections (AS): identify the atomic sections,
2- Composite sections (CS): derive the composite sections from the AS,
3- Transitions: identify transitions and interactions between different CS and AS,
4- Transition rules: identify transitions rules, and
5- Model the web component from ATS, CS, and transitions.

3.3.1 The Server Side Programs Model Testing Techniques

Testing techniques for this model are the same as those proposed by Wu and Offut [15]. Basically the component is tested by applying test cases which are paths in the web component graph (WCG). The prime criterion (touring paths with sidetrips) is used to cover the graph and select test cases. Detailed analysis of this coverage criterion and examples can be found in [14], [15], and [3].

4. AN AUTOMATED BLACK BOX REGRESSION TESTING TECHNIQUE

White box testing in software engineering deals with testing and validating the internal structure of a software component, is not enough to validate the correctness of any application and here comes the importance of black box testing techniques that take the whole application as one entity and generates test cases validating output versus input.

The Proposed Technique

The proposed regression testing technique is automated, which means that we run the test cases manually one time and later in regression testing, the test cases are automatically executed and validated. We highlight the algorithm for this technique and the basic structure of a custom tool to be used in this technique.

The basic steps of the 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, the developed tool saves HTML output for later comparison and validation in the regression testing stage.
4- In the regression testing step, the tool executes the sequence of saved URLs automatically and collects the output values specified in step 3. After executing the test cases, the tool compares the output values collected in this step with those collected in step 2 and it provides the user with the sections that produced different output. The tester will have to analyze those differences manually.

As noticed, the technique is divided into two parts: the first part consists of selecting the test cases and this is done manually by the tester (step 1). 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, we propose an efficient algorithm for test case selection, and then we explain the architecture of the tool and the algorithm of its functionality.

Test Case Selection

The test cases created for black box testing should traverse the application based on a set of input values. 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 combinations of all input on all pages may be extremely large or even impossible. In order to make test case generation efficient and manageable, we define input domains and input patterns instead of choosing specific input.

For numeric input we 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 identify the type of numeric values allowed in this field; for example, integer, decimal, even numbers, odd numbers, etc.

After defining the range, we 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 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, zero value, negative Value.

g. Boundary values for domains that are defined by a range.

For text input, we manually analyze the available possible input and create an input pattern for each set of input values yielding similar output. For selection input (dropdown menus, checkboxes, and radio buttons) we identify the different selection patterns that would result in similar output and group those in a single pattern. Now that we have a set of input patterns, we create a set of test cases satisfying all our input patterns.

The Tool

The proposed tool is used by the tester in the testing phase to exercise test cases. While navigating specific URL, HTML values (indicators) are saved. In the regression testing phase, the saved URLs are re-executed automatically and the tool saves and compares the same HTML indicators.

The key idea behind our approach for performing 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 analysis, we conclude that each time we call the same pages in the same sequence and passing the same parameters, we should receive the same output.

The reason why we decided to save and compare specific sections of HTML indicators and not the entire output is because most of the times certain small HTML values on the page are enough to give us information about the output of the page and thus, we can deduce if the page functions correctly or not. Moreover, saving the entire page is too bulky and non-efficient for later analysis. However, the tool makes it feasible for the tester to save specific entire pages prior to re-running the test cases.

The Tool's Architecture

The tool is divided into three parts: the browser, the capturing tool and the analyzer. The browser and the capture tool have to co-exist within the same screen of the application because we need to capture data as we navigate the test cases manually the first time. The analyzer, which re-executes tests and compares the collected output and validates is in a separate screen since its functionality is not used at navigation time.

The Embedded Browser

The embedded browser is used to perform the black box testing manually and to collect the needed information automatically for later use. The browser collects a list of all visited URLs, in addition to all HTTP parameters passed to this URL via HTTP POST and HTTP GET methods. The captured data is saved in flat files that will be read by the analyzer in the regression testing phase.

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. For each new page displayed in the browser, all possible HTML indicators whose values can be captured are displayed in a side screen. Each HTML component (tag) that can be named in HTML is a candidate for being an indicator and those are:

1- Form elements (text input, radio buttons group, checkboxes, drop down lists),
2- HTML tables, table Rows, table cells, and
3- and <layer> tags that can enclose text or other HTML sections.

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 HTML section should be enclosed within a div or layer tag and given a name. It is important to mention that adding a 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.

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 the tester with the difference in indicator values. Those values are analyzed by the tester. Of course the tool does not re-execute all the test cases but only the relevant ones and those are subjectively selected 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 using the URL

and the parameters associated with it as per the HTTP protocol standards.

2- Submit this http request to the server and receive the response.
3- Identify the list of indicators corresponding to this page and capture them.
4- Read the next URL in the URLs file and repeat the same process.
5- 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 any difference exists between the content of the files then they are copied to a sub folder for later manual inspection.

Sometimes, before starting the regression testing we may need to capture additional indicators that were not captured in the testing phase. Here, the tester can edit the saved file and add new HTML indicators (similarly for removing indicators).

After running the analyzer tool, we obtain a set of indicator files that are not matching. The role of the tester is to inspect and analyze the modified files to verify that the changes are as desired.

5. CASE STUDY

In order to further elaborate and support our presented ideas, we present a case study on an example web application and we apply to it our proposed model and the testing techniques.

Application Description

The web application 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. We avoided choosing a huge and complicated application since that would result in complicated models and long test cases while our aim is to materialize and apply our basic ideas in the clearest way possible. This web application operates as follows: the user starts by entering the 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 (see Figure 1).

By default, the page opens with all the available flights listed (see Figure 2). 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 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 the session variable and redirects the user to the login page.

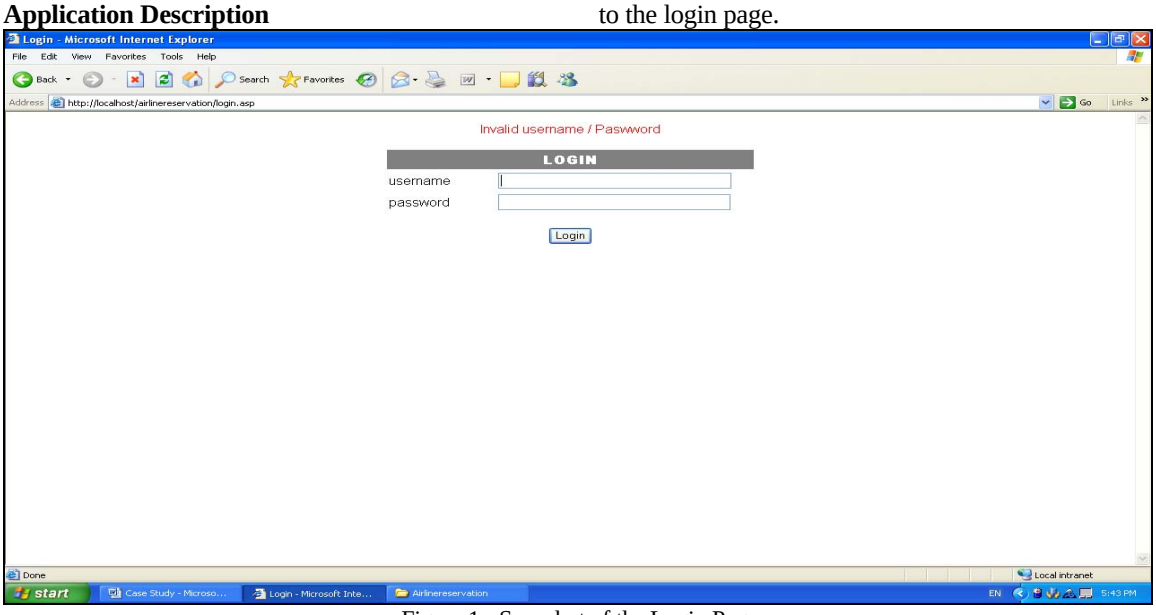

Figure 1 - Snapshot of the Login Page

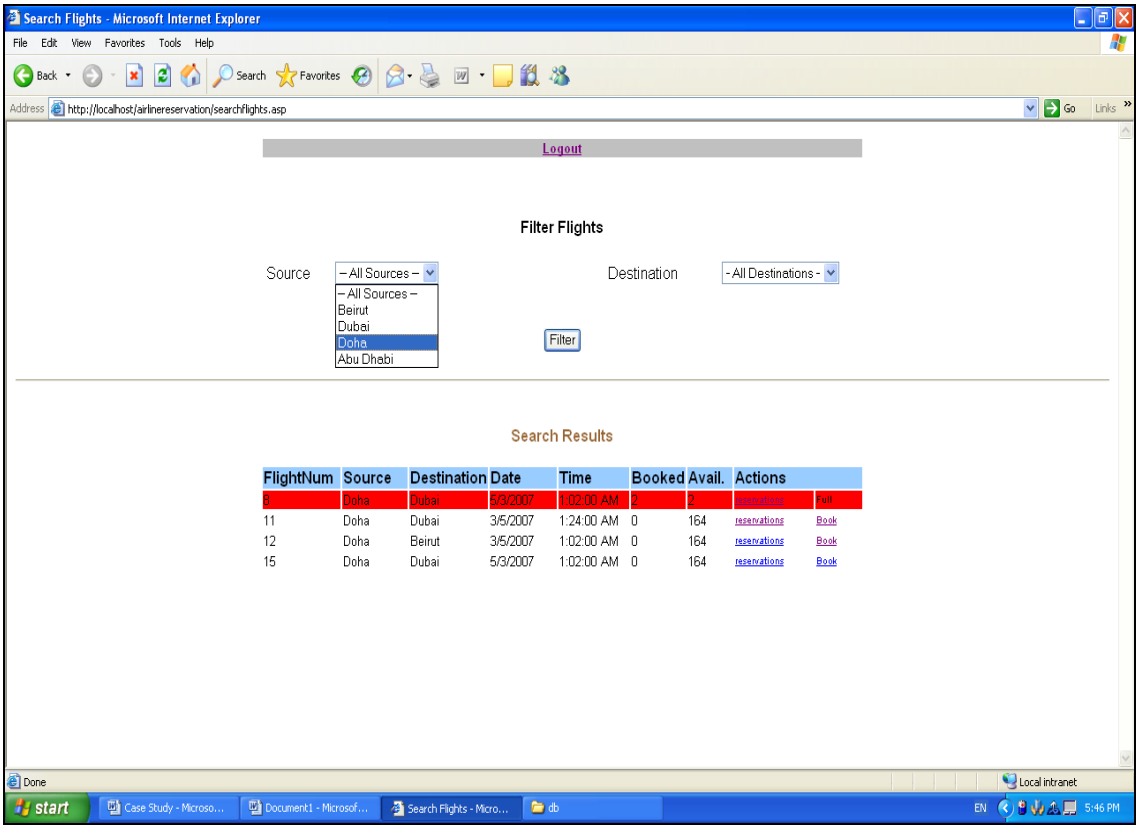

Figure 2 - Snapshot of the Search Flights Page

Clicking the booking link transfers the user to a page to create a reservation (see Figure 3). On that page, the user has to enter the passenger's information, in addition to 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 itself 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.

Clicking the reservations link transfers the user to a page summarizing the current bookings for this flight (see Figure 4). 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/her 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.

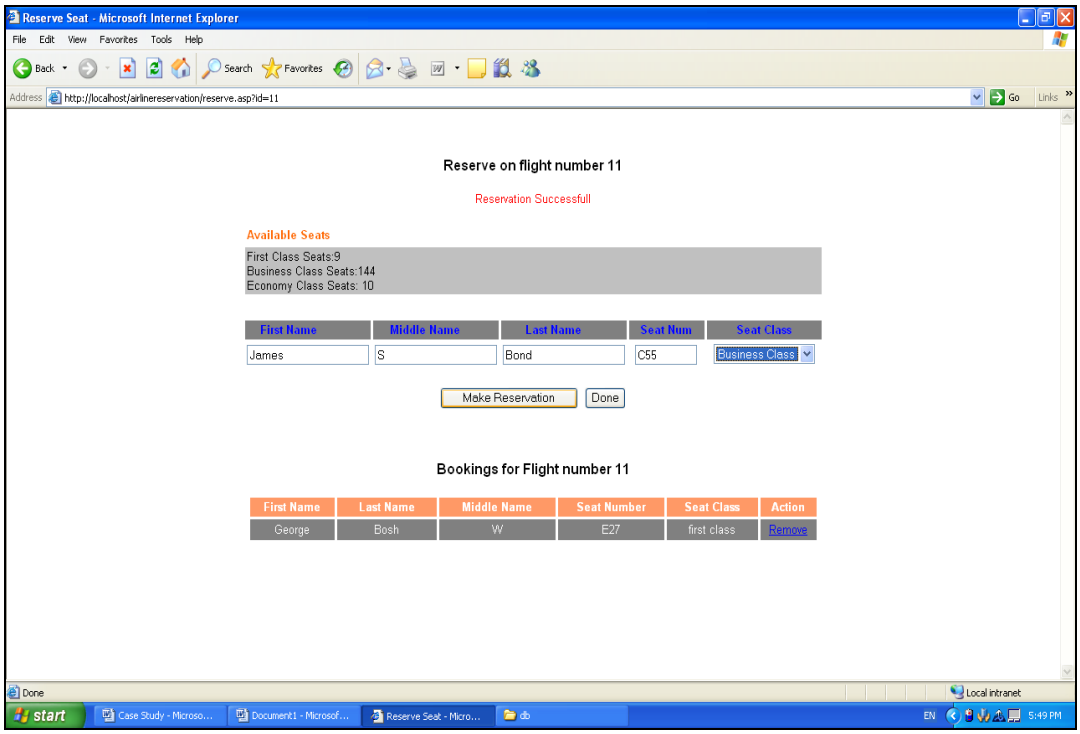

Figure 3 - Snapshot of the Reserve Page

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. We decided to run our 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 is an SQL Server running on a separate Windows machine.

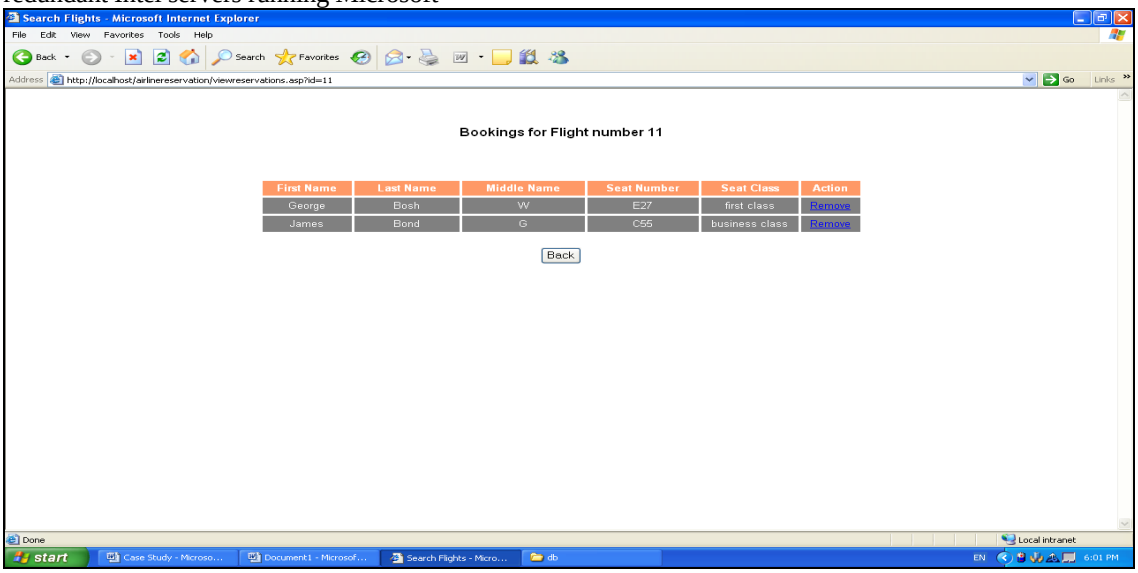

Figure 4 - Snapshot of the View Reservations Page

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

Figure 5 defines the architecture of the approach.

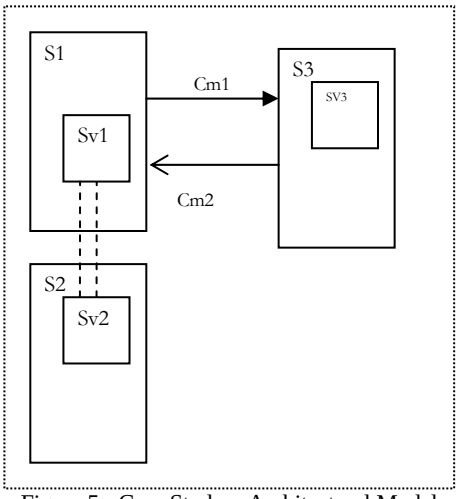

Figure 5 - Case Study – Architectural Model

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 tests are used to evaluate other quality attributes of the architecture. We apply each of the 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 we can conclude that the 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 the 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 the 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, the 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 the architecture.

- The eighth test checks the level of scalability of the architecture. We test salability based on the following formula, 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 0.5, but we should be aware that the database cannot be scaled which might create a bottle neck in the future.

The Client Side Model

This model considers all the pages included in the application as seen for the end user. We start by listing all the web pages in the application:

1. Login.asp: This page has a form allowing the user to enter the 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: The 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 two 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 identification 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 "Searchflights.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 page deletes the session variables of the logged in user and it redirects to the login page.

The graph of the client side model is defined in Figure 6: We have named the pages by p1, p2, p3, p4, and p5 for easier reference in the test cases.

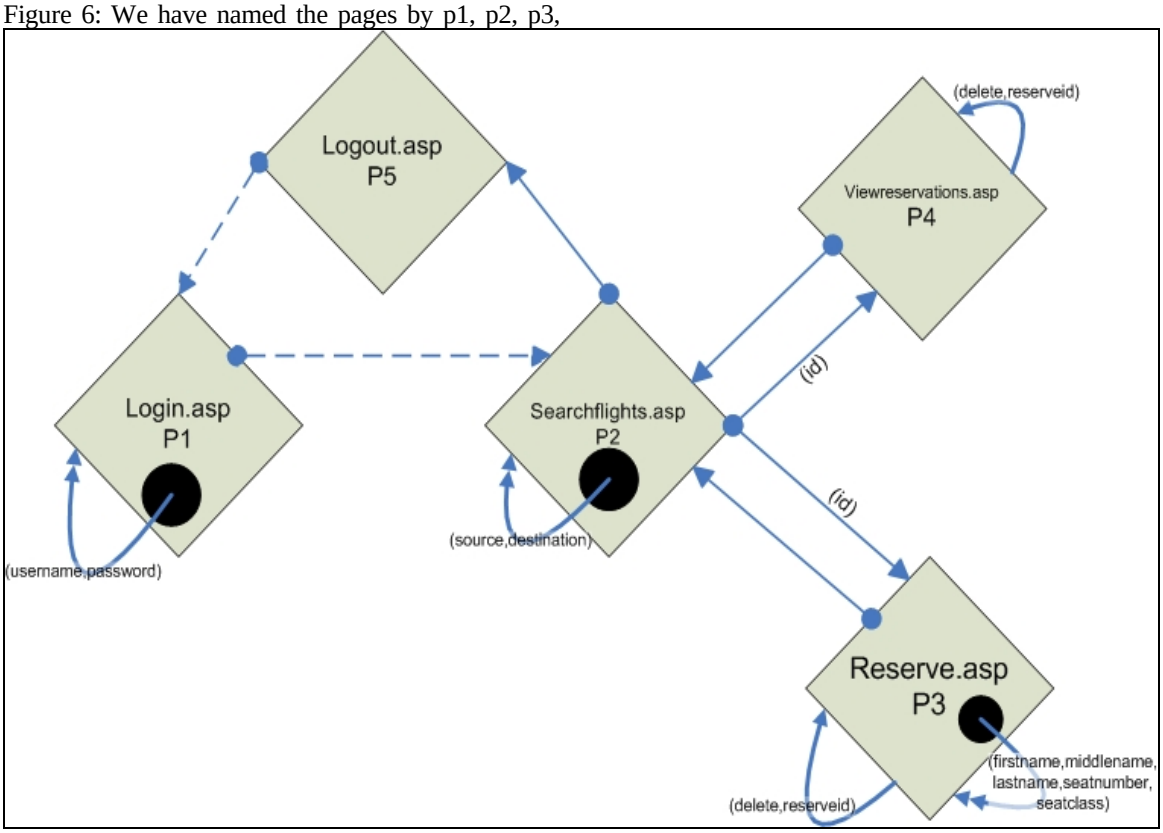

Figure 6 - Case Study – Client Side Model

Testing Techniques

After constructing our web application graph, we have to follow the test techniques discussed earlier in order to validate the correctness of the application and to ensure 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 the tests, we will define the sets N of nodes and E of edges (as discussed earlier):
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 searching and 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 odes 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 two 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, 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 identification number 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 ensure 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 ensure 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 the 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 with the test results.

The Server Side Model

In the server programs model, we considers all dynamic pages and server components that generate HTML dynamically. Thus, we 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. We present the composition rules, the graph, and the test paths of each of those components based on the algorithms proposed by Wu and Offut [14] [15]. The details of deriving the atomic and composite sections, drawing the graphs, and selecting test paths based on the prime coverage criterion are found in [3].

The page "Login.asp" has the following composition rule C1=P1.P2.P3 |=> S1 | e, and its WCG looks like the picture in Figure 7:

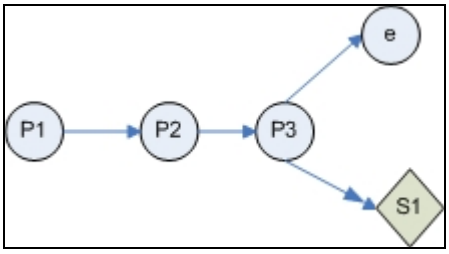

Figure 7 - Case Study – Graph for server component login.asp

The page "Searchflights.asp" has the following composition rule:
C2= S1|-> p1.(p2.(p3|p4).cs1.(cs2)|p25).p26)*.p27
Here we combined the sequential atomic sections ($P_i.P_{i+1}\ldots$) in composite sections CS_j. The graph is shown in Figure 8.

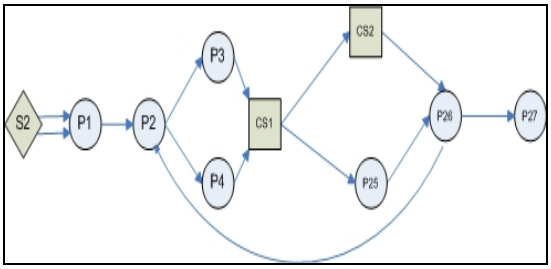

Figure 8 - Case Study – Reduced Graph of searchflights.asp

The page "viewreservations.asp" has the following composition rule (after combining sequential atomic sections into composite sections)
C3=S2|->cs1.(cs2)*.p20

The graph is shown in Figure 9.

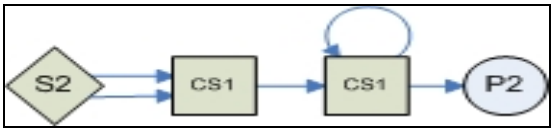

Figure 9 - Case Study – Reduced Graph of viewreservations.asp

The page "reserve.asp" has the composition rule (after combining sequential atomic sections into composite sections) as follows:
C4= S2|->cs1.(p13|e).(p14|e).(p15|e).p16.(p17|p18).cs2.(cs3)*.p37
The graph is shown in Figure 10.

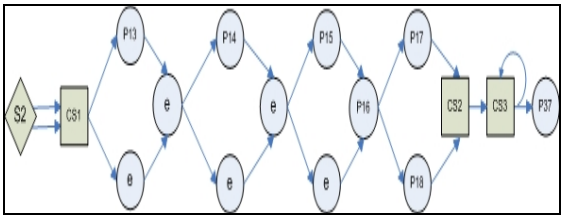

Figure 10 - Case Study – Reduced Graph of reserve.asp

Testing Techniques

After identifying the atomic sections, deriving the composition rules, and drawing the graphs for each of the dynamic pages in our application, we test the web components by creating test cases or test paths based on the prime criterion.

- The Page Login.asp has the prime paths:
  [P1, P2, P3, S]
  [P1, P2, P3, e]
  No need to find paths satisfying the prime coverage criterion (touring prime paths with side trips) because the graph does not have any cycles.

- The page Searchflights.asp has the prime paths:
  [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]

  The paths (test cases) satisfying the prime coverage criterion for this page (touring prime paths with side trips):
  [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]

- The page Viewreservations.asp has only prime path is:
  [cs1, cs2, p2]
  The path (test cases) satisfying the prime coverage criterion for this page (touring prime paths with side trips) is: [cs1, cs2, cs2, p2].

- The page Reserve.asp has a huge number of prime paths but we choose the following to that visits all nodes:
  [cs1, p13, p14, p15, p16, p17, cs2, cs3, p37]
  [cs1, p13, p14, p16, p18, cs2, cs3, p37]

  The paths (test cases) satisfying the prime coverage criterion for this page (touring prime paths with side trips):
  [cs1, p13, p14, p15, p16, p17, cs2, cs3, cs3, p37]
  [cs1, p13, p14, p15, p16, p17, cs2, cs3, cs3, p37]

6. CONCLUSION AND FUTURE WORK

In this paper, we presented a complete theoretical analysis model for modeling and testing web applications, which is divided into three sub models; the architectural environment model representing operational environment hosting the web application, the client side model representing the web pages as seen by the user and the navigation between them, and the server side programs models that presents the server programs, which execute at run time and that produce dynamic HTML to the user. Moreover, we presented a technique for automated black box regression testing. The theoretical analysis in this work was supported by a case study on a real web application.

Future work includes covering new architectures like .Net. Future work will focus on adding a model for representing and testing server side logic and external content sources.

REFERENCES

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

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

[3] Hamzeh K. Al Shaar. Modeling, Testing, and Regression Testing of Web Applications (2006). Thesis submitted for MS Computer Science, Lebanese American University.

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

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

[6] Hong Zhu. Software Unit Test Coverage and Adequacy (1997). ACM Computing Surveys, Vol. 29, No. 4.

[7] Hui Xu, Jianhua Yan, Bo Huang, Liqun Li, and Zhen Tan (2003). Regression Testing Techniques and Applications. Technical Paper from Concordia University, Canada, Department of Computer Science.

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

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

[10] Martina Marre´ and Antonia Bertolino (2003). Using Spanning Sets for Coverage Testing. IEEE Transactions on Software Engineering, VOL. 29, NO. 11.

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

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

[13] Simeon C. Ntafos (1988). A Comparison of Some Structural Testing Strategies. IEEE Transactions on Software Engineering, VOL 14, NO 6.

[14] Ye Wu and Jeff Offutt (2002). Modeling and Testing Web-based Applications. GMU ISE Technical ISE-TR-02-08.

[15] Ye Wu, Jeff Offutt, and Xiaochen Duz (2004). Modeling and Testing of Dynamic Aspects of Web Applications. GMU ISE Technical ISE-TR-04-01.