Queen's University

School of Computing

Depth Area: Software Testing

Depth Topic: Web Applications Testing

Supervisory committee: Dr. J.R. Cordy, Dr. T.R. Dean and Dr. S. Knight

Prepared by

Ben Kam

Student Number: 5285549

February 28, 2007

Abstract

A study from the Business Internet Group San Francisco (BIG-SF) [2, 3] found approximately 70% of websites contain bugs and suffer some kind of failure; not only commercial websites but also US government managed websites. These faults reside in both static pages and dynamic pages. In this survey paper, we review how the existing web application testing methods test static and dynamic pages. It is our objective to better understand why testers are unable to reveal these faults. As a result, we suggest a novel approach for web applications testing.

Table of Contents

List of Figures

List of Tables

Chapter One

1.0 Introduction

In 1990, the first web browser Nexus was born. At that time, no one considered web application testing, perhaps because of simplicity and very limited application features. A web application is an application that is accessed via a web browser using the Hypertext Transfer Protocol (HTTP) over a network such as the Internet or intranet. To be more precise, HTTP is a stateless request and response protocol between clients and servers. An HTTP client (the browser) makes an initial request to an HTTP server (the web server) via Transmission Control Protocol (TCP) establishing a connection on port 80 by default. An HTTP server listens to this port and sends the request message (application) to the HTTP client.

Nowadays, web applications have become more sophisticated and complicated than ever before. Many legacy software systems have also been migrated to web-based systems. Web applications tend to change more rapidly with shorter development times (time-to-market concerns) than traditional software. The short lifecycle of web application software is normally three to six months [24].

According to a study from the Business Internet Group San Francisco (BIG-SF), of forty-one web sites under US government management, twenty-eight (68.29%) contained bugs that caused web application failures [2]. These faults reside in the web documents (static pages and dynamic pages). The failures are the file not found error and incorrect data responses. Have these websites been tested before being put on the web server? As government managed websites, time-to-market constraint should be less than for commercial business websites. Does this mean that the failure rate of commercial websites are better or worse? The answer to this question was revealed on November 22, 2002, the biggest shopping day of the year – the day after Thanksgiving. The Black Friday report [3] stated consumers spent $196 million online shopping. This report also stated that 72.5% of the websites suffered some kind of failure. If it were not for such a high percentage of website failures, would consumers have spent more than $196 million on this day?

A web application is nothing more than a file with a program (web page) written in programming languages such as HTML, PHP, JSP, VBScript, etc. Why then do websites have such

high percentages of failures? If the web application is tested before delivery to market, then why are there still so many errors? In this survey paper, we present and analyze several web application testing methods and explain why traditional software does not have the high failure rate that web applications incur. The ultimate goal is to find a robust testing technique to improve the performance of web application testing.

In next chapter, we define the scope and motivation of this paper. The software testing overview, taxonomic web applications, and potential problems originating from static and dynamic pages will be addressed in Chapter 3. Chapter 4 investigates six groups of web application testing methodologies. Individual testing method will be discussed and compared. Chapter 5 contains an overall evaluation of these six group of testing methods. Chapter 6 will answer questions posed in Chapter 2. Finally, the conclusion will be drawn in the last chapter.

Chapter Two

2.0 Scope and Motivation

The problem of web application failure is twofold. First, there are a significant high percentage of failures [2, 3] reflecting web application implementation. Second, why were software testers unable to reveal these faults? This brings forth the question, how do test engineers conduct web application testing? How effective are the testing methods?

The operation of a simple web application [9] is illustrated in Figure 1. The discussion of this survey paper will focus on the circle (as these faults originate from the static and dynamic pages) and how the existing web application testing methods test the correctness of static and dynamic web pages. The existing testing methods will be categorized, and we will discuss, evaluate, and summarize each taxonomic group's approach. As a result, we can differentiate the strengths and weaknesses of the majority of existing web application testing techniques.

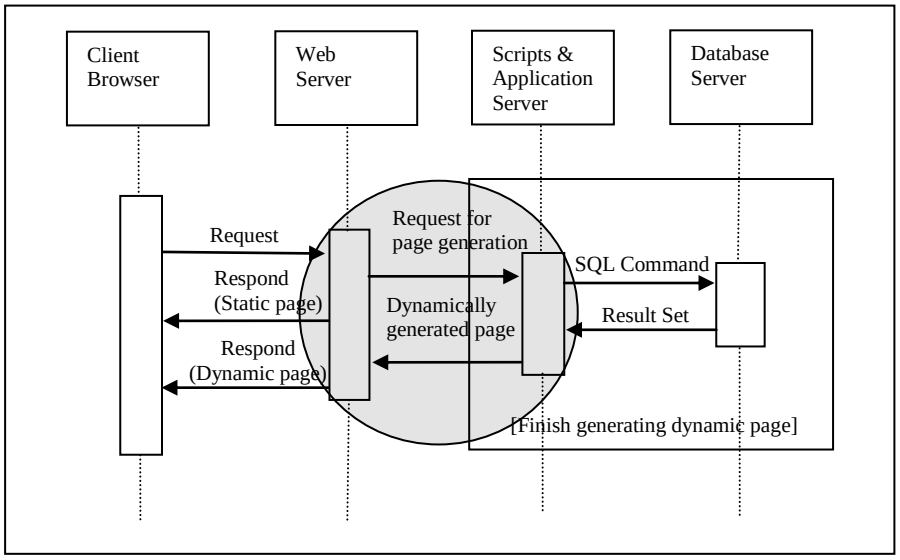

Figure 1: A simple web application operation [9]

In order to help understand the web application testing problems, we also aim to answer the following questions;

1. What are the similarities between traditional software testing and web application testing approaches?

2. What are the differences between traditional software testing and web application testing approaches?

Then based on the answers, we will address the open question;

3. Can we apply traditional software testing methods to perform web application testing?

Consequently, we will suggest a complete and robust web application testing methodology.

After a survey of existing web application testing methods, we have categorized these testing methods as shown in Table 1. (Table 1 shows the categorization of existing web application testing methods.) There are a total number of six groups. Each group contains the testing models, testing methods, testing criteria, and testing targets. Using notations and symbols to model software systems is common practice for traditional software development. This can also be applicable to web applications. Some models can be used for requirements validation and the others can be used for specifications verification (model-based testing techniques). For example, the formal method group contains two testing models, TestUML and State Chart. The models represent the web application structures and use for test case generation. The testing methods describe the testing mechanism; white box or black box approach. The testing criteria show the coverage requirement (data and/or control flow). The testing targets describe the functional or non-functional requirements of the software to be tested.

For the sake of consistency, four confusing words in software testing will be defined here for their use throughout this paper.

Fault – an actual fragment of code in the program that causes failure
Failure – a function of the program is performed incorrectly
Bug – an informal name for failure and fault
Error – a mistake made by a programmer

	Model	Testing Methods		Testing Criteria		Testing Targets		References
		White Box	Black Box	Control flow	Data flow	Functional	Non-functional	
Formal Method	TestUml	x		x		x		[4, 5]
	Stated-base		x	x		x		[22, 23]
Object-oriented	Agent-Based	x	X		x	x		[26]
	Object-Based	x			x	x		[20, 21]
	OO Web Test	x			x	x		[17]
Statistical	Markov Chain	x	X				x	[6, 7, 14, 15, 43]
UML	ReWeb/TestWeb	x		x		x		[29, 30, 31,34,36,37]
Slicing	WebApp slicing	x			x	x		[32, 35]
User Session	User Session Data	x	X	x		x		[9, 10, 38, 39, 40]

Table 1: Categorization of existing web application testing methods

Chapter Three

3.0 Software testing overview

Software testing plays important role in traditional software development. Software testing must be conducted thoroughly before we can deploy or deliver it to clients. The software testing and web application testing process should consist of at least two activities: validation and verification [13]. Validation ensures that the software has a correct implementation (in agreement with the user requirements). Verification checks that the software has been implemented correctly (in accordance with its specifications). In other words, we want to make sure the implemented software is what the client wants and is flawless (if possible) before delivery.

The process of software verification can be achieved by formal or informal testing. High security and human life related software should be tested using formal methods. However, formal testing requires deep professional knowledge to model and to prove the system's correctness. For example, the denotational semantics method uses mathematical objects to represent functions and prove its correctness. This approach is not commonly used for industrial software development; it is especially not suitable for the ever changing, time-to-market constrained web application software.

Informal testing is widely accepted by industry. One of the reasons is that it is relatively easier to master these informal testing techniques compared to formal testing methods. However, a drawback of informal testing techniques is often the tedious and confusing software testing jargon. Different testing teams may test a program with the same approach but call the testing method different names. For example, "complete testing" and "exhaustive testing" are understood to have the same meaning. We can easily find more than a hundred technical testing names in the glossary (appendix A). International Software Testing Qualification Board [47] has sought the views of different parties (industries, organizations, and government agencies) and attempted to standardize the glossary of terms used in software testing. Consequently, most terms in the glossary are widely accepted.

Test engineers commonly use black box and/or white box approaches to conduct testing. It is because most testing originates from these two fundamental methods. In order to achieve black box testing, there are several major methods that can be used, such as equivalence partitioning on the

input and output domains, and analysing the boundaries and extreme values. As a result, three testing methods (equivalence partitioning testing, boundary value testing, and extreme value testing) originate from black box testing. As for white box testing, the basic approach is to find test cases that can execute the code differently, in other words, looking at different criteria of code coverage. Therefore, white box testing yields condition testing, decision testing, path testing, statement testing, and so on. In short, black box and white box testing are only the starting point to determine whether the source code is needed for testing. We have suggested an alternative view of Table 1 which is shown in Table 2. (Table 2 suggests an alternative view of Table 1.)

	Model								
	Formal		**Object-Oriented**			**Statistical**	**UML**	**Slicing**	**User Session**
Testing Methods	Test Uml	State Based	Agent Based	Object Based	OO Web Test	Markov Chain	ReWeb TestWeb	WebApp Slicing	User Session Data
Data Flow testing			√	√	√				
Functional testing		√							√
Maintainability testing						√			
Path testing	√						√	√	
Performance testing						√			
Robustness testing	√		√	√	√		√	√	√
Reliability testing						√			
Regression testing									√
State transition testing		√							
Statistical testing						√			
Syntax testing		√							
Structural testing	√		√	√	√		√	√	

Table 2: Suggested alternative view of Table 1

Both testing approaches have advantages and disadvantages concerning performance in terms of effectiveness and efficiency. Thus, many researchers focus on finding a way to maximize the thoroughness of testing while minimizing the processing time as an ultimate research goal. This goal also applies to web application testing.

3.1 Potential Web Application Problems

Browsing websites can be considered daily routine activities. Based on the popularity of websites, the web contents are ever changing and have short time-to-market constraints; thus fast-paced web application development is a natural necessity. Therefore, an efficient and effective testing method should be in demand. Testing plays important role for software development. However, web applications testing falls by the wayside because developers consider the web application testing time consuming and lack significant payoff [12].

Web applications can be very complex. It is more than just navigating the websites by clicking the link provided by the web page and the web server will deliver the corresponding the page to the user's computer screen. Indeed, web application interprets the users input data and based on session cookies, server state and database state, generates a dynamic page to the user. If we know what potential problems could occur from the static and dynamic pages, then we can better strategically perform web application testing. In this sub-section, we list as many potential problems of web applications as possible arising from the static and dynamic pages.

The first problem is the static link problem. Sometimes a web page provides some links which link to other static pages. When users click these provided links, the message "404 file not found" is displayed on their screen. This kind of failure is caused by the broken link is a common fault, which often occurs when navigating websites. The main reason for this problem is the corresponding page or target objects cannot be found at the target location, such as missing pictures, clips, and/or files.

The second problem is the dynamic link problem. When a dynamic link (or button) is used, this link triggers software possibly written in JavaScript or VB Script to perform the task. This kind of action does not require the user to input any data, unlike a form link. A possible reason for this problem is programming error.

The third problem is a form link problem. A form link application is often encountered by the user, an example is online purchasing. This problem is more complex than the dynamic link situation as described above because in addition to programmer errors, users might provide with special characters that exercise the robustness of the application may also cause web application failure.

The forth problem is the dynamic page creation problem. The dynamic page generation involves database state and/or server state. Many web applications have a dynamically created web page to reply to users requests. For example, a dynamic page creation may depend on the database server processing an SQL query and format the result in a page. It is worth mentioning that the server state may incur errors under some circumstances. For example, the page contains a time sensitive greeting. If a user logs on to a particular server before noon, then the dynamic created greeting page will display good morning! However, if a user logs on to the same server also before

noon but at different geographic region, then a failure could occur because of the time zone difference.

The fifth problem is an uncontrolled flow transaction problem. This kind of problem may cause an inconsistent web page rendering result even though the web application code is flawless. Navigating a website can be done by clicking the provided links from the web application and/or pressing the control buttons (back, forward, refresh) provided by the web browser. In addition, some advanced users might change the URL information from the URL address bar to arbitrarily jump from one web page to another. There are also many inconsistent cases when rendering web page. One example, after reading a given email and then using the back button to return to the inbox page, the inbox shows that the email message has not been opened yet (MSN Hotmail)! Another example, a web application provides web-based user interface for editing inventory. The JavaScript OnInit() function will retrieve the data from the database that is displayed in a form which can then be updated and submitted back to the database. If information is updated *without* clicking the submit button and the back button is utilized to return to the previous page, followed by the forward button to arrive at the updating page, all the changed data previously entered will be missing and the old data is reloaded onto the form again!

The sixth problem is the syntax error problem. For the HTML code, the common syntax error is missing the closed bracket. For example, <h1>Advanced Search</h1>, if we omit the </h1> closing bracket, the most updated web browsers would still be able to render the "Advanced Search" with the text size "h1". However, if we are missing too many closing brackets, then the rendered page will have the anomaly situation especially if are rendered tables.

Finally, other problems associated with software connections such as the database connectivity problem (JDBC, ODBC etc), migration of legacy software components, web services, security problems, etc. These problems are outside the shaded circle (Figure 1) which is beyond the scope of this paper. Table 3 lists the potential problems already discussed and occur within the shaded circle.

	Potential Web Application Problems
1	Static link
2	Dynamic link
3	Form link
4	Dynamic page creation
5	Uncontroled flow transaction
6	Syntax error

Table 3: Potential Problems

3.2. Website Taxonomies

The complexity of web application testing depends on the class of the website. There are three classes of websites [42]. Class 1 is the "brochure-ware" type. This kind of website is static and provides very limited functionality with no user interactivity. This class of website is the easiest to develop, maintain and test. Class 2 websites provide client side interaction. It contains client side script languages like JavaScript, which makes them more difficult to test and analyze than the Class 1 website. Class 3 websites contain the same capabilities as Class 1 & 2 with the additional dynamic contents that are supplied by the database server via database connectivity. This class is the most complicated to design, implement, test and deploy. This paper will focus on the web application testing of Class 3 websites.

3.3. Industrial Web application testing

Although there are many industrial web application testing tools in existence, they primarily focus on testing non-functional requirements rather than functional requirements. For example, some of the tools address the compatibility of web applications for different browsers, usability, portability, syntax, broken link detection, page properties and handling large amount of concurrent users' request. There are more than 300 web-testing tools that can be found at www.softwareqatest.com at the present time. However, less than 15% of web application tools target the testing functional requirements.

Chapter Four

4.0 Anatomy of web application testing

In this section, we will discuss each group of testing methods listed in Table 1. Within a group, we will discuss individual methods in detail and compare their testing approaches, testing target, and coverage criteria. Consequently, strengths and weaknesses among methods within the same category will be evaluated.

4.1 Formal Methods

In this group, two testing models use formal methods to conduct web application testing. The first method is the TestUml model. The second model is the statechart model. The statechart is the extension of the finite state machines of its states-transition diagrams (state diagram for short) by David Harel [11].

4.1.1 TestUml Model

Bellettini *et al* developed the tools WebUml [5] and TestUml [4] for web application testing. WebUml generates class and state diagrams via static and dynamic analysis of web applications in the Unified Model Language (UML). TestUml uses the generated models from WebUml to perform the web application testing.

An automatic design recovery tool, WebUml, can construct a class and state model by analysing source code and interacting with the web server. This tool focuses on static, active, and dynamic web page analysis. A static page is a simple HTML page. An active page is a client page that contains client side scripting languages such as Javascript. A dynamic page is a page that executes server side code.

WebUml consists of three phases to construct web application models. The three phases are analysis and information extraction, model construction, and model visualization. Figure 2 shows the WebUml architecture. The class diagram constructor uses a meta class model similar [8] to define a generic web application structure and generate class models represent as a UML class diagrams (the structure and components of the web application). The state diagram constructor requires a live connection with the web server hosting the application and takes the class diagrams

data, source code and the user input setting information to generate state diagrams [5] to create the UML model. This model consists of web application components, behaviours and navigational structure. Information archive contains common information of the web application for both conductors to analyze. Diagrams are saved in XMI (XML Metadata Interchange) format and can be visualized by ArgoUML and/or MagicDraw.

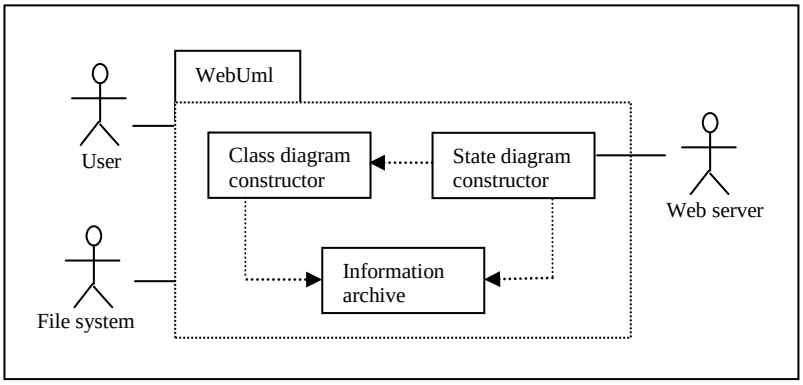

Figure 2: WebUml tool architecture [5]

TestUml takes the set of WebUml generated UML models to define test cases and chooses test paths based on random walk analysis. Random walk testing [18] ensures arbitrary path selection with the same probability to be chosen. The user also prepares the input data, expected output and the coverage criteria. The input data can also be obtained from the web server log file (user sessions) by TestUml. The expected output is used as an oracle to validate the correctness of the output. The coverage criteria are the number of web pages, objects, transition states, functions and so forth.

A strength of TestUml is the use of UML and statecharts together to generate the UML model for testing. TestUml can potentially be a complete testing method to perform fully automatic V&V processes. UML is a de facto standard modeling language that models the system, validates the user requirements (i.e. to check the system whether or not implementing user's intent to fulfill the process of validation) and is widely accepted by industry. Statechart is a formal method. Formal methods are often used to describe the system development [41] and can fulfill the process of verification. Also, TestUml does not require documentation because WebUml uses reverse engineering to extract class models.

One weakness is the coverage criteria lacks a code coverage metric. If there is dead code or a Trojan horse program residing in the web application, it will not be revealed according to their coverage criteria (page coverage, object coverage, etc). TestUml requires the availability of the source code. In addition, TestUml cannot test problems such as uncontrolled flow transaction and syntax error from (5 & 6 in Table 2).

4.1.2 State-base web browser testing

Di Lucca *et al* [22, 23] are concerned that users navigating the website using the control buttons (back, forward, and refresh buttons) instead of using the actions or events (links, form submission etc) provided by web application itself could cause failures. They propose modeling the web browser behaviour using a statechart. This model would then be used for testing. The refresh button normally does not cause errors while reloading, it will reload the page as it should. The exception is if the reloading page requires user information such as a password and the login session has expired, then the desired page will not reload successfully. For this situation, whether or not this is considered an error, we would need to consult the web application documentation. Therefore, Di Lucca *et al* suggested the removal of this reload link from the transition tree to reduce test cases.

A user clicks links or buttons provided by the web application to navigate the website. The web browser renders and displays the requested page according to the request order and builds a sequential history of visited pages. The back and forward buttons will enable or disable depending on the sequence position of the page rendered by the browser. The refresh button is always enabled. Therefore, the web browser has a total number of four states for these buttons. The first state is both back and forward buttons are disabled (BDFD). This state occurs when a user opens the web browser and the default home page is displayed. The rendering page position in the visited pages sequence is 1 and the length of the sequence is N = 1. The second state is the back button enabled and the forward button disabled (BEFD). This state occurs when a user clicks links to visit pages and the rendering page position is at the end of the sequence. The length of visited pages sequence is N where N > 1. The third state is the back button disabled and forward button enabled (BDFE). This state occurs when a user uses the back button to go back to the first page of a sequence. The length of the visited sequence is N > 1. The last state is both back and forward buttons are enabled (BEFE). This state is shown when a user uses the back button to go back to a previous page. The rendering page position has to be in between the first and the last pages of the sequence. The length of the visited sequence is N > 2. Figure 3a shows the states of the back and forward buttons according to the rendering page at different positions of the sequence.

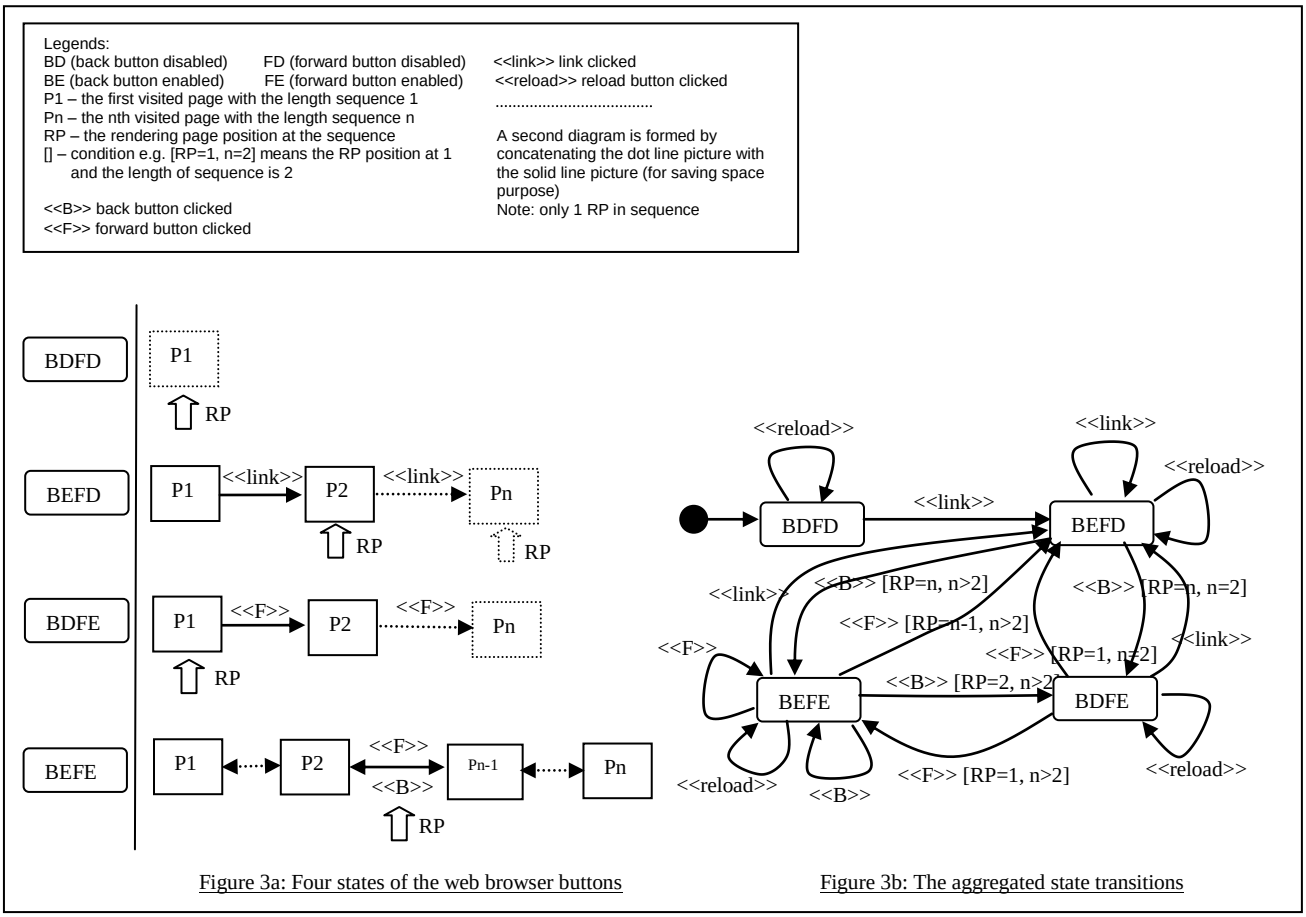

Figure 3a: Four states of the web browser buttons

Figure 3b: The aggregated state transitions

Figure 3: Four states of web browser buttons and the aggregated state transitions

Among these four states, the BDFD is a special starting state. Once a transition from this state to another state is made, this state cannot be reached again. Figure 3b is the aggregated state transitions diagram which displays changing states due to a user's action such as clicking links or buttons. It is worth mentioning that if a user clicks a link to visit a new page at the BEFE state, then the chain of forwarding pages is terminated and the new page will be the last page of the chain. Then the browser state will change from BEFE to BEFD. In addition, if a user clicks a link and a new window pops up, then a new instance of statechart will be instantiated. As a result, a composite statechart will be used to join the interaction between the old statechart and the new statechart.

This web browser testing cannot be utilized alone, it needs to be integrated with the other testing techniques. This means that web browser testing itself is not a complete testing method. This web browser testing method creates supplementary test cases for web application testing. Suppose we are going to test the function provided by the web application for user login. The Login page provides two input boxes (UserId and password) and one submit button. If the user identity

and password are both correct, then the LoggedPage will be displayed. Otherwise, a LoginErrorPage will be displayed. In order to choose test cases, we need to flatten the statechart of these three pages and build a state transition tree. Figure 4a shows the flattened statechart of these three pages and the transition tree for the correct input. The LoginPage, LoggedPage, and LoginErrorPage are represented by A, B and C, respectively.

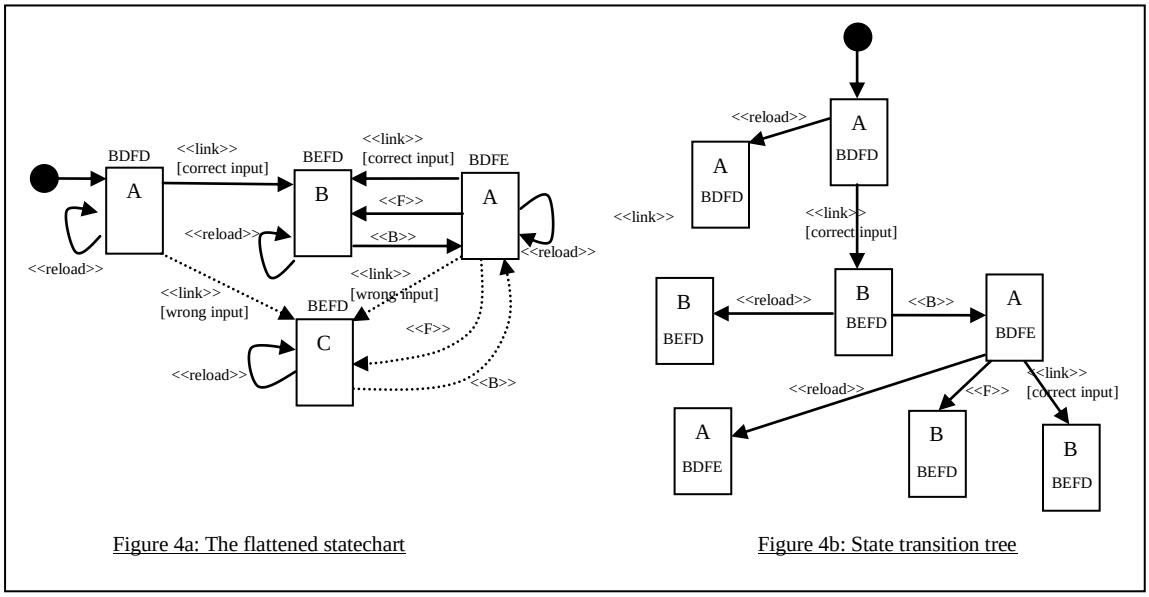

Figure 4a: The flattened statechart

Figure 4b: State transition tree

Figure 4: The flattened statechart and state transition tree

As mention before, web browser testing needs to be integrated with another testing method. The other method generates test cases is called baseline test cases. For example, to test the LoginPage, Table 4 shows the baseline test cases. From Table 4, there is only one test case (9) that should allow the user to login successfully. Indeed, we can expand this test case for more thorough testing by using the web browser testing method to generate more test cases that are obtained from the state transition tree (Figure 4b). Table 5 shows the supplementary test cases for testing the LoginPage. Test case generation can be based on the testing coverage criteria; all states, all transitions, all transition k-tuples, and all round-trip paths.

	Input variables		Results
	UserID	Password	
1	empty	any thing	LoginError Page
2	empty	correct	LoginError Page
3	empty	empty	LoginError Page
4	any thing	any thing	LoginError Page
5	any thing	empty	LoginError Page
6	any thing	correct	LoginError Page
7	correct	empty	LoginError Page
8	correct	any thing	LoginError Page
9	correct	correct	LoggedPage

Table 4: Baseline test cases

A strength of State-base web browser testing is extending the existing testing methods to generate more test cases for thorough testing. It can test whether or not there are any side effects

caused by users using the back, forward, and refresh/reload control buttons to navigate the website. Consequently, this method is good for testing uncontrolled flow transaction (Problem 5 in Table 3).

The weakness of this testing method is its integration with other testing methods. Besides, the transition tree will be large (lots of test cases) when more pages are tested. The remedy is to prune the reload path branches to eliminate redundancy (suggested by the author) because refreshing the page will reload the same page. However, this is not always the case. If we don't test the reload button, the security of time sensitive expiration pages (financial and email accounts) are overlooked.

Precondition & starting state	Current Page	User Action	Next page & resulting state
BDFD	LoginPage	test case 9, <<link>>	LoggedPage, BEFD
BEFD	LoggedPage	<<back>>	LoginPage, BDFE
BEFD	LoggedPage	<<reload>>	LoggedPage, BEFD
BDFE	LoginPage	<<reload>>	LoginPage, BDFE
BDFE	LoginPage	<<F>>	LoggedPage, BEFD
BDFE	LoginPage	test case 9, <<link>>	LoggedPage, BEFD

Table 5: Supplementary test cases

4.2 Object-Oriented Method

4.2.1 Agent-Based Testing

Agent-based framework for web application testing is based on the BDI formalism [27] and uses the UML to create web testing models [16]. BDI architecture consists of beliefs (B), desires (D), and intentions (I) with agents. Beliefs are the agents to observe the environment and communicate with other agents. Desires are the goals or targets to be achieved. Intentions are the planning of the actions in order to achieve the goals. The web testing model, any artifacts to be tested, the functional requirements and specifications, and the test results are modeled as the agent's beliefs. Agents take charge of their own task and complete the testing with other agent's co-operation. The test criteria or requirements, such as the percentage of requirements coverage for black box testing and code coverage for white box testing are desired goals. An example of intentions would be the necessary actions/activities required to improve statement coverage from 70% to the goal of 90%.

Qi *et al* [26] use data flow testing incorporated with agent-based framework for web application testing. Their testing approach analyzes the data flow information of the web application and performs four levels of testing on the web applications (function level, function cluster level, object level, and web application level). A different agent handles each level of testing. A high level test agent can create a set of low level test agents to perform the low level testing. Consequently, the

testing is completed by co-operation of agents. Test cases are obtained from the def-use chain and the criteria to stop testing require a minimum of 80% of def-use pair coverage. Details about data flow testing and def-use will be discussed in subsection 4.2.2.

The strength of the Agent-based framework is that it can effectively model and analyze complicated real world systems. This will be suitable for testing complicated web applications (built on Internet, open standard technologies, heterogeneous, dynamic behaviours, html/xml documents as variable, etc). The weakness is this method of testing relies on other testing methods. In other words, agent-based web application testing is only a framework that uses BDI formalism to steer testing; agent-based testing itself has no testing methods.

4.2.2. Object-based Data Flow Testing

Liu *et al* [20, 21] use traditional data flow testing for web application testing. In this test approach, named Web Application Testing Model (WATM), all components of the web application are modeled as objects and the approach uses flow graphs to capture the data flow information of the web application.

Data flow testing [28, 44] uses a control flow graph (CFG), to obtain test sets. Nodes in a CFG annotate variable information and edges represent the control flow between blocks of statements. The program variable in a node will be marked by either def-p-use (predicate-use) or def-c-use (computation-use) so that the def-use chain can be yielded from a graph. Control flow graphs can be extended to an inter-procedural control graph, an object control graph, and a composite control flow graph. Figure 5 shows CFG extended to an inter-procedural graph. All new introduced variables will be marked as def(variable-name, node-number). For example in fooA(), variable x appears for the first time at Line 3 (x=read()) so it is marked as def(x,1). Later, when the variable reappears in the code, it will have an annotation associated with its role of use. In the diagram, the node labelled "2-4" refers to

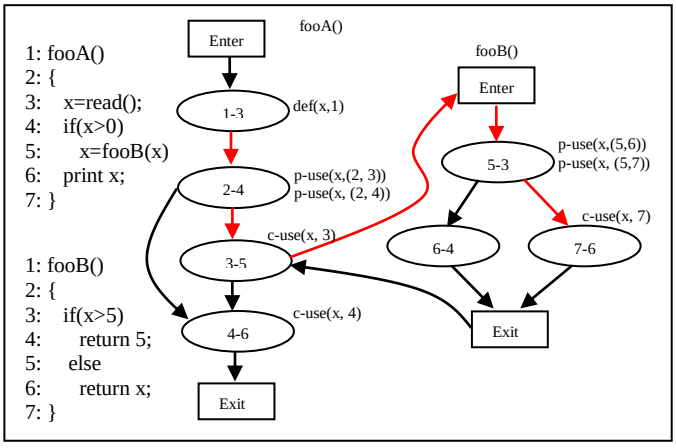

Figure 5: Control Flow Graph

the second node of the graph and line 4 of the source code. The annotation at node 2 is p-use(x, (2,3)) because node 2 refers to the if statement at line 4 and the (2,3) is associated with nodes 2 and 3. Therefore, def-use chains can be obtained from the flow graphs. For example, the chain for c-use(x,7) is (1-3)->(2-4)->(3-5)->(5-3)->(7-6).

In WATM, there are three types of objects (client page, server page, and component). Client page is HTML documents and client side scripts. Server page is a page with server side code. A component can be a java applet or ActiveX control that interacts with client pages, server pages or other components. The relationship between the objects are classified into seven types, which are inheritance, aggregation, association, request, response, navigation and redirect. An Object Relation Diagram (ORD) is a directed graph representing the objects relationships. The directed graph ORD = (V, E), where V is a set of nodes and $E \subseteq VxV$ is a set of edges connecting some nodes to establish objects relationships (each node is an object).

The data flow through the system can occur within the intra-object, inter-object, and/or inter-client. The WATM extracts test cases from these objects at five different levels. First, function level testing tests individual functions. Second, function cluster level testing tests a set of functions that interact within an object. Third, object level testing tests functions invocation sequences. Fourth, object cluster level testing tests messages passing between objects within a cluster. Fifth, application level testing tests application scope variables. Function level, function cluster level, and object level test cases can be obtained by extracting def-use chains from an intra-object. Object cluster level and application level test cases yield from inter-object and inter-client respectively.

The strength of this testing method stems from the idea of their inside-out approach. A change in state for software depends on the value of its variables. Data flow testing precisely tests all possible data flows in the system, which tests for errors within the objects. In addition, data flow testing tests for errors outside the objects, like side effects caused by different invocation sequences and the application level problem. This addresses problems 2, 3, and 4 in Table 3.

The weakness of this testing method is that test engineers must have the source code and documentation. Otherwise, it is impossible to model the control flow graphs precisely to yield def-use chains. There is also no mention about how to set the testing criteria (when and how to stop testing).

4.2.3. Object-Oriented Web Test Model Testing

Kung *et al* [17] use Object-Oriented Web Test Model (WTM) to support web application testing. WTM deploys forward and reverse engineering tools to extract static and dynamic test artifacts from a web application to create a WTM instance for testing. The forward engineering tools obtain the related test artifacts from the requirement specifications. The reverse engineering tools capture the test artifacts from the source code. Both processes can be done automatically.

In the WTM, the structural and behavioural test artifacts of a web application have three aspects; the object, the behaviour and the structure perspectives. The object perspective shows the structural relationships of a web application in object-oriented fashion by using an Object Relation Diagram (ORD). According to the ORD, test engineers can easily understand the relationships among objects. The WATM mentioned in subsection 4.2.2 is a WTM differing only in name.

The behaviour perspective consists of page navigation behaviour and state-dependent behaviour. Kung *et al* use the Page Navigation Diagram (PND) to depict the page navigation

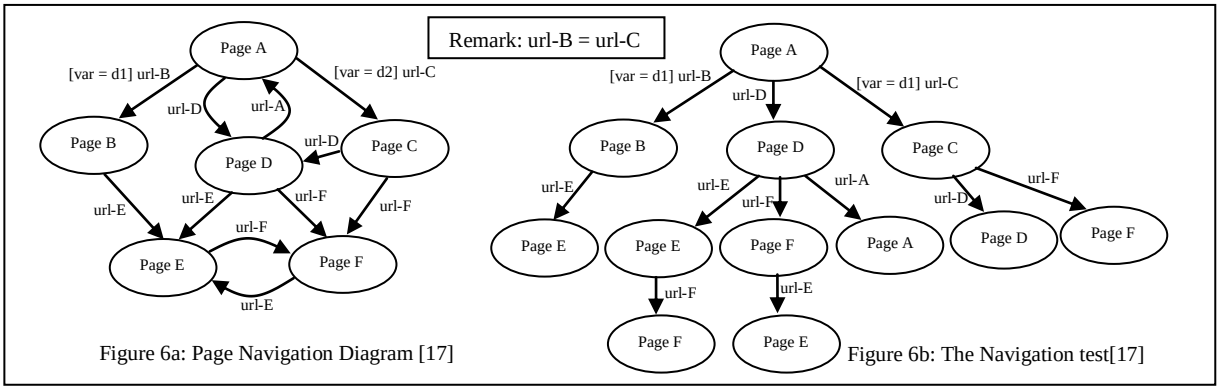

Figure 6a: Page Navigation Diagram [17]

Figure 6b: The Navigation test[17]

Figure 6: Page navigation diagram and Navigation test

behaviour. PND is a finite state machine and each state represents a client page. Transitions among states represent hyperlinks and are labelled with URL. In addition, the same link may transit to different pages. To handle this kind of situation, an extra annotation (a guard condition) will be marked within the square brackets. Figure 6a shows an example of PND. In Figure 6a, the same hyperlink in Page A can visit Page B or Page C depending on the user input value under the guard condition. Page C can visit Page D and Page F by following two different links url-D and url-F. PND helps to generate a navigation test tree which is shown in Figure 6b. As a result, test cases for testing page navigation behaviour can be extracted from this test tree. Any path, from root to any branch, can be a test case.

An object state diagram (OSD) can represent the state-dependant behaviour of the objects in a web application. OSD is similar to statecharts. It can also represent concurrent communication between objects by using the composite object state diagram (COSD). Figure 7a and 7b shows an example of OSD and COSD. Figure 7a documents the primary interacting objects performing the transfer balance for a web banking application. The OSD C, T, S, A represent the TransBalance client page, TranBalance component, TransBalance server page, and Auth server page respectively. The OSD C has three states (idle, accept, and wait), T has five states, S has three states, and A only has 2 states. When there is no input at the client page, the states of all four OSDs are idle i.e. (C_1:S_1:T_1:A_1). If an input is accepted by the TransBalance client page, then the state of C is C_2 and the others still remain idle i.e. (C_2:S_1:T_1:A_1). When a user clicks the submit button, it triggers the other OSDs to change their states and the COSD diagram depicts the concurrent communication among objects. For example, after submission, TransBalance client page will be at state 3, waiting for the TransBalance server page response. In the TransBalance server page, the OSD S will change its state from idle to process and the others remain at the idle state i.e. (C_3:S_2:T_1:A_1). Since the OSD S is at the process state it can get information from TransBalance component or redirect to Auth server page; therefore, the subsequent state changes would be from (C_3:S_2:T_1:A_1) to, (C_3:S_3:T_2:A_1)

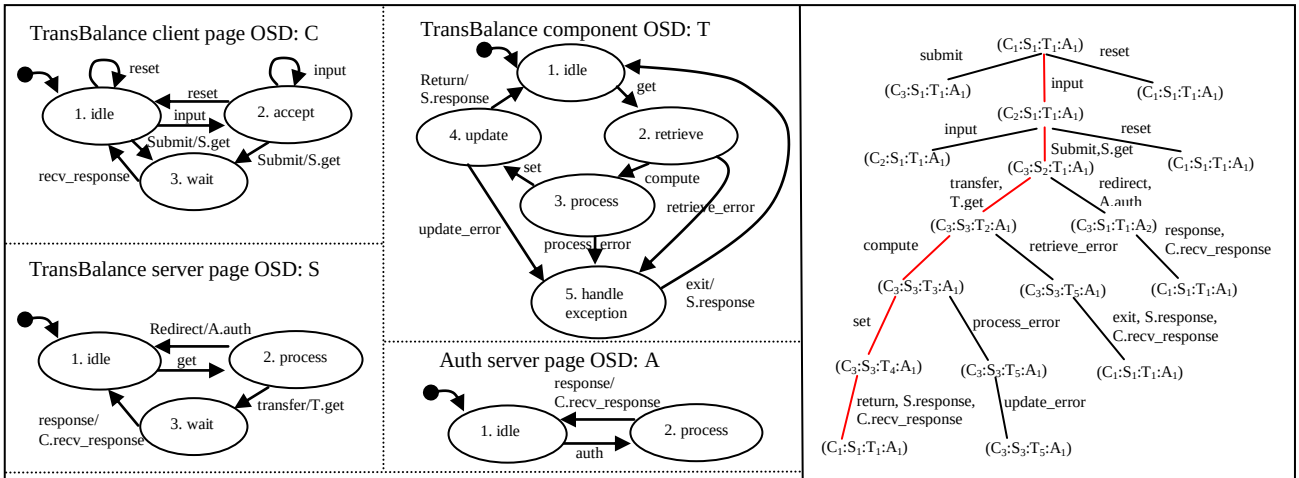

Figure 7a: An example of OSD and COSD [17]

Figure 7b: COSD test tree [17]

Figure 7: An example of OSD and COSD, and COSD test tree

or (C_3:S_1:T_1:A_2). Consequently, a COSD test tree can be built based on the COSD. Then we can obtain test cases from the test tree for testing object state dependant behaviour. In Figure 7b, the nodes of test tree flatten the objects concurrent communication. In order to test the correctness of their interaction behaviour, test cases can yield from the root to any nodes of the tree. Testing the behaviour (navigation and state-dependant) of a web application has already been discussed. The

remaining perspective aspect is to test the web application structures. The original paper mentions using a Block Branch Diagram [17] to create a test model, the idea of choosing test cases is similar to data flow testing. Therefore, there is no need to repeat the discussion at this section.

The strength of WTM testing is that it not only tests the correctness of code implementation; but it also tests the navigation and state-dependant behaviours of a web application. This is a very thorough testing method. It can test Problems 1, 2, 3 and 4 from Table 3.

On the contrary, this testing method has no obvious weaknesses. Its weakness is the common weakness of white box testing methods that requires the availability of the source code. On the other hand, they have not mentioned the coverage criteria to stop testing. It is important to know the type of coverage criteria to stop testing in order to yield optimal performance of the testing system.

Throughout the discussion of this group, we find that Agent-based testing is merely a testing framework and WATM model is considered a subset of WTM. Therefore, it can be concluded that WTM is the best method for Object-oriented approach.

4.3 Statistical Method

There are three groups of researchers that use a statistical testing method for web application testing [6, 7, 14, 15, 43]. Coincidently, all of them use the Markov chain model for statistical testing. However, their ways of creating a usage model of the web application and finding errors are slightly different. The Markov chain is a discrete-time stochastic process model with a memoryless property (Markov properties). This means state transitions from a current state to another given state is primarily dependent on the current state and is not related to the history of transitions. Contrastingly, modelling card games to predict card players' next move cannot use the Markov chain model because the memory of each card drawn from the deck influences the player's decision. Before we further discuss statistical testing using the Markov model for web application testing, we need to determine if the Markov chain model is suitable for web application testing. Li *et al* [19] answered this question affirmatively. They set up two web usage models (history independent and history dependent) for statistical testing. The history dependent usage model contained users using search engines to find the starting page then from the starting page

transitioning to another page of information. The probability of arriving at a designated webpage using either history dependent out-links or history independent (Markov chain) out-links is similar.

In order to conduct statistical testing for web applications using a Markov chain, test engineers need to prepare the web usage model. Each state of the model represents the possible navigation of the web application. Figure 8a shows the simple web usage model (the Exit state indicates a user visiting pages at other websites). After building the usage model, each state's out going edge has to be labelled with its transition probability. If there is no usage information available, then the transitions probabilities will be uniformly distributed (shown in Figure 8b). If the usage information is known, then it is most likely that the transition probabilities will not be uniform. For example, many web servers record user navigation activities on the website into a log file. After extracting the information from the log file (3 times from Page A to Page B, 2 times from Page A to Page C, 2 times from Page A to Page D …), the probability is obtained by normalizing the count of usage frequencies into relative frequencies. When the transitions probabilities are established (shows on Figure 8c), the Markov chain is completely defined with a best estimate of a real usage model.

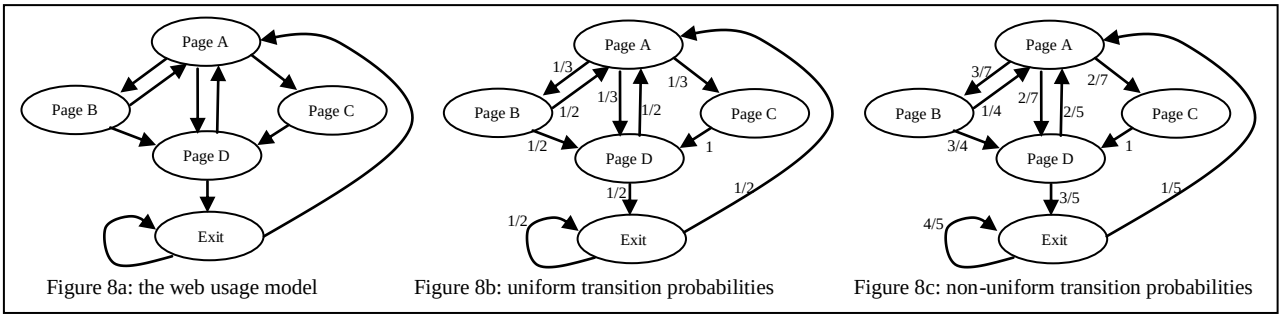

Figure 8a: the web usage model Figure 8b: uniform transition probabilities Figure 8c: non-uniform transition probabilities

Figure 8: Web usage model, uniform transition probabilities, and non-uniform transition probabilities

The usage of statistical testing can be applied to different aspects, for example, the most frequent failures in practical use early in the test period can be revealed. We can analyze users navigation habits or predict the length of time required to finish testing to comply with time-to-market requirements. More precisely, if the Markov chain is stationary and we want to know the probability of a user visiting Page D from Page B, then we can obtain the stationary probability of Page B to Page D by solving a set of equations: $\pi_j = \sum_i \pi_i U_{ij}$. The stationary probability π_j accounts for the asymptotic appearance rate of state j, where the Page B and Page D are i and j, respectively. The U_{ij} is the transition probability from i to j (Page B to Page D). Details and some useful equations can be found at [45].

Among the three groups of researchers, the procedures and purposes of conducting statistical

testing for web application are pretty much the same, such as finding broken links, reliability measure, realistic evaluation of system performance, and mean time to failure for the software certification. Test case generation are obtained by randomly choosing a Markov chain, using probability thresholds (e.g. if threshold is 0.1, then the test case from Figure 8c will be Page B to Page A to Page D since 1/4 times 2/7 less than 0.1), and/or utilizing the real usage patterns from the log file. As for the criteria of stop testing, the occurrence of failures or percentage of states coverage can cause testing to halt. Although their procedures and purposes are similar, there are slight differences in their models and consideration of failures.

Chang *et al* [6, 7] use Microsoft FrontPage to build a navigation map of a website to create a Markov chain model and use ToolCertify from Q-Lab [25] to generate test cases, perform statistical analysis and provide the test report. We are unable to get information from this tool's user guide; therefore, nothing more can be addressed here. Kallepalli *et al* [14] use FastStats [46] log file analyzer to produce the hyperlink tree of the website structure. Then they use this structure to create a Markov chain model named Unified Markov Models. The only failure consideration of their method are errors present in the log file such as file not found or permission denied. Tonella *et al* use UML to capture the dynamic aspects of a web application to create a Markov chain model. Therefore, they can test errors beyond the static failure information logged in the log files. However, this requires the tester to manually determine the correctness of the dynamic generated pages. Table 6 shows summary their methods.

	Testing Methods		Testing Problems						Coverage Criteria		
Model	Black box	White box	1	2	3	4	5	6	Failures	Probability threshold	State transitions
Unified Markov	x		x						x	x	x
UML Markov	x	x	x	x	x	x			x	x	x

Table 6: Unified and UML Markov models summary

The strength of web application statistical testing is the use of statistical techniques to unfold realistic problems in the website being tested, such as mean time to failure, users navigation habits etc. This method is very good for website maintenance; to maintain the high standards of the website. Between the Unified Markov model and the UML Markov model, the latter can also test the correctness of dynamic generated pages. This method can handle problems 1, 2, 3 and 4 defined earlier in this paper (Table 3).

The weakness of this method are the difficulties modelling the Markov chain properly and

finding the eigenvector (π) of the transition matrix U with the eigenvalue of 1 ($\pi = \pi U$); which requires proper training in stochastic problem solving and strong mathematics skills, respectively. It may not be widely adopted by industries. In addition, we do not have real transition probabilities to test the first hand copy of the web application before deployment. Therefore, it may not accurately unfold realistic situations (user navigation habits, mean time to failure, etc), except perhaps finding the broken links.

4.4 UML Method

Ricca *et al* use ReWeb and TestWeb tools to support web application testing. ReWeb and TestWeb [29, 30, 31, 33, 34, 36] perform their operations (analysis and testing) based on the abstraction of the web application. ReWeb uses the Unified Modeling Language (UML) to build this abstraction model.

Basically, ReWeb is a design recovery tool. It consists of three components, which are a spider, an analyzer, and a viewer. A spider downloads all pages from a given URL. For any interested dynamic pages, users have to provide input values before the spider starts downloading the page(s). Any page(s) related to the given URL but outside the website host, the spider will not download. An analyzer uses a website UML model and all the downloaded pages to conduct web application analysis. A website UML model is a meta model of a web application. This will be discussed in more detail in the next paragraph. A viewer facilitates the view of the structure and the system of the web application and the textual reports in a GUI.

The core entity of the web server is the web pages. There are four levels of web page classification [33]. Level 0 is a static page without frames. Level 1 is a static page containing frames. Level 2 is a dynamic page with no data input from a client. Level 3 is a dynamic page with data input from a client. A website UML model is a meta model of a web application. Figure 9a shows the meta model of a web application. Figures 9b and 9c show two instances of this meta model which are a web page with frames and a web page with form. In Figure 9b, p1 is a web page containing two frames and the two edges e1 and e5 indicate p1 is split into f1 and f2. The link e2 between f1 and p2 represents the initial page is loaded to p2 and the same explanation applies to the link e6 between with f2 and p3. The link e7 between p3 and p5 is a normal navigation connection

between HTML pages and the same explanation applies to the e4 between p4 and p5. As for the link e3, if the data member of the association class LoadPageIntoFrame f:Frame is f2, then the p4 will be forced to load at f2. In Figure 9c, a static page contains a form for a user to input two variables (x1 and x2). The dynamicPage will generate a static page either p3 or p4 based on the value of x1.

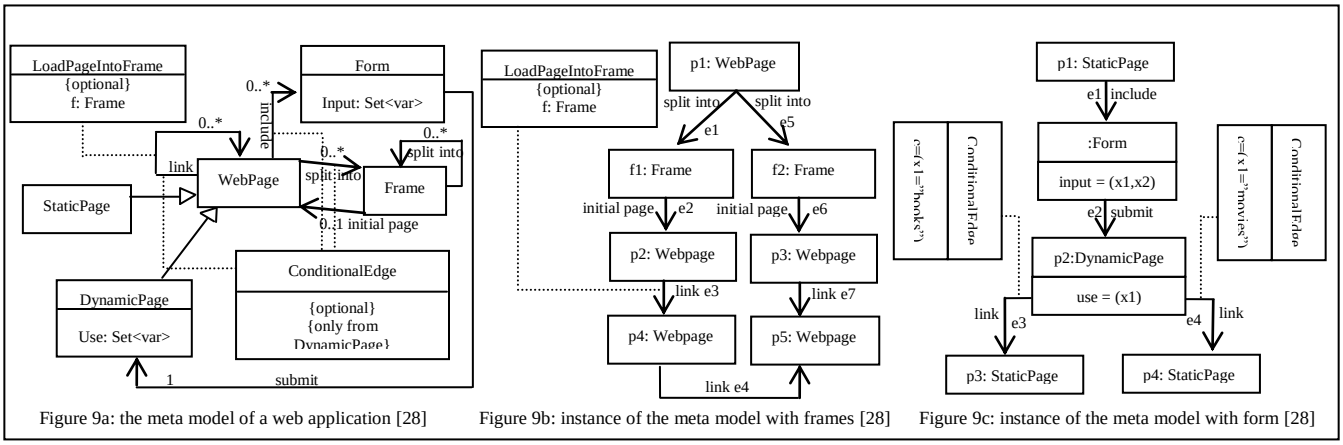

Figure 9a: the meta model of a web application [28] Figure 9b: instance of the meta model with frames [28] Figure 9c: instance of the meta model with form [28]

Figure 9: The meta model of a web application, and instance of the meta model with frames and form

TestWeb consists of two modules (the test cases generator and the executor). The test cases generator generates test cases from the instance of the web application UML model. The generator uses the method of path expression (algebraic representation of the paths in a graph) [1] to compute the testing path from the web application UML model. Paths are the edges and variables are the labels such as e1, e2, etc in a graph. Variables can be combined together with two operators ('+' or '*'). The '+' and '*' operators represent selection and loop, respectively. The path is the test case and should be a linearly independent path (at least one edge has not been traversed before). An example of a test case obtained from Figure 9b is e1e2e3e4 + e5e6e7 (from page 1 to page 5) and from Figure 9c is e1e2(e3 + e4) i.e. the initial page starts at p1 and reaches to p2, then makes a selection at p2 to either p3 or p4. In the case where loops exist in a graph (Figure 9c), we can use '*' to represent loops. For example, if p3 and p4 have a connection back to p1, then the test case will be (e1e2(e3 + e4))*.

A strength of ReWeb/TestWeb is that it not only validates the user requirements by using UML to model the web application but it also verifies the correctness of the code implementation. In addition, ReWeb also provides anomaly detection (like syntax check), monitors the history of website changes and analyzes the website structure to provide restructuring information to increase the maintainability of the website. The coverage criteria can be pages, hyperlinks, data flow, among

a few. It handles problems 1, 2, 3, 4 and 6.

The weakness of ReWeb/TestWeb is the ReWeb can only fully handle level 0 to level 1. The test generator is unable to distinguish between static and dynamic pages and requires tester intervention. Otherwise, ReWeb/TestWeb can be considered an adequate tool for web application testing.

4.5 WebApp Slicing Method

Ricca *et al* introduce the novel method of Web application slicing [32, 35] to perform web application testing. Web application slicing is more complex than slicing traditional software. It is because the data flows in the web application system are not limited to passing variables from statement to statement. The web application can generate a web page dynamically and embed the variables into the generated code of the new page. The dynamic generated code can also pass along these variables to another dynamic page.

The definition of traditional program slicing is to reduce the size of a given program while preserving the original behaviour of the given program so that testers can focus on this part of interest for testing. This definition also holds for web application slicing. In traditional program slicing, control dependence and data dependence are the ingredients for slicing the program. An individual program is represented by a program dependence graph (PDG). In order to handle a program that contains more than one procedure, we need to first build a PDG for each procedure. Second, we add an extra "call node" to the graph to connect the actual arguments to the formal arguments. The resulting graph is called a system dependence graph (SDG). A SDC is used for interprocedural slicing. Ricca *et al* simply use a System Dependence Graph (SDG) explicitly displaying all kinds of dependencies. In the following subsection we will discuss how to model a SDG for control dependence, data dependence, and call/parameter-in dependence applies to web application slicing. For other dependences, details can be referenced from the original papers [32, 35].

4.5.1 Nesting/Control dependences

Control dependences take place at the conditional statement and loop statement of traditional programs. The instruction of execution depends on the truth value of the predicate. This is also true for server side script of the web application. Moreover, an additional kind of control dependence (nesting) is found in the HTML code because HTML code can be nested with scripting languages and HTML, or HTML statements itself e.g. <TABLE><TBODY> <TR> <TD ></TD></TR></TBODY><TABLE>. The browser can interpret the HTML page correctly if all enclosing tags are available. Control dependence is depicted by the SDG in Figure 10. A snippet of cd.php contains a total number of 10 lines of code. Each line of statement will be represented by a node in the SDG. This SDG will not contain lines 5, 6, and 10 because there are no statements related to them. Considering line 2, if the value of $x is "xx", then lines 3 and 4 will be executed and the variables $y and $z will be assigned a new value. Therefore, control dependences exist for lines 3 and 4 based on the truth value of the predicate "if" at line 2. As for line 7, regardless of the truth value of the predicate, it will be executed anyway. Therefore, the line 7 does not have any dependence. It is for this reason that line 7 is the starting node of a new sub-tree in a SDG. Lastly, the lines 8 and 9 nest between line 7 and line 10; therefore, they are dependent on the line 7.

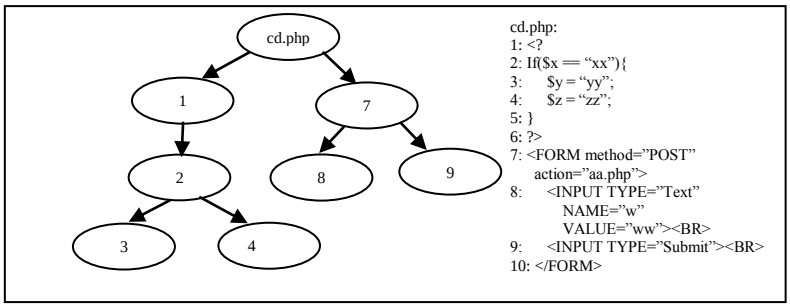

Figure 10: A SDG for nesting/control dependences

4.5.2 Data dependences

The definition of data dependences in tradition program is if B is data dependent on A, then A declares or modifies a variable that B references for its value and there exists a clear path (no redefinition of A) from A to B. This also applies to web applications when the data flow occurs within the server side code or from the server side code statements to HTML statements. On the contrary, if the data flow occurs from HTML code to the server side code, then it

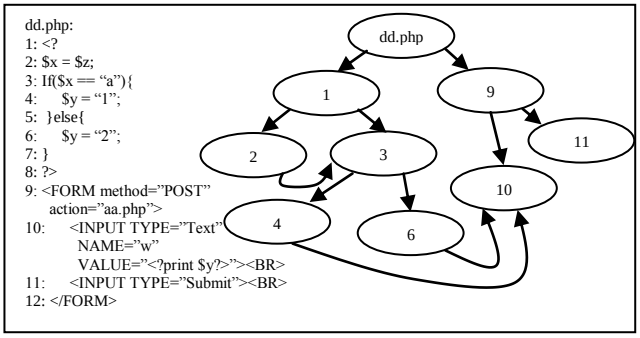

Figure 11: A SDG for data dependences

must be propagated through other dynamic pages as a submission parameter. These situations will be discussed in sub section 4.5.3 (call dependence condition). Figure 11 shows a SDG to model data dependences. In Figure 11, there are two kinds of data dependences (within the script code and from script to HTML code). The straight line with arrow head represents the control dependences and the curve line with arrow head represents the data dependences.

4.5.3 Call/parameter-in dependences

Interprocedural data dependences in traditional program has a calling context problem (difficult to keep track of the data flows associated with different call sites). This problem also exists within the server side code but the client code to the server side code does not suffer. This is because the client page invokes the server side program, passes the execution control to the server side program and the invocation never returns to the calling page. Figure 12 depicts the variable flow from client code to the server side code. In this graph, the call dependence occurs at line 5 (aa.asp) and bb.asp. The values of parameters (x and y) from aa.asp will be assigned to the local variable ‘a’ and ‘b’ in bb.asp. Therefore, the parameter-in dependences exist between nodes 6 and 11, and nodes 7 and 12. The dotted line with arrowhead represents the call dependence.

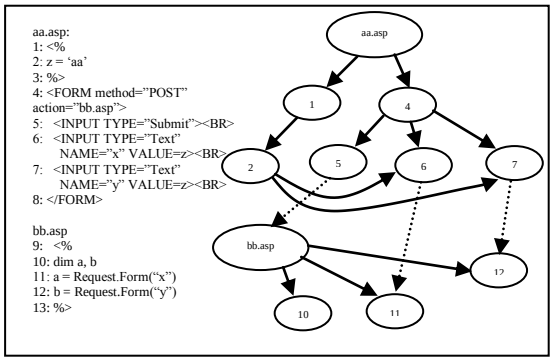

Figure 12: A SDG for data dependences from HTML code to server side code

Based on the nature of the slicing method, it slices out a small piece of the program for comprehension and correction. Test engineers are not distracted by a huge amount irrelevant code. It is well fit for debugging, maintenance (what is the consequence to change a couple lines of code), integration (add new methods), and testing.

On the other hand, what are the benefits of slicing to test an initial copy of a program? If we test all the slices and find no errors, do we still need to test the whole program again? If yes, why would we use the slicing method to test an initial copy of a program? The slicing method will have sliced such that all possible test cases would have been covered – but the program as a whole has still not been run once, only bits and pieces of it. The original paper did not mention how test cases are chosen (maybe test all the slices), and the criteria for stop testing. In my opinion, slicing should

be used for special purposes of testing as described above but not for validating and verifying an initial copy of software, including web applications. This illustrates a weakness of the slicing methodology.

4.6 User Session Data Method

Elbaum *et al* introduce an inexpensive User Session Data method [9, 10] for web application testing. Sprenkle *et al* suggest other ways of test cases generation for user session testing [38, 39, 40] to overcome the problem created by the role of state in web application while replaying the user session. Sprenkle *et al*'s approach explains why Elbaum *et al*'s multi-user data session test cases is unable to reveal more faults than single user session test cases.

User session based testing is an inexpensive testing method. It can obtain test cases on-the-fly with minimal effort from the instrumented web server. The instrumented web server logs user's URL and name-value into a log file, then applies heuristics to generate test cases. This is completely unlike other methods that require building an expensive static model from the code inspection followed by obtaining the test cases from this model. There is nothing interesting about how they log the user's information. We are interested in how they extract test cases from the log file and how they replay these user sessions in different ways to yield different testing results.

Elbaum *et al* suggest three approaches of replaying user sessions. Let $U = \{u_1, u_2, u_3, \ldots\ldots u_m\}$ is a set of user sessions, each u_i consists a total number, n, requests $r_1, r_2, r_3, \ldots\ldots r_n$ and each request r_i has a user URL and name-value pair. The first approach is the most straightforward one. It is to replay an individual user's sessions sequentially. The technique is $\forall u_i \in U$ transformed to a test case by formatting each of its requests $r_1, r_2, r_3, \ldots\ldots r_n$ into a HTTP request. Consequently, each session has one test case and the test suite will have a total number of m test cases. This approach is a constrained version (only captures the URL and name-value pairs of the entire session) of a capture-replay tool. The second approach is to replay different combinations of users sessions. The purpose is to expose errors that are caused by different users providing conflicting data. The algorithm to generate the test case is as follows:

1. select an unused session u_a, where $u_a \in U$
2. copy a total number of i requests (1 to i) from u_a, where i is a random number and $1 < i < n$

3. select session u_b randomly, where $u_b \in U$ and $b \neq a$, then search for any r_j in u_b with the same URL as r_i
4. if no matching URL is found in u_b, then go back to step 3. If no more session remain, then use u_a as a test case
5. if a matching URL is found in u_b, then append all the requests following r_j from u_b to r_i to make a test case
6. mark u_a as "used", then repeat the process until all sessions in U are used.

The third approach is to reuse user session with Form modifications. This method can be achieved by random deletion of some characters in a string, associated with the name-value pairs. The algorithm is below:

1. select an unused session u_a, where $u_a \in U$
2. select unused r_i from u_a, if no more unused r_i from u_a, then use u_a as a test case
3. if r_i does not have any name-value pair, mark it as used and repeat previous step
4. if r_i has name-value pair(s), then modify by deleting characters randomly to create test cases
5. mark u_a as used and repeat the same process until there are no more unused sessions available in U

Elbaum *et al* conducted an empirical study to measure the effectiveness of this user session testing technique. Their empirical study used three different approaches (white box, user session, and hybrid) to generate test cases to test a mutated E-commerce site by seed faulting. They extracted the white box test cases from a static model, which was built based on Ricca *et al*'s idea with some assumptions. The hybrid test cases were a mix of white box testing requirements and user sessions data i.e. obtaining a test path from a static model followed by translation into a URL sequence, if this sequence matches any user sessions, then the matched session is selected to be a test case. In this paper, we do not discuss which kind of test case is better. We are interested in why the second approach (combining different users' sessions) of user session test case generation cannot reveal more faults. The primary reason for this situation stems from ignoring the role of state in the web application while replaying the users' sessions data. Sprenkle *et al* address this situation and provide a new way of generating test cases.

Captured Log		Deployed Behavior
User1	index.jsp	Enter bookstore site
User2	addBook.jsp	Add Steve Martin's Pure Drivel to database
User1	search.jsp	List Steve Martin's books
User1	buy.jsp	Buy Pure Drivel

Figure 13a: Captured Log [37]

Replayed Log		Replayed Behaviour
User1	index.jsp	Enter bookstore site
User1	search.jsp	List Steve Martin's books (does not include Pure Drivel)
User1	buy.jsp	ERROR: Attempt to buy Pure Drivel, a non-Existent title in database
User2	addBook.jsp	Add Steve Martin's Pure Drivel to database

Figure 13b: Replayed individual user session sequentially [37]

Figure 13: Captured log and replayed individual user session sequentially

Sprenkle *et al* found that mixing different user session test cases lost the application state dependencies. For example, referring to Elbaum *et al*'s first algorithm to generate a user session

test case is to replay an individual user session sequentially. If the server logs the user's requests as shown in Figure 13a, and the test cases for the test run is based on replaying the individual user session sequentially, then arises the problem caused by mismatching the state of real usage of the web application and the replay state. Figure 13b shows the individual user session sequentially replayed and the problem caused by this first algorithm. In Figure 13b, User1 made a request to buy a book after browsing the site and the error occurred because this desired book did not exist in the database. In fact, User1 did buy the book! This reflects that the replayed test cases are unable to reflect the real time usage situation because of the loss of state dependency. In order to overcome this situation, they suggest three new automatic test case generation methods; partition the full log by fixed time blocks, server inactivity threshold, and augmented user sessions. Figure 14 is an example of a captured thirty-six minute server run log used to demonstrate Sprenkle *et al*'s three methods of test cases generation. The three methods are summarized as follows;

Fixed-Time Blocks

The first method is simply partitioning the web server log into fixed-time blocks. Figure 14c shows the full log broken down into smaller time blocks of ten-minute intervals. This approach can lower the chance of losing state dependency compared to replaying the user sessions test cases sequentially. In addition, the test case is a multi-user test case rather than a single user test case. Therefore, the test cases can closely reflect the real usage situation. Also, smaller test logs are easier to debug than a full log. The main disadvantage of Fixed-time Block is the logical user session may be split into different test cases by stringent partitioning of the log into a fixed time length.

Server Inactive Threshold

In order to maintain a high level of multi-user test cases and to reduce the number of logical sessions split across into different test cases, they propose partitioning the log based on a threshold of a server idle time. Figure 14d shows this method using a four-minute threshold of idle time to create a total number of three test cases.

Augmented User Sessions

Splitting logical user sessions into different test cases can also be overcome by

using augmented user sessions approach. The test case is extracted from one user's request as a starting point and ending when the same user has no more requests. Figure 14e shows test cases generated by augmented user sessions.

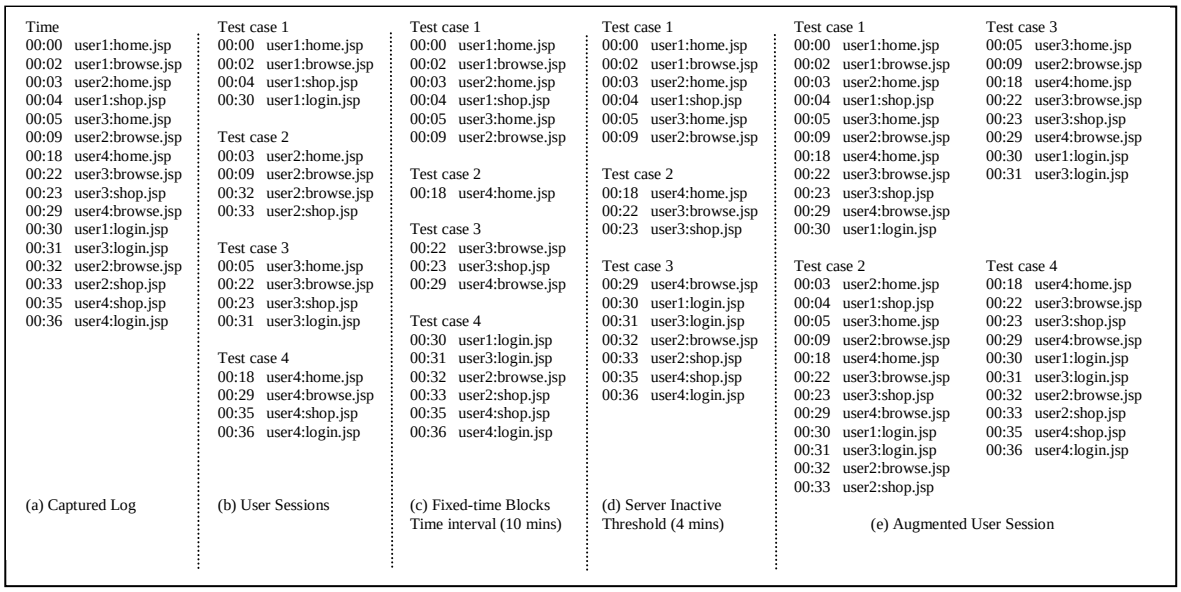

Figure 14: Test cases generation demonstration

Comparing the user session testing with different ways of test cases generation by Elbaum *et al* and Sprenkle *et al*, the former method is unable to reflect multi-users usage of the web application and the test cases lose the web application state dependency. Among Sprenkle *et al*'s three methods, Fixed-time Block requires careful choice of the right time interval. If the time interval is too long, then it becomes harder to debug when bugs are found. On the other hand, if the time interval is too short, it will increase the number of logical sessions split across test cases. As a result, redundant testing process occurs and degrades the testing performance. The success of Server Inactive Threshold method is dependent on choosing the appropriate threshold (idle time). In my opinion, this method is not applicable for any busy servers. Some busy servers never have idle time! Apparently, the Augmented User Sessions do not have any problems but enlarge the test case size, thus slowing down the testing process.

User session testing generates virtually costless test cases. This method does not have a tight coupling relationship with the web application protocols. In the other words, whenever web application technologies change, this method of test case generation will not be affected. On the other hand, if the generation of test cases is based on a static model, then the model may not be compatible with new changes in technologies. These are the strengths of the user session testing

method, enabling it to address problems 2, 3 and 4 in Table 3.

The weakness is that we do not have initial user session data to test the initial copy of the web application. This means we can only use this method for regression testing.

Chapter Five

5.0 Overall evaluation

In this section, we start with a brief review of what each testing group can and cannot achieve in order to evaluate their overall performances. In such a way, we are hoping to find one testing method that can address all the potential problems presented in Table 3 of this paper.

Group 1 - Formal methods

TestUml tests static links, dynamic links, form links, and dynamic page creation problems. The state-base web browser tests uncontrolled flow problems.

Group 2 - Object-oriented methods

Agent-based testing itself is not a testing tool. Object-based data flow testing can handle dynamic links, form links, and dynamic page creation. Object-oriented web test model tests static links, dynamic links, form links, and dynamic page creation problems.

Group 3 - Statistical methods

Unified Markov only tests the static links. UML Markov tests static links, dynamic links, form links, and dynamic page creation problems.

Group 4 – UML methods

Reweb/TestWeb tests static links, dynamic links, form links, dynamic page creation problems, and syntax error problems.

Group 5 – Slicing methods

Slicing can handle dynamic links, form links, and dynamic page creation problems.

Group 6 – User Session Data methods

User session testing addresses dynamic links, form links, and dynamic page creation problems.

In order to reflect each groups overall properties, we proposed an algorithm to work out an individual group's capabilities. The algorithm is counting number of edges formed at the binary relation of M and P (mRp). Where M is a set of testing group and the elements of M will be the testing methods in this group i.e. $M = \{m_i \mid i \geq 1\}$. P is a set of testing problems and the elements of

$P = \{p_i \mid 1 \leq i \leq 6\}$ where $p_1, p_2, p_3, p_4, p_5, p_6$ are static link problem, dynamic link problem, form link problem, dynamic page creation problem, uncontrolled flow transaction problem, and syntax error problem. Once the edge is formed by the element of m_i with the element of p_j, then one star will be awarded to this group and the element of p_j will be removed out of the set P. The algorithm is below:

1. select an unused session u_a, where $u_a \in U$
2. select unused r_i from u_a, if no more unused r_i from u_a, then use u_a as a test case
3. if r_i does not have any name-value pair, mark it as used and repeat previous step
4. if r_i has name-value pair(s), then modify by deleting characters randomly to create test cases
5. mark u_a as used and repeat the same process until there are no more unused sessions available in U

For example, Statistical methods contain Unified Markov testing method and UML Markov testing method; therefore, $M = \{m_i \mid i \geq 1\}$ where m_1 = Unified Markov testing method and m_2 = UML Markov testing method.

we use an introduce a formula $\{G = M_iT_p \cup M_jT_q \cup \ldots \mid p \neq q \neq \ldots \ \& \ i \geq 1\}$ to calculate individual group capabilities. G and M_i represent a group name and testing methods available in a group respectively. T_p represents the testing problems, which are static link, dynamic link, form link, dynamic page creation, uncontrolled flow transaction, and syntax error. For example, if group G has methods i and j that can test static link and form link, respectively, then we can treat this group's capability as testing both static link and form link. Table 7 compares the capability of six groups of testing methods to test different kinds of faults of web applications by using this formula $\{G = M_iT_p \cup M_jT_q \cup \ldots \mid p \neq q \neq \ldots \ \& \ i \geq 1\}$.

Group Method	Testing Problems						Scores
	Static link	Dynamic link	Form link	Dynamic page creation	Uncontrolled flow	Syntax error	
Formal	x	x	x	x	x		★★★★★
Object oriented	x	x	x	x			★★★★
Statistical	x	x	x	x			★★★★
UML	x	x	x	x		x	★★★★★
Slicing		x	x	x			★★★
User Session		x	x	x			★★★

Table 7: Six groups testing methods summary

After using the proposed formula to determine the capabilities of each individual group, we

summarized each group's capabilities in Table 7. The right most column (Scores) visually illustrates the tally of the capabilities of each group. One star is rewarded to a method if it can test one particular testing problem. (In Table 7, one star is rewarded to a method if it can test one particular testing problem.) The highest score obtained is shared by the Formal methods (Group 1) and UML methods (Group 4). Both of them score five stars out of six, still not a perfect score (six stars)! No single group was found that could test all the potential problems associated with the shaded area in Figure 1. Interestingly, all groups were able to handle the problems related to dynamic links, form links, and dynamic page creation. This suggests that these are the core features of web applications. However, a good web application testing method at minimum, should be able to test all the potential problems listed in this paper. As no single group can handle all the potential problems defined in Table 3 of this paper, we suggest creating a new robust testing architecture by choosing appropriate methods from different groups in combination to form a new testing tool. Details will be discussed in Section 6 Contributions and future work.

After careful evaluation, it can be concluded that no single group can comprehensively test all the potential problems listed in this paper nor does one group dominate the testing scores. However, all groups can test the core problem features of web applications. With regard to the aspect of testing performance (efficiency and effectiveness), we are unable to set up a controlled experiment to empirically study this performance.

Elbaum *et al* attempted to compare their user session testing method with Ricca *et al*'s ReWeb/TestWeb. However, the conclusion drawn did not accurately reflect performance because they made assumptions when building Ricca's testing model. Similarly, Sprenkle *et al* compared their way to generate user session test cases (e.g. Fixed-Time Block) to Elbaum *et al*'s User Session. Their comparison was based on code coverage, which is not strong enough to claim superiority. Since comparing the performance of individual methods requires careful controlled experimental setup (not a trivial task) and the testing method resources (which we lack), the evaluation of each method is therefore based on its capabilities not performance.

After surveying different groups of web application testing, we better understand the difficulty of web application testing. We are able to answer the questions posed at the beginning of this paper (Q1 and Q2) and address the open question (Q3).

Q1. What are the similarities between traditional software testing and web application testing approaches?

When we test a traditional program or web application, there are two fundamental common testing methods; white box testing and black box testing. White box testing requires the availability of the source code. We use the source code to comprehend the program's structure, and build the test model to obtain test cases. Testing criteria for example, can be based on the path coverage of the test model (path coverage) or the source code (code coverage). All these steps can be applied to both traditional software testing and web application testing. Another similarity involves state dependency. Although HTTP is a stateless protocol, the state dependency of the web application must be maintained by the system. Web applications also allow concurrent invocation by user(s). This parallels the multi-threading traditional programs and thus, both traditional software testing and web application testing must consider state dependency. As for black box testing, traditional software testing and web application testing both require input domain, expected output, pre-condition and post-condition information for each targeted function(s) with no exceptions.

Q2. What are the differences between traditional software testing and web application testing approaches?

In traditional software development, a tested program would never contain syntax errors in the code. It is because syntax errors would be detected in the earliest stage of compilation. Therefore, traditional software will not suffer syntax errors. HTML code and client side scripting languages in the web document do not require pre-compilation and thus syntax errors go undetected. Fortunately, most web browsers can handle some degree of malformed HTML code (missing close tags). However, this can lead to behavioural anomalies, in the web document, associated with the undetected syntax errors when the web browser renders the web page. This translates to less attention paid to syntax errors by the test engineers. In addition, the control flow of traditional software is predictable but the web application control flow could be nondeterministic (could be changed by web browser's control buttons). Also, web applications run under uncontrolled environments.

Q3. Can we apply traditional software testing methods to perform web application testing?

Among the six groups of web application testing, all testing methods originate from traditional software testing techniques but there are other traditional testing techniques that have not yet

been used for web application testing e.g. random testing. Basically, web application techniques are only a subset of traditional software testing techniques. Therefore, the answer to whether or not we can apply traditional software testing methods to perform web application testing is affirmative.

In summary, we found (1) testing web applications is very similar to testing traditional software, (2) syntax check is often ignored or neglected by test engineers, (3) the control flow of web applications are nondeterministic, and (4) no single testing method is capable of addressing all the problems outlined in Table 3. And we have determined that traditional software testing methods can be applied to test web applications.

So, why is there a greater failure percentage associated with web applications? According to the information we have obtained, failures should stem from the lack of a complete testing method to test ever changing, short time-to-market web applications. Therefore, the contribution of this survey paper will be to suggest a new robust web application testing method by choosing some appropriate methods from the different groups and combining them to form a new testing tool.

Chapter Six

6.0 Contribution and Future work

The first consideration when developing software is to create according to user requirements. The correctness of implementation then follows. The features of the new testing tool are based on these two premises. A new testing architecture should include three properties. First, it should help the software developer understand the client's problems so that they can validate the user requirements i.e. the process of validation of requirements. Second, the tool should have a low-in-cost and high-in-value testing mechanism to ensure that the software developer provides correct solutions to the client i.e. the process of verification of specifications. Third, the tool should provide maintenance services for the system for future modifications and integration of new components. We will now discuss choosing candidate methods to create this new testing tool, the Validation-Verification-Maintenance (VVM) testing method.

Candidate(s) for fulfilling Validation

The most appropriate candidate for fulfilling this task is the Group 4- UML method. UML is a de facto standard modeling language used to model systems that is widely accepted by industries. The UML diagram depicts the system's structure. The UML diagram can validate correctness by comparing it to the documentation obtained from the client. It checks whether the system implements the user's intent.

Candidate(s) for fulfilling Verification

In Table 6, Group 1 and Group 4 obtained the highest score; therefore, we choose these two groups as a starting point. In order to pick the right candidate(s) from these two groups, we need to analyze the similarities and differences between these two groups. Figure 15 is a set theory diagram showing their capabilities. In Figure 15, the set Group 1 intersects the set Group 4, clearly showing that the uncontrolled flow and syntax error are mutually exclusive. Neither one of these groups possess both testing capabilities (uncontrolled flow

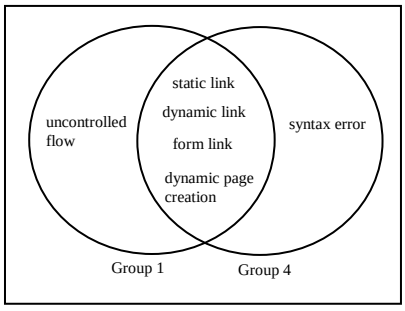

Figure 15: Group 1 and Group 4 relationships

and syntax error). Interestingly, Group 4's ReWeb/TestWeb can test not only the core features but also the syntax error. However, no single method in Group 1 can test both the core features and the uncontrolled flow problem. Between the two methods in Group 1, the state-base web browser testing method specifically tests the uncontrolled flow problem. Intuitively, this method must then be integrated with another testing method. Therefore, the candidate for fulfilling verification is Group 1's state-base web browser testing method combined with Group 4's ReWeb/TestWeb.

Candidate(s) for fulfilling Maintenance

In order to choose the right candidate for this task, we need to consider what the software developer does when performing web application maintenance. If we add a new application to the system or remove an out-dated web application from the system, then regression testing must be conducted to ensure the changes do not affect the other (unchanged) parts of the system. Therefore, we need to prepare enough test cases for regression testing. Preparing test cases is a tedious and time-consuming task. Ideally, if we can reuse test cases, then it will save a tremendous amount of time. Group 6's User session testing method is the most suitable candidate for fulfilling maintenance. It generates test cases on-the-fly at virtually no cost. Although Group 3's statistical testing method can also apply to web application maintenance (such as which page gets the most hits, mean time to failure etc.,), it is less important because the cost of test cases generation outweighs its benefits.

As a result, the new testing tool, VVM, should combine the Ricca *et al*'s ReWeb/TestWeb with Di Lucca *et al*'s State-base web browser testing method for validation and verification, and Sprenkle *et al*'s User Session testing method for maintenance of web application systems. We are not implying that the suggested idea is the best approach but currently, we have no single testing tool that addresses the problems associated with the shaded area completely in Figure 1. The principle focus of future work should be tackling the problems presented in Table 3. After which, improving testing performance (effectiveness and efficiency) for any new proposed testing tools would then follow.

Chapter Seven

7.0 Conclusion

Web application testing is more challenging than traditional software testing. Simply, web application software is tested in an uncontrolled environment because of the nature of the web system itself and the complexity of the internet. Practically, we don't house all servers (web server, application server, mail server, and database server) in one box. This implies that the web system may consist of multiple platforms sub-systems. For example, a web server may run in a windows platform, a mail server in a Linux box, and a database server in Mac. Also, we have no control over the kinds of computers used to access the web system. Despite its challenges, we can still apply many traditional testing techniques to web applications and create new ones to test problems that traditional techniques cannot, like nondeterministic control flow problems.

The new proposed VVM testing tool encompasses all three aspects of testing vectors (validation, verification, and maintenance), from the beginning of the web application design to the end of its implementation, and beyond for system maintenance, thus following through the web application life cycle. It ensures that the web application is implemented correctly without flaws. It also provides web application maintenance on-the-fly. Importantly, it stops error propagation caused by a ripple effect from the early development stage to the later implementation stage or maintenance. So far, we have not seen any existing web application testing method that can provide all of these services. Although the VVM testing tool can handle the problems presented in Table 2, this represents only a portion of a web application operation (shaded area of Figure 1). Errors may still exist outside this shaded area, therefore web application testing research still has a long way to go.

Appendix A (provided by the 'Glossary Working Party' ISTQB)

acceptance testing: Formal testing with respect to user needs, requirements, and business processes conducted to determine whether or not a system satisfies the acceptance criteria and to enable the user, customers or other authorized entity to determine whether or not to accept the system. [After IEEE 610]

accessibility testing: Testing to determine the ease by which users with disabilities can use a component or system. [Gerrard]

ad hoc testing: Testing carried out informally; no formal test preparation takes place, no recognized test design technique is used, there are no expectations for results and arbitrariness guides the test execution activity.

agile testing: Testing practice for a project using agile methodologies, such as extreme programming (XP), treating development as the customer of testing and emphasizing the test-first design paradigm. See also *test driven development*.

alpha testing: Simulated or actual operational testing by potential users/customers or an independent test team at the developers' site, but outside the development organization. Alpha testing is often employed for off-the-shelf software as a form of internal acceptance testing.

arc testing: See *branch testing*.

back-to-back testing: Testing in which two or more variants of a component or system are executed with the same inputs, the outputs compared, and analyzed in cases of discrepancies. [IEEE 610]

beta testing: Operational testing by potential and/or existing users/customers at an external site not otherwise involved with the developers, to determine whether or not a component or system satisfies the user/customer needs and fits within the business processes. Beta testing is often employed as a form of external acceptance testing for off-the-shelf software in order to acquire feedback from the market.

big-bang testing: A type of integration testing in which software elements, hardware elements, or both are combined all at once into a component or an overall system, rather than in stages. [After IEEE 610] See also *integration testing*.

black-box testing: Testing, either functional or non-functional, without reference to the internal structure of the component or system.

bottom-up testing: An incremental approach to integration testing where the lowest level components are tested first, and then used to facilitate the testing of higher level components. This process is repeated until the component at the top of the hierarchy is tested. See also *integration testing*.

boundary value testing: See *boundary value analysis*.
branch condition combination testing: See *multiple condition testing*.

branch testing: A white box test design technique in which test cases are designed to execute branches.

business process-based testing: An approach to testing in which test cases are designed based on descriptions and/or knowledge of business processes.

code-based testing: See *white box testing*.

compatibility testing: See *interoperability testing*.

complete testing: See *exhaustive testing*.

compliance testing: The process of testing to determine the compliance of the component or system.

component integration testing: Testing performed to expose defects in the interfaces and interaction between integrated components.

component testing: The testing of individual software components. [After IEEE 610]

concurrency testing: Testing to determine how the occurrence of two or more activities within the same interval of time, achieved either by interleaving the activities or by simultaneous execution, is handled by the component or system. [After IEEE 610]

condition combination testing: See *multiple condition testing*.

condition determination testing: A white box test design technique in which test cases are designed to execute single condition outcomes that independently affect a decision outcome.

condition testing: A white box test design technique in which test cases are designed to execute condition outcomes.

configuration testing: See *portability testing*.

confirmation testing: See *re-testing*.

conformance testing: See *compliance testing*.

conversion testing: Testing of software used to convert data from existing systems for use in replacement systems.

data driven testing: A scripting technique that stores test input and expected results in a table or spreadsheet, so that a single control script can execute all of the tests in the table. Data driven testing is often used to support the application of test execution tools such as capture/playback tools. [Fewster and Graham] See also *keyword driven testing*.

data flow testing: A white box test design technique in which test cases are designed to execute definition and use pairs of variables.

data integrity testing: See *database integrity testing*.

database integrity testing: Testing the methods and processes used to access and manage the data(base), to ensure access methods, processes and data rules function as expected and that during access to the database, data is not corrupted or unexpectedly deleted, updated or created.

decision condition testing: A white box test design technique in which test cases are designed to execute condition outcomes and decision outcomes.

decision testing: A white box test design technique in which test cases are designed to execute decision outcomes.

design-based testing: An approach to testing in which test cases are designed based on the architecture and/or detailed design of a component or system (e.g. tests of interfaces between components or systems).

development testing: Formal or informal testing conducted during the implementation of a component or system, usually in the development environment by developers. [After IEEE 610]

dirty testing: See *negative testing*.

documentation testing: Testing the quality of the documentation, e.g. user guide or installation guide.

dynamic testing: Testing that involves the execution of the software of a component or system.

efficiency testing: The process of testing to determine the efficiency of a software product.

elementary comparison testing: A black box test design techniques in which test cases are designed to execute combinations of inputs using the concept of condition determination coverage. [TMap]

exhaustive testing: A test approach in which the test suite comprises all combinations of input values and preconditions.

exploratory testing: An informal test design technique where the tester actively controls the design of the tests as those tests are performed and uses information gained while testing to design new and better tests. [After Bach]

field testing: See *beta testing*.

finite state testing: See *state transition testing*.

functional testing: Testing based on an analysis of the specification of the functionality of a

component or system. See also *black box testing*.

functionality testing: The process of testing to determine the functionality of a software product.

glass box testing: See *white box testing*.

incremental testing: Testing where components or systems are integrated and tested one or some at a time, until all the components or systems are integrated and tested.

installability testing: The process of testing the installability of a software product. See also *portability testing*.

integration testing: Testing performed to expose defects in the interfaces and in the interactions between integrated components or systems. See also *component integration testing, system integration testing*.

interface testing: An integration test type that is concerned with testing the interfaces between components or systems.

interoperability testing: The process of testing to determine the interoperability of a software product. See also *functionality testing*.

invalid testing: Testing using input values that should be rejected by the component or system. See also *error tolerance*.

isolation testing: Testing of individual components in isolation from surrounding components, with surrounding components being simulated by stubs and drivers, if needed.

keyword driven testing: A scripting technique that uses data files to contain not only test data and expected results, but also keywords related to the application being tested. The keywords are interpreted by special supporting scripts that are called by the control script for the test. See also *data driven testing*.

link testing: See *component integration testing*.

load testing: A test type concerned with measuring the behavior of a component or system with increasing load, e.g. number of parallel users and/or numbers of transactions to determine what load can be handled by the component or system. See also *stress testing*.

logic-coverage testing: See *white box testing*. [Myers]

logic-driven testing: See *white box testing*.

maintenance testing: Testing the changes to an operational system or the impact of a changed environment to an operational system.

maintainability testing: The process of testing to determine the maintainability of a software product.

migration testing: See *conversion testing*.

module testing: See *component testing*.

multiple condition testing: A white box test design technique in which test cases are designed to execute combinations of single condition outcomes (within one statement).

mutation testing: See *back-to-back testing*.

N-switch testing: A form of state transition testing in which test cases are designed to execute all valid sequences of N+1 transitions. [Chow] See also *state transition testing*.

negative testing: Tests aimed at showing that a component or system does not work. Negative testing is related to the testers' attitude rather than a specific test approach or test design technique, e.g. testing with invalid input values or exceptions. [After Beizer].

non-functional testing: Testing the attributes of a component or system that do not relate to functionality, e.g. reliability, efficiency, usability, maintainability and portability.

operational profile testing: Statistical testing using a model of system operations (short duration tasks) and their probability of typical use. [Musa]

operational testing: Testing conducted to evaluate a component or system in its operational environment. [IEEE 610]

pair testing: Two persons, e.g. two testers, a developer and a tester, or an end-user and a tester, working together to find defects. Typically, they share one computer and trade control of it while testing.

partition testing: See *equivalence partitioning*. [Beizer]

path testing: A white box test design technique in which test cases are designed to execute paths.

performance testing: The process of testing to determine the performance of a software product. See also *efficiency testing*.

portability testing: The process of testing to determine the portability of a software product.

program testing: See *component testing*.

random testing: A black box test design technique where test cases are selected, possibly using a pseudo-random generation algorithm, to match an operational profile. This technique can be used for testing non-functional attributes such as reliability and performance.

recoverability testing: The process of testing to determine the recoverability of a software product. See also *reliability testing*.

recovery testing: See *recoverability testing*.

regression testing: Testing of a previously tested program following modification to ensure that defects have not been introduced or uncovered in unchanged areas of the software, as a result of the changes made. It is performed when the software or its environment is changed.

regulation testing: See *compliance testing.*

reliability testing: The process of testing to determine the reliability of a software product.

requirements-based testing: An approach to testing in which test cases are designed based on test objectives and test conditions derived from requirements, e.g. tests that exercise specific functions or probe non-functional attributes such as reliability or usability.

resource utilization testing: The process of testing to determine the resource-utilization of a software product. See also *efficiency testing*.

re-testing: Testing that runs test cases that failed the last time they were run, in order to verify the success of corrective actions.

risk-based testing: Testing oriented towards exploring and providing information about product risks. [After Gerrard]

robustness testing: Testing to determine the robustness of the software product.

safety testing: Testing to determine the safety of a software product.

sanity test: See *smoke test*.

scalability testing: Testing to determine the scalability of the software product.

scenario testing: See *use case testing*.

security testing: Testing to determine the security of the software product. See also *functionality testing.*

serviceability testing: See *maintainability testing*.

site acceptance testing: Acceptance testing by users/customers at their site, to determine whether or not a component or system satisfies the user/customer needs and fits within the business processes, normally including hardware as well as software.

smoke test: A subset of all defined/planned test cases that cover the main functionality of a component or system, to ascertaining that the most crucial functions of a program work, but not bothering with finer details. A daily build and smoke test is among industry best

practices. See also *intake test*.

specification-based testing: See *black box testing*.

standards testing: See *compliance testing*.

state transition testing: A black box test design technique in which test cases are designed to execute valid and invalid state transitions. See also *N-switch testing*.

statement testing: A white box test design technique in which test cases are designed to execute statements.

static testing: Testing of a component or system at specification or implementation level without execution of that software, e.g. reviews or static code analysis.

statistical testing: A test design technique in which a model of the statistical distribution of the input is used to construct representative test cases. See also *operational profile testing*.

storage testing: See *resource utilization testing*.

stress testing: Testing conducted to evaluate a system or component at or beyond the limits of its specified requirements. [IEEE 610] See also *load testing*.

structural testing: See *white box testing*.

syntax testing: A black box test design technique in which test cases are designed based upon the definition of the input domain and/or output domain.

system integration testing: Testing the integration of systems and packages; testing interfaces to external organizations (e.g. Electronic Data Interchange, Internet).

system testing: The process of testing an integrated system to verify that it meets specified requirements. [Hetzel]

top-down testing: An incremental approach to integration testing where the component at the top of the component hierarchy is tested first, with lower level components being simulated by stubs. Tested components are then used to test lower level components. The process is repeated until the lowest level components have been tested. See also *integration testing.*
traceability: The ability to identify related items in documentation and software, such as requirements with associated tests. See also horizontal traceability, vertical traceability.

unit testing: See *component testing*.

usability testing: Testing to determine the extent to which the software product is understood, easy to learn, easy to operate and attractive to the users under specified conditions. [After ISO 9126]

use case testing: A black box test design technique in which test cases are designed to

execute user scenarios.

user acceptance testing: See *acceptance testing*.

user scenario testing: See *use case testing*.

volume testing: Testing where the system is subjected to large volumes of data. See also *resource-utilization testing*.

white-box testing: Testing based on an analysis of the internal structure of the component or system.

References:

1. Boris Beizer, *Software testing techniques*. Van Nostrand Reinhold Company, New York. 1990. ISBN:0-442-24592-0

2. BIG-SF. *Government Web Application Integrity.* The Business Internet Group of San Francisco, 2003. http://www.tealeaf.com/news/press_releases/2003/0611.asp

3. BIG-SF. *Black Friday Report on Web Application Integrity.* The Business Internet Group of San Francisco, 2003. http://www.tealeaf.com/news/press_releases/2003/0203.asp

4. Carlo Bellettini, Alessandro Marchetto, and Andrea Trentini, Web technologies and applications (WTA): TestUml: user-metrics driven web applications testing, *Proceedings of the 2005 ACM symposium on Applied computing SAC '05.* March 2005

5. Carlo Bellettini, Alessandro Marchetto, and Andrea Trentini, WebUml: Reverse Engineering of Web applications. *19th Annual ACM Symposium on Applied computing. Web Technologies and Applications track(Sac 2004), Nicosia, Cyprus.* March 2004

6. Wen-Kui Chang, Shing-Kai Hon, and C. C. William Chu, A systematic framework for evaluating hyperlink validity in Web environments, *Quality Software, 2003. Proceedings. Third International Conference* on 6-7 Nov. 2003 Page(s):178 - 185

7. Wen-Kui Chang, Shing-Kai Hon, Assessing the quality of Web-based applications via navigational structures, *IEEE Multimedia, Volume 9, Issue 3,* Jul-Sep 2002

8. Jim Conallen, Modeling web application architectures with UML, *Communications of the ACM, Volume 42 Issue 10,* October, 1999

9. S. Elbaum, S. Karre, and G. Rothermel, Improving web application testing with user session data, *Software Engineering, 2003. Proceedings. 25th International Conference* on 3-10 May 2003. Page(s):49 - 59

10. S. Elbaum, G. Rothermel, S. Karre, and M. Fisher II, Leveraging user-session data to support Web application testing, *Software Engineering, IEEE Transactions on Volume 31, Issue 3,* March 2005. Page(s):187 – 202

11. David Harel, Statecharts: A visual formalism for complex systems, *Science of Computer Programming,* Volume 8, issue 3 (June 1987) Pages: 231 – 274

12. E. Hieatt and R. Mee, Going faster: testing the Web application, *Software, IEEE Volume 19, Issue 2,* March-April 2002. Page(s):60 - 65

13. IEEE Computer Society, IEEE Standard for Software Verification and Validation, *IEEE Standards,* 3 Park Avenue, New York, NY 10016-5997, USA. 8 June 2005.

14. C. Kallepalli and J. Tian, Measuring and modeling usage and reliability for statistical Web testing, *Software Engineering, IEEE Transactions on Volume 27, Issue 11,* Nov. 2001. Page(s):1023 - 1036

15. C. Kallepalli and J. Tian, Usage Measurement for Statistical Web Testing and Reliability Analysis, *Seventh International softeare Metrrics Symposium (METRICS'01)* p.148

16. D.C. Kung, An agent-based framework for testing Web applications, *Computer Software and Applications Conference, 2004. COMPSAC 2004. Proceedings of the 28th Annual International Volume 2*, 2004. Page(s):174 - 177 vol.2

17. D.C. Kung, Chien-Hung Liu, and Pei Hsia, An object-oriented web test model for testing Web applications, *Quality Software, 2000. Proceedings. First Asia-Pacific Conference* on 30-31 Oct. 2000 Page(s):111 - 120

18. D. Lee, K. Sabnani, D. M. Kristol, and S. Paul. Conformance Testing of Protocols Specified as Communicating Finite State Machines - a Guided Random Walk Based Approach. *IEEE Trans. on Communications*, 1993.

19. Zhao Li and Jeff Tian, Testing the suitability of Markov chains as Web usage models, *Computer Software and Applications Conference, 2003. COMPSAC 2003. Proceedings. 27th Annual International*, Volume , Issue , 3-6 Nov. 2003 Page(s): 356 - 361

20. Chien-Hung Liu, D.C Kung, Pei Hsia, and Chih-Tung Hsu, Object-based data flow testing of web application, *The first Asia Pacific conference on quality software*, October 2000. Hong Kong, China

21. Chien-Hung Liu, D.C Kung, Pei Hsia, and Chih-Tung Hsu, Structural testing of Web applications, *Software Reliability Engineering, 2000. ISSRE 2000. Proceedings. 11th InternationalSymposium* on 8-11 Oct. 2000. Page(s):84 - 96

22. G.A. Di Lucca, A.R. Fasolino, F. Faralli, and U. De Carlini, Testing Web applications, *Software Maintenance, 2002. Proceedings. International Conference* on 3-6 Oct. 2002. Page(s):310 - 319

23. G.A. Di Lucca, and M. Di Penta, Considering browser interaction in Web application testing, *Web Site Evolution, 2003. Theme: Architecture. Proceedings. Fifth IEEE International Workshop* on 22 Sept. 2003 Page(s):74 - 81

24. S. Manley and M. Seltzer, Web Facts and Fantasy, *In proceedings of the 1997 Usenix Symposium on Internet Technologies and Systems*, Monterey, CA, 1997.

25. Q-Labs, ToolCertify User Guide, Version 5.0, 2000

26. Yu Qi, David Kung, Eric Wong, An Agent-based Testing Approach for Web Applications, *Proceedings of the 29th Annual International Computer Software and Applications Conference (COMPSAC'05)*, 2005

27. A. Rao and M. Georgeff, BDI agents: From theory to practice, *Proceedings of the First International Conference on Multi-Agent Systems (ICMAS-95)*, San Francisco, pp 312-319

28. S. Rapps and J. Weyuker, Selecting software test data using data flow information, *IEEE transactions on software engineering, Vol. SE-11, NO.4,* April 1985.

29. F. Ricca, Analysis, testing and re-structuring of Web applications, *Software Maintenance, 2004. Proceedings. 20th IEEE International Conference* on 11-14 Sept. 2004 Page(s):474 - 478

30. F. Ricca and P. Tonella, Analysis and testing of Web applications, *Software Engineering, 2001. ICSE 2001. Proceedings of the 23rd International Conference* on 12-19 May 2001 Page(s):25 - 34

31. F. Ricca and P. Tonella, Anomaly detection in Web applications: a review of already conducted case studies, *Software Maintenance and Reengineering, 2005. CSMR 2005. Ninth European Conference* on 21-23 March 2005 Page(s):385 - 394

32. F. Ricca and P. Tonella, Construction of the system dependence graph for Web application slicing, *Source Code Analysis and Manipulation, 2002. Proceedings. Second IEEE International Workshop* on 1 Oct. 2002 Page(s):123 - 132

33. F. Ricca and P. Tonella, Understanding and restructuring Web sites with ReWeb, *Multimedia, IEEE Volume 8, Issue 2,* April-June 2001 Page(s):40 - 51

34. F. Ricca and P. Tonella, Visualization of web site history, *In 2nd international workshop on web site evolution,* Zurich, Switzerland, March 2000

35. F. Ricca and P. Tonella, Web application slicing, *Software Maintenance, 2001. Proceedings. IEEE International Conference* on 7-9 Nov. 2001 Page(s):148 – 157

36. F. Ricca and P. Tonella, Web site analysis: Structure and evolution, *Software Maintenance, 2000. Proceedings. International Conference* on 11-14 Oct. 2000 Page(s):76 - 86

37. F. Ricca, P. Tonella, and I.D. Baxter, Restructuring Web applications via transformation rules, *Source Code Analysis and Manipulation, 2001. Proceedings. First IEEE International Workshop* on 10 Nov. 2001 Page(s):150 - 160

38. Sara Sprenkle, Emily Gibson, Sreedevi Sampath, Lori Pollock, Automated Relay and Failure Detection for Web Applications, *Proceedings of the 20th IEEE/ACM International Conference on Automated software engineering ASE '05.* November 2005

39. Sara Sprenkle, Emily Gibson, Sreedevi Sampath, Lori Pollock, A case study of automatically creating test suites from web application field data, *Proceedings of the 2006 workshop on Testing, analysis, and verification of web services and applications TAV-WEB* '06

40. Sara Sprenkle, Sreedevi Sampath, Emily Gibson, Lori Pollock, Amie Souter, An empirical comparison of test suite reduction techniques for user-session-based testing of Web applications, *Software Maintenance, 2005. ICSM'05. Proceedings of the 21st IEEE International Conference on* 26-29 Sept. 2005 Page(s):587 - 596

41. R. D. Tennent, *Specifying software*, Cambridge University press, ISBN 0-521-00401-2

42. Scott Tilley, Shihong Huang, Evaluating the Reverse Engineering Capabilities of Web Tools for Understanding Site Content and Structure: A Case Study, *23rd International Conference on Software Engineering (ICSE'01),* p. 0514, 2001.

43. P. Tonella and F. Ricca, Dynamic model extraction and statistical analysis of web applications, *Web Site Evolution, 2002. Proceedings. Fourth International Workshop* on 2 Oct. 2002 Page(s):43 - 52

44. J. Weyuker, More experience with data flow testing, *IEEE transactions on software engineering, Vol. 19, no.9,* September 1993

45. J. A. Whittaker and M. G. Thomason, A Markov chain model for statistical software testing, *IEEE Trans. Software Eng.*, vol. 20, no. 10, pp.812-824, Oct. 1994

46. www.mach5.com

47. www.istqb.org