RP
8
C.1

B

**COMPARISON OF TECHNIQUES
FOR TESTING WEB APPLICATIONS**

By

Marie Joseph Sahyoun

A project submitted in partial fulfillment of the
requirements for the degree of Master of Science
in Computer Science

Department of Computer Science
LEBANESE AMERICAN UNIVERSITY
2005

BY 90592.

COMPARISON OF TECHNIQUES FOR TESTING WEB APPLICATIONS

Marie Joseph Sahyoun

PROJECT

Submitted in partial fulfillment of the requirement of the degree of Master of
Science in Computer Science at the Lebanese American University
Beirut, Lebanon
Jul. 2005

Dr. Nash'at Mansour

Associate Professor of Computer Science
Lebanese American University

Dr. May Hamdan

Assistant Professor of Mathematics
Lebanese American University

Abstract

Quality assurance is an important aspect in web applications. Testing is a basic element of quality assurance; however, due to the complexity of web applications, testing these applications becomes more difficult than testing usual software applications. A few techniques and tools have been developed for testing web applications. In this work, we present a comparison of these techniques. The comparison is based on the models used, their fault detection capability and computational cost.

Table of Contents

1. Introduction	6
1.1. Web Applications	6
1.2. Software Testing	7
1.3. Web Application Testing	8
1.4. Project Objectives and Scope	10
1.5. Report Organization	11
2. Web Applications: Quality and Test Models	12
2.1. Quality Dimensions	12
2.2. Modeling Test Models: Fault types.....	13
3. Web Application Testing Techniques.....	16
3.1. Using Formal Specifications for Web Application Testing	16
3.2. Database Testing for Web Applications using AGENDA.....	17
3.3. Concentrating on the Event Driven Nature of Web Applications for Testing	19
3.3.1. Fault Types	19
3.3.2. Web Application Dependences	19
3.3.3. System Dependence Graphs.....	20
3.3.4. Testing Techniques.....	20
3.4. UML Representation of Web Applications for Static Analysis	21
3.4.1. UML Model	21
3.4.2. Static Validation.....	22
3.4.3. Dynamic Analyses	22
3.4.4. Test Case Generation.....	22
3.4.5. Tools	23
3.4.6. Experimental Analysis.....	24
3.5. Web Architecture and Application Testing.....	24
3.5.1. Tools	24
3.5.2. Web Architecture Testing	24
3.5.3. Web Application Testing.....	25
3.5.4. Experimental Analysis.....	25
3.6. Testing for the Dynamic Nature of Web Applications.....	26
3.6.1. Analysis Model	26
3.6.2. Testing	27
3.7. Modeling Web Application Testing with Object Orientation	27
3.7.1. Object Oriented Model	27
3.7.2. Objects and their Relationships.....	27
3.7.3. Behaviors.....	28
3.7.4. Structure	28
3.7.5. Testing Techniques.....	28
3.8. Web Application Testing through an Agent Based Framework	29
3.8.1. BDI Agent Model	29
3.8.2. AOSE Framework	29
3.8.3. Test Agents for Web Applications	30
4. Experimental Results	31
4.1. Experimental Procedure.....	31
4.2. The Web Application Models	33

4.3. Fault Injection	39
4.3.1. Functional Faults	39
4.3.1.1. Functional Fault 1	39
4.3.1.2. Functional Fault 2	41
4.3.2. Page Structure Faults	43
4.3.3. Security Faults	46
4.3.4. Performance Slowdown	49
4.3.5. Database Faults	50
4.3.6. Dataflow Errors	52
4.3.7. Event Flow Faults	55
4.3.8. Architectural Errors	58
4.3.9. Navigation Errors	58
5. Comparison of Testing Techniques	62
5.1. Discussion of Results	62
5.1.1. Graph Based Techniques	63
5.1.2. Non Graph Based Techniques	64
5.2. Computational Cost	64
5.3. Summary	65
6. Conclusion	67
Appendix A	72

CHAPTER I

INTRODUCTION

1.1. Web Applications

Almost any person in this day and age would be able to give some kind of definition for what a Web Page or Web Site is, even with limited use of the WWW. However, can the same be said about a Web Application? What's the difference between a Web Page, a Web Site, and a Web Application? First and foremost, we will define a web page. Throughout this paper, when we use the term Web Page, we are referring to "The information available and displayed by a Web browser as the result of opening a local file or opening a location (URL). The contents and format of the Web page are specified using HTML." (Internet and Web Glossary, 2000). With that in place, we must introduce the ideas of static and dynamic content in a web page. Static content is just what the name suggests, information (such as text, graphics, charts ...) that is part of the page and can by no means change "on the fly", i.e. the content is placed at the time of development, and can only be changed through hard coding. An example of static content is a company logo that is placed on a web page, nothing about this logo can be changed unless the source code of the page is altered. If a web page is constructed of purely static content, we can say it is a static web page. In contrast, dynamic content consists of information that can change "on the fly", possibly depending on user input, certain system factors, possibly even through somewhat intelligent means. For example, a Math exercises web page may have a picture of a cartoon face, which changes expression according to a user's answers to some Math problems, for example turning into a frown if the answer is incorrect, and changing to a smile otherwise. A web page that has some dynamic content can be considered a dynamic page. With those definitions in place, a web site can be defined as "The entire collection of web pages and other information (such as images, sound, and video files, etc.) that are made available through what appears to users as a single web server. Typically all of the pages in a web site share the same basic URL." (Glossary of Internet Terms, 2004), and of course, a Web Site with any dynamic content can be considered a dynamic site. The introduction of dynamics into web pages has not only increased the functionality of the WWW, but has also increased the complexity of the sites themselves. The early days of the WWW saw large amounts of web sites that contained purely static content consisting of mostly HTML statements, text,

images, and possibly a simple sort of animation. For the most part, these pages were utilized as centralized repositories of information. Dynamic content began to make an appearance and along with that a barrage of scripting languages that facilitated the coding required for that. The scripting languages were close to their standalone language counterparts, but had not the same complexity. Finally, in this era, the Web Application has made its appearance with full fledged programming languages, complex database systems and more savvy and experienced coders. The Web Application can be defined as "an application delivered to users from a web server over a network such as the World Wide Web or an intranet" (Web Application, 2005). Some well-known web applications include e-Bay and Amazon. The structure of a Web Application generally consists of a three tier architecture: the client, server and data store (Mansour and Hourri, 2005). It's obvious that this increase in the functionality of the simple web page will also lead to an increase in complexity, and of course, this suggests an increased level of difficulty in ensuring that the application actually works, and works well!

1.2. Software Testing

Testing is an imperative part of the software development process - arguably it is the most important aspect of the cycle. For the sake of formality, we will define software testing as "a process used to identify the correctness, completeness and quality of developed computer software." (Software Testing, 2005). Note that this is applicable to both standalone software applications and Web Applications, although the testing methods do differ, sometimes significantly. At a higher level, all testing methods can be divided into two major categories (Dyck and Sloane, 1999):

- White box testing: finds errors in the internal structure of the program.
- Black box testing: test how well a program matches the customer specifications.

Examples of specific white box testing techniques include: statement testing, branch testing, path testing, predicate testing, dataflow testing, structured testing and domain testing (Beizer, 1990, Mansour and Salame, 2004, Offutt and al., 1999, Rapps and Weyuker, 1985). For black box testing, the following techniques have been suggested: equivalence partitioning, boundary value analysis, random testing and functional analysis-based testing (Beizer, 1990, Howden, 1986).

In addition, there are generally five levels of software testing, outlined in the following:

- Unit testing: tests one unit of the entire software in isolation from the remainder of the system. Examples of units within software include functions, or even classes.
- Integration testing: tests multiple and related units within a system, i.e. units that together create modules, subsystems, or the system itself.
- Product: tests that the system (as a whole) properly implements certain functionalities. This is also known as alpha testing.
- System: similar to Product testing but is performed on platforms that very closely resembles actual user's systems, with respect to aspects such as hardware, database size, etc... This testing may also be termed alpha testing.
- Acceptance: tests performed by a series of end users (i.e. not by developers and not by system testers) to test the system as a whole, in completion. This is also known as beta testing.

1.3. Web Application Testing

Although Web Application testing techniques may have some similarities to the testing techniques utilized for standalone applications, it could be a grave mistake to assume they are identical. First of all, we can define Web Application testing as the process of "uncovering errors in Web Application content, function, usability, navigability, performance, capacity and security.(Pressman, 2005)" The architecture of a Web Application differs from the standalone application – it's a three tiered architecture consisting of the server, client and a repository for data. This alone can account for some of the difficulty that arises when testing a Web Application, for example, errors in a web application may only be side effects of the actual problem since an error may occur server side while the manifestation of that error occurs client side. Other problems include the fact that testing is done in a different environment than that of actual usage, and that the testing environment is generally static with respect to configurations, while the actual environment is dynamic. In addition, errors can be attributed to the web application configuration, and not always to code. All of these problems are magnified in difficulty of tracing when it comes to the client/server architecture and network channels (Pressman, 2005). The evolution of a web application is generally very rapid, and may require large amounts of modifications in short periods of time (Ricca and Tonella, 2001). For example, in (Sneed, 2004), the

author uses a web application system for the Austrian Chamber of Commerce as an experimental test bed. That system has, on average, a requirement of 18 changes per month – such a figure is virtually unheard of in typical software. These factors make adequate testing necessary, since finding and correcting faults after deployment can be extremely costly, both in terms of time, money and company/organization reputation. In fact, with respect to the latter, it's been shown that there is no real loyalty to specific websites. So if one company puts up a web application with a certain functionality, and another company puts up a web application with the same functionality at a later time, the users of the first application will generally migrate to using the latter if it seems to be a better application – again, this is not the case with standalone software (Wu and Offutt, 2002). So ensuring that a web application is reliable is imperative from the business perspective. Other interesting points brought up in (Wu and Offutt, 2002) are that a web application must be up and running 24 hours a day, web applications must be easily maintained since there is no selling and distribution required after updates have been made and they must be easily scalable since the number of users could easily increase exponentially. Moreover, scripted implementations according to (Bedi and Schroeder, 2004) are strong enough to implement sophisticated applications as long as the software management is properly designed.

Xiong and Probert applied multiple test methods such as grey box, black box and white box (Xiong and Probert, 2003). They also used TTCN-3 to integrate specification of test cases, which is the third generation of the TTCN standard (Tree and Tabular Combined Notation). TTCN-3, is a modern and flexible language for specifying many types of system tests over a wide variety of communication interfaces.

Many commercial tools have been introduced for testing web applications, such as:

- HttpUnit (HttpUnit, 2004) is a freely available black box testing API for web pages written in Java. Web application tests written using HttpUnit do not test pieces of the application code but rather query the Web server externally and examine the responses received. HttpUnit is appreciated for its price and simplicity. It adds to Java the basic mechanisms of maintaining client state, sending requests, retrieving responses, and following links. It does not offer any GUI or automated test scheduling or graphical reports. HttpUnit can be thought of as a cluster of two functionalities :

- A Web client that maintains state and provides the facility to send requests and retrieve responses.
 - A variety of methods that simplify the verification of response content.
- Load testing web applications using Microsoft's Web Application Stress Tool (Microsoft, 2000) can simulate large numbers of users against a Web application, and can also make intelligent decisions about hardware and software load and how much traffic a given machine or group of machines can handle. This tool will then generate a performance data as an output.
 - Puffin testing framework (Puffin, 2002) is an open source framework for testing web applications. It helps testers in building dynamically driven regression tests for complicated web sites.
 - Web Application Testing (WAPT, 2004) is a load and stress testing tool that provides testers with an easy to use and cost-effective way of testing web sites and analyzing its performance characteristics under various load conditions by automating the testing phase of the development process. It then provides test results in form of descriptive graphs and reports.

1.4. Project Objectives and Scope

The objective of this project is to study the most common techniques used to test web sites. We will first summarize the papers that introduced them, then, through an experimental procedure, we will inject nine types of faults into an existing website and evaluate the efficiency of each technique in finding these errors. Errors in a web site are classified in nine categories. They can be functional, structural, security related, performance, database, data flow, event flow, architectural or navigational. The test website will be injected with nine faults each one representing a typical error in its category. We will then perform an analysis of the results to see which of the techniques is more appropriate for finding which type of errors. Finally, we will conclude with some recommendations and guidelines for further work on this topic.

1.5. Report Organization

This project is about comparison of techniques for testing web applications. Chapter II consists of a literature survey and Chapter III of an experimental analysis. The literature survey discusses a number of previously developed web application testing and analysis techniques. Each section is dedicated to a particular technique, with a description and a summary of the method itself and

its testing prowess if the information is available. In our experimental results, Chapter IV, five of the testing techniques discussed in the literature survey are tested against a live website www.sjsalumni.org. Chapter V consists of a comparison of testing techniques as stated in previous chapter with some sort of assessment of each technique. Chapter V consists of the conclusion and further related work.

CHAPTER II

WEB APPLICATIONS: QUALITY AND TEST MODEL

2.1. Quality Dimensions

The quality of a Web Application can be measured by a number of characteristics (Pressman, 2005, Splaine and Jaskiel, 2001). These characteristics are listed and briefly described in the following:

- **Content:** informational components. A high quality application would have both syntactically and semantically correct information.
- **Function:** what the application is meant to do. Application functionality should be as close to client specifications as possible.
- **Structure:** the code, it's organization, architecture, ... A well designed application should be easily modified and extended, and this is especially important in a web application since requirements may be quickly changed.
- **Usability:** how intuitive it is to use an application, from the perspective of a diverse set of people. Again, this is an imperative feature in a web application, since it is accessible at a world wide scale, and has virtually no age or gender barriers.
- **Navigability:** the ability to move from page to page via links in a web application. Large web applications are especially at risk of having problems with navigability, as the number of links between pages increases.
- **Performance:** how well the application operates under different systems, with varying user loads, etc... A well performing web application is a difficult goal to achieve and to test, due to amount of possible systems that may eventually view/use the application.
- **Compatibility:** the application's compliance with various clients and servers.
- **Interoperability:** how well the application can be integrated with other applications and databases.
- **Security:** whether the application is vulnerable to malicious attacks.

2.2. Test Models: Fault types

In Pressman (2005), a plan of action is suggested for web application testing that is a somewhat customized and tweaked version of the testing done for typical software. This testing includes: content testing, interface testing, navigation testing, component testing, integration testing, configuration testing, security testing and performance testing. These are briefly described in the following.

Content testing ultimately ensures that a Web Application's content is correct, from both syntactic and semantic perspectives. Therefore, checking such issues like spelling and grammar are equally important to content testing as are accuracy and completeness of information. Interface testing checks that a Web Application's user interface is well designed, user friendly and has no logically incoherent interaction mechanisms. Navigation testing attempts to ensure that problems such as dead links, incorrect transfers or unnecessary links are not found a Web Application. In a Web Application, a "unit" can be considered a Web Page – consisting of a logical chunk of content and functionality. Component testing ensures that each of these units is properly functioning according to its role in the given application. Both component and navigation testing can be together considered a part of integration testing that occurs as the application is gradually developed. Configuration testing ensures that errors that are specific to certain computing environments (both server and client side) are uncovered. The computing environment includes operating systems, browsers, hardware platforms and communication protocols. Security testing attempts to ensure that the Web Application is not vulnerable to either intentional or non-intentional security breaches. Ironically, the goals of such tests are to show that security breaches are feasible. Performance testing measures a number of characteristics of a Web Application's performance. Some aspects include the application's response time in the face of varying amounts of network traffic, the location of components that are highly susceptible to performance degradation, and how important the performance of an application is in regards to its overall objectives.

In this paper, we discuss a number of web application testing approaches that address the following faults:

1. Functional: A functional fault occurs if the web application does not properly perform some function that it should perform, i.e. does not exactly meet client specifications with respect to some aspect of its functionality.
2. Page Structure: A page structure fault occurs if some component that should be on a page is not, or if the component is improperly placed or

configured. For example, a credit card application page that does not contain a form for client information, although the form should be there.

3. Security: A security fault occurs if an unauthenticated user can enter some part of the application that requires authentication.
4. Performance: A performance fault occurs if a web application buckles under certain pressures such as large amount of users accessing application at once, or large loads of data being sent.
5. Database: A database fault occurs if after some data transaction, the records in the database are not properly updated, i.e. there is some inconsistencies in data. Another alternative for a database fault is that queries are incorrect, or inaccurate. For example, consider an online shopping application in which users may have specialized and personalized accounts that include the user's personal information. If a user attempts to change his/her address, and this does not take effect in the database containing the information, then this is a database fault.
6. Data Flow: A data flow fault occurs if there is some problem in the transfer of data within a web application, whether that be in the declaration of the variables that hold the data, or in the loss or change of information between pages. For example, consider a text field that can be filled in one page with a user's first name. If, upon entering the data and clicking to move to the proceeding page, the data is lost and becomes null, this is a data flow fault.
7. Event Flow: An event flow fault occurs if an event fires and the action that the event should trigger doesn't happen. For example, consider a page in which the clicking on a button should move the user to another page. If the button is clicked and nothing happens, this is an event flow fault (as long as the problem is NOT because the next page is nonexistent, or is down - which would become a navigation fault).
8. Architecture: An architectural fault occurs if a web application and the environment that application is in are not compatible. For example, consider a .NET web application that is perfectly functional on the test server, but is completely useless on the deployment server, because the latter does not have the .NET framework installed.
9. Navigation Behavior: A navigation behavior fault occurs if the transferal of pages, from one to another does not act as required. For example, a user clicks on the "Next" button of page₁, in an attempt to reach page₂, but reaches page₃ instead.

The faults are summarized in the Table 1.

Table 1: Faults addressed in web application testing approaches

Functional	Web application does not properly perform some function that it should perform
Page Structure	When a component is improperly placed or configured
Security	Unauthenticated user entering part of the application that requires authentication
Performance	Web application buckles under certain pressures such as large amount of users accessing the application at once
Database	Inconsistencies in data: Records are not properly updated, Incorrect or inaccurate queries
Data Flow	Problem in transfer of data within a web application
Event Flow	Event fires and the action that the event should trigger doesn't happen
Architecture	Web application and the environment that application is in are not compatible
Navigation Behavior	Transferral of pages, from one to another does not act as required

CHAPTER III

WEB TESTING TECHNIQUES

3.1. Using Formal Specifications for Web Application Testing

In (Jia and Liu, 2001), Jia and Liu propose an approach for testing web applications using a formal specification language (for specifying test cases and results) implemented in XML syntax and a test engine that automates the actual running, result gathering and reporting of the application. More specifically, their proposed XML files contain templates of test cases which provide input and expected output of each case. These files are then input into the test engine, which generates and runs each test case according to the suggested input. The actual output is compared to the expected output specified in the formal specifications. Finally, a report is generated to summarize the outcome of the tests.

Before going into more detail about Jia and Liu's work, we will discuss what the authors suggested was the main testing functionality of their concepts. They state that their approach can test the following in a web application: functionality, page structures, security and performance.

Formally defining test cases first requires that there is some language that facilitates the definitions. The authors defined a syntax for their language. With the language in place, the test specification can be defined with three primary components:

- Test Suite: contains test steps for a certain web application features that must be tested (i.e. functionality, security and performance). A specification can have one or more test suites.
- Test Case: tests a specific aspect of a web application's features, i.e. functionality security and performance. Each case is made up of series of test sequences, where each sequence is made up of a series of test steps.
- Test Step: specifies one step in the interactions that make up a web application, where one step includes a pair of request and response specifications.

The request and response specifications are defined as follows:

- a. Request: a series of HTTP requests containing the URL of the request, a header list (if applicable), a parameter list (if applicable), and variable declarations (if applicable).
- b. Response: the assertions on the HTTP request in one test step containing a header list (if applicable), a list of predicates (which are defined later in this section), and variable declarations (if applicable).

Predicates specify constraints on the test results, there are primarily two types: simple and compound. Each of these can also be divided, so that simple predicates check for result matching, result subsets and comparisons (e.g. greater than, greater than or equal, etc...)

And compound predicates can be used for forming logical statements (using operators such as and, or, etc...).

Using these components, a test specification can be represented as a tree of test steps, where any path from the root can be considered a test sequence. Each node in a path represents a test step where parent nodes must be executed prior to their child nodes, and sibling nodes represent alternative test steps within one sequence.

The authors developed the tool WebTest, to test their approach. The tool is comprised of the input XML files (recall that these contain the formal specifications of the test cases), the Test Generator, the Test Runner (which together form the Test Engine) and the Test Report. The Test Generator was developed in Java – its role is to read and parse the XML files and generate testing source code, also known as the Test Runner. The test runner executes the tests, while the Test Report provides the results.

3.2. Database Testing for Web Applications using AGENDA

In (Deng, Frankl and Wang, 2004), the authors extend an already existing relational database testing tool set named AGENGA. Their extension is aimed at properly testing web database applications, and was implemented for applications written as Java Servlets with JDBC for database access. The authors noted that most previous web application testing approaches did not consider the effect of the database the application was using, and that this could lead to potentially weak applications. Therefore, their approach deviates from the standard black box testing methodology that had been widespread among web application testing developers, by taking a white box approach that can provide:

- a. More thorough coverage of data aspects that are not usually found by conventional web application testing techniques.
- b. Better fit form input values.
- c. Better test pruning, by removing tests that seem to be unnecessary.

In a brief step by step analysis, the author's method for testing web database applications is therefore primarily based on the following five steps:

- a. From the application source code, URL links and URL inputs are found. The first are divided into two categories – either static page or data-based pages. Note that the first refers to a page that contains static content, and the latter contains content dynamic content that requires some sort of data-store, in this case we assume that store is a database.
- b. Using the URL and linkage information, a graph is constructed such that each node represents a URL and each edge represents a link between URLs.
- c. Each path in the graph is considered at least one test case, and some of the paths are selected.
- d. Inputs for each URL are generated by AGENDA. An XML file is created for representing each test case, where a test case is made up of path and the appropriate inputs.
- e. Each test case is run, and AGENDA is used to compare expected and actual outputs, and the expected and actual database state.

Note that the authors made the assumption that URLs are hard coded into a web application.

AGENDA takes as input a database schema, the source code of an application and a number of files containing suggested attribute values – or data groups – since the data are categorized according to some logical partition. These data are ultimately used as database data to be used as test input for the application being tested. AGENDA also requires that a user specify a number of heuristics, and using these and the input data, a finite number of abstract test cases are generated, and the user can specify the expected output of each test case, in addition to the expected state of the database. Finally, each test case is actually run by AGENDA, and the actual output is compared to the expected output, as is the database state.

The authors tested this database web application testing approach on the TPC-W benchmark – a web application that models an online bookstore. Although their

analysis does not measure how well their method works as a testing approach, it concludes that the method does not require great amounts of overhead for test case generation, XML file generation or test case execution.

3.3. Concentrating on the Event Driven Nature of Web Applications for Testing

Mansour and Hourri (2005) suggest web application testing methods based on dependence graphs. Their work places heavy emphasis on the fact that web applications are largely event driven, i.e. a web page can cause events that will eventually be taken care of by server side code.

3.3.1. Fault Types

The authors state that web application faults can basically be classified into four types. These types are listed in the following:

- Navigation Faults: related to the linkage of web pages
- Interface Faults: these have to do with how the user can interact with an application and what the user is exposed to through their senses, mainly through the "user interface"
- Integration Faults: these are related to the "bits and pieces" that, when working together, make up an application (e.g. database and application, presentation file and code behind file)
- "Traditional" faults: the problems that can be found in any type of computer application (e.g. logical errors, syntax problems)

3.3.2. Web Application Dependences

The authors go through a clear analysis of the types of dependences that can be found in web applications. These are again listed and described in the following:

1. Call Dependences: these can be divided into two categories.
 - a. Internal call dependences: when a method and calling statement are in the same class or presentation file
 - b. Inheritance call dependences: when data definitions are done in a presentation file, but they are used in the code behind files
2. Semantic dependences: these are dependences that occur when one object holds information about another, for example what plain text gives to a web page.
3. Event based dependences: these can be further divided into three categories.

- a. Link dependences: when one page calls another through some kind of an event, but doesn't modify the called page
- b. Visible effect dependences: when one page causes a visible change to occur in another page
- c. Invisible effect dependences: when one page causes an invisible change to occur in another page

3.3.3. System Dependence Graphs

Mansour and Houri go on to discuss the construction of System Dependence Graphs while taking into consideration these dependences. Such a graph gives a complete overview of a web application. They also introduce the notion of an Event-Based Dependence Graph, which, as the name suggests, is a graphical representation of the event dependences within a web application.

3.3.4. Testing Techniques

Mansour and Houri then thoroughly describe the types of testing that would be beneficial to web applications. These testing methods are listed and described in the following:

- Data Flow Testing: This can further be divided into three levels – function level, function cluster level and object level. The authors suggest that data dependence graphs be used for function level testing, while call dependence graphs be used for function cluster and object level testing.
- Event Flow Testing: Event flow testing has two categories of testing levels – retrieving and updating. Retrieve level testing is based on link dependences and update level testing is based on visible and invisible effect dependences. In discussing event flow testing, the authors introduced a concept called the trigger effect chain, that formally has the following syntax: <Page m, Page p, (Event I, Event ii, ..., Event n)>, where Page m is a page which triggers an event, Page p is the page in which the event causes some kind of an effect, and the sequence of (Event i, ..., Even n) are the events that must occur for the effect to subsequently take place.
- Coverage Based Testing: These testing methods are primarily used for choosing test cases to test segments of a program. For web applications, the authors suggest three levels of this type of testing: All-hyperlinks testing is aimed at testing (i.e. using) all hyperlinks within an application. All-input-GUI testing is aimed at testing (again,

using) all GUI elements that can accept input. All-events testing is aimed at testing all events in an application, i.e. all trigger and effect chains.

- **Mutation Testing:** These methods evaluate how suitable a suite of test cases are for the testing of an application. The authors propose using mutation operators for the code-behind and events in a .NET web application. For these operators, they suggest separating operators into four groups, two for the code-behind files, one for the presentation file and one for events.

3.4. UML Representation of Web Applications for Static Analysis

In 2001, Ricca and Tonella propose a UML model of Web Applications as a means of aiding the analysis of static testing, i.e. testing that does not require a program to be run but analyses static components such as HTML pages, or application models. The generated UML can ultimately be used to aid in dynamic testing of web applications, i.e. those tests that do require program execution. They implemented their concepts in two tools, called ReWeb and TestWeb.

3.4.1. UML Model

The authors describe a UML based diagram for modeling a web application. Their diagram includes the following components: web pages (which are then extended to either static or dynamic), frames (i.e. segments in other pages that can be considered their own pages) and forms (which take in input that is eventually sent to a web server). The UML for a web application also has Conditional Edge relations (specifically for dynamic pages) and associations such as link (for linking pages), and link and split into (for frames).

3.4.2. Static Validation

The authors propose a number of scenarios in which static analyses, through their suggested UML, would be useful to the testing of Web Applications. Scenarios include the following:

- **Locating unreachable pages:** pages that cannot be linked to from any application page, even though they do exist and are saved on the server.
- **Locating ghost pages:** pages that are referenced by other pages but do not exist.
- **Analyzing reaching frames:** the set of pages in which a given frame can appear.

- Determining data dependences: the association of a variable definition to a node actually using the definition.
- Dominator analysis: the intermediate pages that must be traversed to get from one page to another are termed the final destination pages dominators.
- Analyzing shortest paths: locating the shortest paths to get from one page to another.

3.4.3. Dynamic Analyses

- Ricca and Tonella's dynamic testing analysis concentrated on white box testing. The testing methodologies they suggest were similar to that used in typical (i.e. not web application) software. These are listed and briefly described in the following:
- Page testing: a test case exists that allows every page in the application to be traversed.
- Hyperlink testing: every hyperlink is traversed from every possible page in at least one test.
- Definition-use testing: every variable definition, followed by every one of that variable's usage should be checked from every possible navigation path.
- All-uses testing: every variable definition, followed by every one of that variable's usage should be checked from at least one navigation path.
- All-paths testing: every possible path is traversed at least once by at least one test case.

3.4.4. Test Case Generation

Ricca and Tonella discuss the specifics behind generating test cases using their approach. They state that in general, web application testing involves choosing a set of paths from a Web Application graph as test cases, and by providing input values for those test cases. They propose automating the process of test case generation by computing the so-called path expression of a web application graph. A path expression is a way of representing the paths in a graph using algebraic notation. The variables are actually edge labels that can be combined with + to show selection statements and * to show loop structures. Take for example the expression $(e_1e_3 + e_2e_4)^*$. Its interpretation corresponds to a site in which one page is linked to two pages that are represented as e_1 and e_2 in a web application graph. These pages are each linked to pages represented by e_3 and e_4 respectively which are also linked to the original page, so they create a loop.

Thus, the path expression provides a simple but powerful way of defining potential test cases. However, when defining cases it is much easier to define them all, than to determine the bare minimum of useful cases (from the path expressions) that satisfy some criteria for testing. But it is not impossible when heuristics are introduced. The authors outline the particular heuristic they used:

```
While criterion not satisfied
  For each alternative
    Choose one alternative never considered before, if any exist
    Or randomly choose one
  If computed path increases coverage
    Add it to the resulting paths
```

The authors then suggest that upon generation of the appropriate path expressions, a tester must only input test values when necessary and the actual testing may be done automatically. Note that they do not seem to suggest an automatic method for comparing output results.

3.4.5. Tools

ReWeb and TestWeb are two tools that were implemented for the analysis and testing of web applications. In short, ReWeb generates the discussed UML model after downloading and analyzing the web pages of a Web Application. TestWeb uses the model generated by ReWeb to generate a set of test cases, and ultimately execute those cases.

3.4.6. Experimental Analysis

The experimental analysis for the authors' propositions was performed on the Wordnet and Amazon websites. Their experiments showed that ReWeb did provide an adequate UML of each web application, and that the UML proved to be useful for understanding site organization. TestWeb also provided a good coverage of test cases.

3.5. Web Architecture and Application Testing

A recent study (Sneed, 2004) suggests a Web Application scheme that consists of two major testing categories:

- Architectural testing: testing of the compatibility of the application within its environment/architecture.
- Functional testing: testing of the functionality of the application.

3.5.1. Tools

In this work, the tool Empirix was chosen for the simulations, i.e. for generating messages as if from the client, and intercepting messages as if the server, and to record tests and play them back.

3.5.2. Web Architecture Testing

The author emphasized the first category as one which may be overlooked but is equally as important as the latter. He was quick to point out that although using an elaborate software development environment will decrease the time necessary for the development of an application, it may increase the time necessary for testing. He also pointed out that reusable components should be tested at each use as if they were never tested before, since the context in which they were used may affect their functionality. He states that the following issues must be addressed when testing the application architecture:

- transactions are managed
- exceptions are caught and handled
- events are caught and addressed
- messages are properly submitted and converted
- data is stored properly, indexed properly and can be retrieved
- memory is properly allocated and de-allocated
- intermediate states can be logged
- unauthorized access is caught and stopped
- ability to handle large transaction loads

Sneed then goes on to describe two approaches to web application architecture testing: the outside-in approach and the inside-out approach. The outside-in approach is done over the user-interface, and the server must be simulated. The inside-out approach has testing start from the data-server, and in this case, clients must be simulated.

3.5.3. Web Application Testing

The author's major contribution to testing the functionality of a web application would be the development of comparison techniques for validating XML output (which were the files returned from the server to be displayed) and for validating the status of SQL databases. The results of tests were stored in XML documents to be sent by the server and displayed by the client. But before being displayed, they are intercepted for comparison against an older version of the tests – for the sake of comparison. Then, a file of assertions is created, where an assertion is a claim about "the relationship between data items". The outcomes of the tests can

be compared according to whether the data from the old and new files of results conform to the assertions. A similar method is used to validate the SQL databases. In that case, old records (to be compared against) are stored in a file, and new records are stored in an indexed table. The data in the file and in the fields are compared – here not by assertions but by whether field values are identical or missing. In both scenarios (i.e. testing XML files and SQL databases) any problems or inconsistencies are saved to a report to be reviewed by the tester.

3.5.4. Experimental Analysis

Sneed tested his propositions on a web application for the Austrian Chamber of Commerce – an application that was developed for users that may have had no previous internet experience, and has become highly complex. It also has an extremely high rate of change, with about 18 new requirements to be implemented per month. For the XML files, 14% of the total errors reported (i.e. errors found by testers and users) were caught by the XML file comparisons. This amounts to 45% of the errors reported by testers. The SQL comparisons located 9% of the total errors reported.

3.6. Testing for the Dynamic Nature of Web Applications

In 2002, Wu and Offutt describe a model that can be used to analyze the dynamics and behaviors behind web applications. The authors listed the dynamic characteristics of a web application as including program behavior that's not easily predictable, the quick changes of the application's requirements and the rapidity in which web technology is being altered, in addition to client components that are generated on the fly.

3.6.1. Analysis Model

They began the description of their model with detailing the interactions that occur in a typical web application (e.g. client receiving initial information from servers, then sequences of the client sending requests to the server which subsequently sends replies). Their model takes the form of regular expressions, with the assumption that every web application has some start page *S*, and by defining the basic elements of a web application as an atomic section, i.e. a static HTML file, or the part of a server side program that translates HTML. They then define a few so-called composition types, which combine the atomic sections of the web application into units. These compositions are:

- Basis: an atomic section.

- Sequence: a coupling of two composite sections in which one follows the other.
- Selection: consisting of two composite sections, of which only one is selected by the server.
- Aggregation: two composite sections in which one is a part of another.

Such sequences can be depicted by typical regular expression syntax – which can then be used to model server components in so-called composition rules. The interactions between client and server are modeled by another concept developed by the authors: dynamic interaction. These interactions are further divided into two categories – HTML links and user actions – these are known as transition rules:

- Link transition: invoking a link in one page causes a transition to another page from the server to the client.
- Composite transition: the execution of a component causes a page to be created and sent to the client from the server.
- Operational transition: transitions between pages caused by the user and system, e.g. using the back button to go to a previously viewed page.

With those definitions in place, Wu and Offutt model a web application as a start page, a set of composition rules for all server components and a set of transition rules.

3.6.2. Testing

The authors then go on to describe a typical test case for a web application. It would include a sequence of transitions that begins with one start page then uses composition and transition rules to reach a certain page – this is called a derivation. Studying the derivations of a web application would provide a test suite for the testing of the application.

3.7. Modeling Web Application Testing with Object Orientation

Kung et. al describe and discuss an object oriented modeling method for web application testing in (Kung, Liu, and Hsia, 2000)

3.7.1. Object Oriented Model

The authors propose a model that represents a typical web application as three aspects:

- Object aspect: represents the entities within a web application.
- Behavior aspect: represents the navigation behavior in a web application.
- Structural aspect: represents the flow of control and data in a web application.

3.7.2. Objects and their Relationships

Kung et. al propose the Object Relation Diagram (ORD) for the representation of the entities in a web application. This diagram is a directed graph where nodes represent objects and edges represent relationships between objects (where there is a predefined set of relationship types). Objects can be further divided into three categories:

- Client pages: HTML documents that can be rendered by browsers on the client side.
- Server pages: web pages that are executed at the server by the server.
- Software components: any other programs – these can interact with all other page types.

Relationship types include those that are known in Software Engineering (i.e. aggregation, association and inheritance) in addition to four new types: navigation, request, response and redirect.

3.7.3. Behaviors

The authors state that the behavior of a web application is defined by both the "state-dependent" behavior of each page and the navigational behavior between pages. Each of these behaviors can be modeled by separate diagrams: the Page-Navigation Diagram (PND) and the Object-State Diagram (OSD). The page navigation diagram can be represented as a finite state machine where each state represents a client page and transitions between states represent hyperlinks. And since the choice of navigation can occur dynamically, guard conditions are introduced to control the choice of the "next page". These conditions specify what criteria must hold in order for navigation to a certain page can occur. The object state diagram can be used to represent state-dependant behavior – it is a "set of hierarchical, concurrent and communicating" state machines. An aggregate object can be modeled using the appropriate object state diagrams to form a composite object state diagram (COSD).

3.7.4. Structure

Block Branch Diagrams (BBD) and Function Cluster Diagrams (FCD) can be used to represent the flow of control and data in a Web Application. The BBD is

constructed for each function in the application to show such details as internal control structure, variables defined and used, parameter list and any other functions that may be invoked. It is similar to a control flow graph. The FCD is a set of clusters of functions within any given object. Each cluster can be represented as a graph where each node represents individual functions and edges represent the calling relations between those functions. This is similar to the Interprocedural Control Flow Graph (ICFG) for data flow analysis across multiple functions.

3.7.5. Testing Techniques

Using the previously described model, the testing of a Web Application can be divided into three main parts: navigation behavior testing, state behavior testing and structural testing. The PND can be used to construct a navigation test tree for navigation behavior testing. This tree can be used for extracting test cases – where each test case is an initial state (the root), an end state, and a sequence of edges between the two. For state behavior testing – a test tree of can be constructed from the COSD. For this, test cases are transitions in tree paths that begin at the root and end at some node. For structural testing, the authors first describe three perspectives for test cases: intra-object, inter-object and inter-client. For the intra-object perspective, data flow test cases for individual functions can be extracted from BBDs, and test cases for multiple functions can be extracted from FCDs. For the inter-object perspective, test cases can be obtained from ICFGs. Finally, for the inter-client perspective, test cases can be obtained from BBDs that represent data flow between multiple clients.

3.8. Web Application Testing through an Agent Based Framework

In (Kung, 2004), David Kung discusses an agent based framework for the modeling of and reasoning about web application testing.

3.8.1. BDI Agent Model

The authors emphasize the use of the Belief - Desire - Intention (or BDI) model for agent modeling. Generally in this architecture, agents can include representations of their beliefs (i.e. information about their environment and other agents), desires (i.e. goals to be accomplished) and intentions (i.e. plans that must be executed to achieve desires). For web applications, these components of the agent architecture correspond to the following:

- Beliefs: web testing models, test results and knowledge of the component under test.

- Desires: test criteria.
- Intentions: test activities.

3.8.2. AOSE Framework

The AOSE framework provides abstract classes that allows for the modeling of agent-oriented systems. These classes are named: Belief, Goal, Plan, Agent, Agent Communication Act and Blackboard. It also has a number of diagrams:

- Agent Goal Diagram (AGD): defines roles of agents and shows relationships between goals and environment.
- Use Case Goal Diagram (UCGD): puts together the Use Case Diagram (UCD) from UML and the AGD to show the affects between use cases and goals.
- Agent Domain Model (ADM): similar to the System Domain Model which summarizes object classes and their relationships and attributes, the ADM incorporates agents into the typical object oriented SDM. There is also the incorporation of an agent's internal knowledge, i.e. its Beliefs, Desires and Intentions.
- Agent Sequence Diagram (ASD): shows the interactions between the Beliefs, Desires, Intentions and all other objects of a particular agent.

3.8.3. Test Agents for Web Applications

One characteristic that makes a software agent an "agent" is that it is very highly specialized. Therefore, testing agents should be specially tailored so that they can test very specific aspects of an application. The authors note that a typical web application consists of multiple heterogeneous documents. In turn, each document has more specific areas of testing, such as structural, functional and state tests. Their strategy therefore is to create test agents for each type of document, and the web application agent then will be composed of multiple heterogeneous agents with highly specialized testing capabilities. All of this can be modeled using the diagrams discussed previously.

CHAPTER IV

EXPERIMENTAL RESULTS

4.1. Experimental Procedure

In this chapter, we evaluate the effectiveness of five of the proposed testing methods by using fault injection. The subject application is www.sjsalumni.org web site. This web site is for the Saint Joseph School Alumni Association (SJS) and is 902 lines of code long; it will be referred to as SJS web application throughout the rest of this report, and its home page is given in Figure 1.

Figure 1: Saint Joseph School Alumni Association (SJS) Home Page

The site is a community website containing an e-commerce module with secure payment. To make a purchase, the user is required to sign in with a user name and a password. Once authenticated, he will be able to navigate the items, select any product he/she wishes to buy, and add them to the shopping cart. The cart

can save its content through multiple sessions, and the user can navigate at will, from and to the cart page, modifying its content when desired. The items contained in the cart can be divided into two categories:

- The products which have their prices fixed where the user can only modify the quantity.
- The donations, where the client is allowed to set the price, but not the quantity.

Our test will be focused on the shopping cart and its neighboring pages. The code for the cart is given in Appendix A.

The shopping cart of SJS is shown in Figure 2. It displays all the items or donations that the user has added to his cart. It also has the following features:

- The Continue Shopping button will take the user back to the categories section of the SJS, so he can continue his shopping.
- The Recalculate button will update the total value of the cart.
- The Check Out button will redirect the visitor to the payment web site.

Figure 2: SJS Shopping Cart Page

Given the nine fault types discussed in section 2.2, we will inject errors for every type and discuss the results in each of the five testing techniques, to see whether it does or does not capture the fault.

To avoid repetition throughout this report, we will use the acronyms listed below to refer to each of the five testing techniques that we evaluate.

Technique	Acronym
Mansour and Hourri Dependency Graphs	MHDG
UML representation for static testing	UMLT
Architectural and functional testing	AFT
Testing for dynamic nature	TDN
Object oriented web testing model	OOWT

4.2. The Web Application Models

Figures 3-5 show the graphs or models as proposed by MHDG, UMLT and OOWT applied to SJS. The MHDG's event based dependency graph for the SJS cart is given in Figure 3.

The UMLT technique proposes to represent a web site topology as a UML object diagram. Although the authors wrote two software ReWeb and TestWeb to run their test cases, we can use the algorithm they proposed to generate a diagram from the sources of the SJS web site. Figure 4 shows such a graph.

The OOWT technique proposes a set of graphs that can be used to extract test cases and test sequences. Figure 5 shows the SJS cart as a navigation test tree.

Figure 3: The Cart in SJS web site as a MHDG

Figure 4: The Cart in the SJS web site as a UML object diagram for UMLT

Figure 5: The SJS web site as a Navigation Test Tree constructed from a PND (Page Navigation Diagram) according to the OOWT technique.

The Navigation Test Tree is a spanning tree obtained from the Page Navigation Diagram or PND. It is used to generate a set of test sequences for testing the navigation of a web site.

The AFT technique is the only one that is based on a commercial product, Empirix. Empirix Inc proposes a rich pack of software all dedicated for testing. One of the products, e-Test Suite, contains features such as: e-Manager Enterprise for testing the process management, e-Tester for automated functional and regression testing, e-Load for load and performance testing. E-Load will be used to uncover faults that cause the web site to fail under a heavy load of requests, by simulating thousands of web surfers navigating the site.

The complete derivations generated by applying the TDN technique against the source code of cart.aspx.cs can be found in Table 2. The TDN was applied on a Java Servlet, which mix is a Java class that generates an HTML page by sending HTML tags to its output stream. It is different from the presentation and code-behind feature of .NET. Since this technique break the flow of control into atomic

sections, the code behind is the closest to a Java Servlet. The table below breaks the cart code-behind page into sections to allow us to generate some test cases.

Table 2: Atomic decomposition of the cart.aspx.cs page.

Composite Section	Code
P11	<pre>private void Page_Load(object sender, System.EventArgs e) { auth(); PageTitle = Application["PageTitle"].ToString(); communityid = int.Parse(Application["CommunityId"].ToString()); msg.Text = ""; classes.storecart cstorecart = new classes.storecart(); orderid = cstorecart.GetCartId(userid); if(!IsPostBack)</pre>
P12	<pre>PopulateShoppingCartList(); }</pre>
P21	<pre>public void auth() { CheckPageRole(Request.RawUrl.ToString(), Request.ApplicationPath.ToString()); UpdateLoginControls(); UpdateLogoutControls(); secure.CustomPrincipal cprUser = this.User as secure.CustomPrincipal; if(cprUser != null) {</pre>
P22	<pre>secure.CustomIdentity cidIdentity = cprUser.Identity as secure.CustomIdentity; userid = cidIdentity.SecureId; rolelvl = cidIdentity.RoleLevel; }</pre>
P23	<pre>menu_side1.UID = userid; menu_side1.RID = rolelvl; }</pre>

Composite Section	Code
P41	<pre> public void PopulateShoppingCartList() { SqlConnection myConnection = new SqlConnection(ConfigurationSettings.AppSettings["ConnectionStri ng"]); string cmd = "SELECT store_product.id as prodid, store_product.product, store_order_items.id, store_order_items.type, store_order_items.quantity, store_order_items.price_unit, (store_order_items.price_unit * store_order_items.quantity) as subtotal FROM store_order INNER JOIN store_order_items ON store_order.id = store_order_items.orderid INNER JOIN store_product ON store_order_items.productid = store_product.id WHERE store_order.id="+orderid; SqlDataAdapter da = new SqlDataAdapter(cmd, myConnection); DataSet ds = new DataSet(); da.Fill(ds, "records"); DataView Source = ds.Tables["records"].DefaultView; Recordgrid.DataSource=Source; Recordgrid.DataBind(); myConnection.Close(); classes.storecart cstorecart = new classes.storecart(); ftotal.Text = cstorecart.GetTotal(orderid).ToString(); } </pre>
P5	<pre> public void CheckOut_Click(object sender, CommandEventArgs e){ Response.Redirect("checkout.aspx"); } </pre>
P6	<pre> public void ContinueShopping_Click(object sender, CommandEventArgs e) { Response.Redirect("category.aspx"); } </pre>
P71	<pre> public void Recalculate_Click(object sender, CommandEventArgs e) { UpdateShoppingCartDatabase(); </pre>
P72	<pre> PopulateShoppingCartList(); } </pre>
P81	<pre> void UpdateShoppingCartDatabase() { for (int i=0; i < Recordgrid.Items.Count; i++) { </pre>

Composite Section	Code
P82	<pre> TextBox quantityTxt = (TextBox) Recordgrid.Items[i].FindControl("Quantity"); TextBox amountTxt = (TextBox) Recordgrid.Items[i].FindControl("Amount"); CheckBox remove = (CheckBox) Recordgrid.Items[i].FindControl("Remove"); Label lblProductID = (Label) Recordgrid.Items[i].FindControl("ProductID"); Label lblType = (Label) Recordgrid.Items[i].FindControl("Type"); int productid = int.Parse(lblProductID.Text); char type = char.Parse(lblType.Text); float quantity = float.Parse(quantityTxt.Text); float amount = float.Parse(amountTxt.Text); try { </pre>
P83	<pre> classes.storecart cstorecart = new classes.storecart(); if (quantity == 0 remove.Checked == true) </pre>
P84	<pre> cstorecart.RemoveItem(orderid, productid); </pre>
P85	<pre> else cstorecart.UpdateItem(orderid, productid, quantity, amount); } catch { </pre>
P86	<pre> msg.Text = "There has been a problem with one or more of your inputs."; }}} </pre>
P91	<pre> public bool Display(string input) { if (input.ToString() == "p") return true; </pre>
P92	<pre> else return false; } </pre>
P101	<pre> public bool Enable(string input) { if (input.ToString() == "d") return true; </pre>
P102	<pre> else return false; } </pre>

Test paths can now be extracted from Table 2. Table 3 below shows some of the most critical paths.

Table 3: Test Case Derivation for the cart.aspx.cs program

Test Number	Test Sequence
1	P11.P21.(P22).(P12.P41)
2	P71.(P81.(P82)*.P83.(P84 P85).(P86)
3	P6
4	P5

4.3. Fault Injection

4.3.1. Functional Faults

4.3.1.1. Functional Fault 1

The first functional fault that we will inject is by removing the 'remove' checkbox. The user will then not be able to remove any item from the cart. The cart.aspx will be modified as follows.

Line	Working Code	Faulty Code
70	<pre> <ItemTemplate> <asp:CheckBox ID="Remove" runat="server" /> <asp:Label ID="ProductID" Runat="server" Visible="False" Text='<%# DataBinder.Eval(Container.Data Item, "prodid") %>' /> </pre>	<pre> <ItemTemplate> <asp:Label ID="ProductID" Runat="server" Visible="False" Text='<%# DataBinder.Eval(Container.DataItem, "prodid") %>' /> <asp:Label ID="Type" Runat="server" Visible="False" Text='<%# DataBinder.Eval(Container.DataItem, "type") %>' /> </ItemTemplate> </pre>

- MHDG Technique: When the graph of the modified web site will be rendered, the node corresponding to the 'remove' checkbox will not appear therefore, the fault would have been uncovered. Figure 6 is the corresponding MHDG graph, clearly showing that the remove node is missing.

Figure 6: The MHDG missing without the remove checkbox

- UMLT Technique: The fault will be uncovered by this technique as a mismatch between the variables submitted by the form holding the 'recalculate' button and the dynamic page retrieving them.
- AFT Technique: Since the HTML table in the cart was modified, number of columns before the fault injection will be greater than after the removal of the 'remove' checkbox. Therefore the fault would have been uncovered by this technique, as the assertion below shows :

File: CartLayout

```
if( object = "cart_html table" &
assert new.table.columncount = old.table.columncount
end;
```

- TDN Technique: Any test case generated against a derivation including P81 such as the test case number 2 in table 3, will fail as the checkbox would not have been initialized in the cart.aspx page. Therefore the fault would have been uncovered by this technique.
- OOWT Technique: This technique will fail to uncover the fault, because the removal of the checkbox will not alter the page navigation diagram PND,

as the navigation within the cart will not cause any error. It will not impact the function cluster diagram, which will be rendered the same after the fault injection. The navigation test tree will remain unchanged, and finally the state diagram will not be affected by the change.

4.3.1.2. Functional Fault 2

To inject functional faults, we will modify the cart page so that any item can have its price changed. According to the SJS specifications, only the donations can have their prices set. To implement this change, the code in the file cart.aspx is altered as follows:

Line	Working Code	Faulty Code
65	<pre><asp:TextBox id="Amount" runat="server" Columns="5" CssClass="mainTxt" Text='<# DataBinder.Eval(Container.DataItem, "price_unit") %>' Enabled='<# Enable(DataBinder.Eval(Container .DataItem, "type").ToString())%>' /></pre>	<pre><asp:TextBox id="Amount" runat="server" Columns="5" CssClass="mainTxt" Text='<# DataBinder.Eval(Container.DataItem, "price_unit") %>' Enabled='True' /></pre>

- MHDG Technique: Conditional nodes are fork within a graph that split the flow into two paths according to a certain property. Unlike the UMLT, the MHDG does not support such representation, which, if supported, could have rendered the graph as shown in Figure 7 and uncovered the fault. This idea will be discussed in details in chapter 6.

Figure 7: The proposed modification to allow MHDG to find the functional fault.

Table 4: Forked Trigger-v-Effect Chain

EDG	Events	Trigger-v-Effect Chain
Cart	Update Quantity / Price	<1,1,donation?63:57>

- UMLT Technique: None of the proposed graphs can explicitly expose this fault, but when generating test data by applying the 'All-uses testing' (every variable definition, followed by every one of that variable's usage should be checked from at least one navigation path), data for a donation item will be generated. The cart will then contain a donation with an editable field item and the tester would have found the fault.
- AFT Technique: Considering that the price of the items did not change, the following assertion will discover the fault once an item's price is modified by the user:

```

file: ItemsPrices
if( object = "category_product" &
new.category_product.type = old.category_product.type)
assert
new.category_product.price = old.category_product.price
end;

```

In this assertion, category_product refers to item in the cart. The 'type' property distinguishes a normal item from a donation. The 'price' property is its price. These objects and their attributes are built from the presentation layer, as recommended by the AFT paper, not from the values retrieved from the database.

- TDN Technique: The fault being purely logical and injected in the presentation layer, its discovery by this technique will not be possible.
- OOWT Technique: Since the fault is not object, behavioral, or structural, this technique will not be capable, through the different diagrams it supports such as the Page Navigation diagram, Object State Diagram or the Structure Perspective, to uncover this fault.

4.3.2. Page Structure Faults

To inject a page structure error, we will remove the 'Checkout' button from the cart page. The file cart.aspx will be altered as follows:

Line	Working Code	Faulty Code
81	<pre> <div align="right"><asp:button id="butt_shop" CssClass="button" Runat="server" Text="Continue Shopping" OnCommand="ContinueShopping_Click "></asp:button><asp:button id="butt_update" CssClass="button" Runat="server" Text="Recalculate" OnCommand="Recalculate_Click"></a sp:button><asp:button id="butt_checkout" CssClass="button" Runat="server" Text="Check Out" OnCommand="CheckOut_Click"></asp: button></div> </pre>	<pre> <div align="right"><asp:button id="butt_shop" CssClass="button" Runat="server" Text="Continue Shopping" OnCommand="ContinueShopping_Click "></asp:button><asp:button id="butt_update" CssClass="button" Runat="server" Text="Recalculate" OnCommand="Recalculate_Click"></a sp:button></div> </pre>

- MHDG Technique: The removal of the 'Checkout' button means that the event triggered by the node 83 will not be generated. Since this

fault is navigational, coverage based testing should reveal it. Unfortunately, since the element removed is not hyperlink, the All-hyperlinks testing will not uncover it. All-input-GUI testing will also fail, since the button is not rendered in the browser window and is therefore invisible. Finally, All-events testing will also fail, since the generator of the event, the checkout button, was removed. Other methods such as the retrieve level testing and the update level testing will not uncover the fault, unless they are used to draw the graph of the altered web site, like shown in Figure 8. This will then clearly show that the shaded area is not reachable, therefore uncovering the fault.

Figure 8: The MHDG Page Structure Fault

- UMLT Technique: When applying the web site reduction, to perform a controlled 'All-paths testing', the checkout page will appear unreachable from any node. Figure 9 shows the UML diagram for the modified web site, and the checkout page clearly appears isolated from the cart, thus the fault will be uncovered.

Figure 9: The UMLT Page Structure Fault

- AFT Technique: When performing the SQL database validation, there will not be a single record of a successfully completed transaction, since the checkout function which is only available in the cart page has now been disabled. This technique would have found this fault.
- TDN Technique: Any test case generated against a derivation including P5 such as the test case number 4 in table 3, will not run the redirection code, since the button was removed from the cart. The fault will be uncovered.
- OOWT Technique: As shown in figure 5, and after injecting this fault, the removal of the checkout button will cause the test sequence number 4 (Url-B, Url-C) to fail, thus uncovering the fault. Figure 10 shows the PND diagram of the modified pages.

Figure 10: The OOWT Page Structure Fault

4.3.3. Security Faults

To inject security errors, we will modify the cart page so that it will not check the validity of the calling users. As a result, any user can now fill the cart without being authenticated. The file `cart.aspx.cs` will be altered as follows:

Line	Working Code	Faulty Code
59	<code>userid = <u>cidIdentity.SecureId</u>;</code>	<code>userid = 0;</code>
60	<code>rolelvl = <u>cidIdentity.RoleLevel</u>;</code>	<code>Rolelvl = 0;</code>

- MHDG Technique:** The MHDG shows that the page 'Any Page' with a hyperlink 'Cart' can successfully access the cart without establishing an authenticated session. In order to detect this fault, the All-hyperlinks testing will be used and all the links in the tested pages will be visited. Figure 11 shows the MHDG graph with a focus on the node 901 which can perform the following illegal trigger-v-effect.

MHDG	Event(s)	Trigger-v-Effect Chain
------	----------	------------------------

Cart	Visit cart from any page	<901,1,902>
------	--------------------------	-------------

Table 5: Trigger-v-Effect Chain for finding the security fault.

Figure 11: The MHDG Security Fault

- UMLT Technique: By applying this technique on the modified web site, the generated UML diagram will look like Figure 12. The shaded area shows that an HTML page called 'Any Page' can successfully connect to the cart page without the need to establish an authenticated session. This technique uncovered the fault.

Figure 12: The UMLT Security Fault

- AFT Technique: When a user is not logged in, the default value for the userid and Rolelvl will be zero. Thus the following assertion will detect the security fault:

```
file: FindUnauthorized
if( object = "userid")
assert
new.userid <> 0
end;
```

- TDN Technique: Any test case generated against a derivation including P21 will uncover this fault.
- OOWT Technique: A page on the web site can access the cart without the need to authenticate. Figure 13 shows the page navigation diagram after the injection of the fault. The page 'Any Page' is allowed to successfully connect to the cart without requesting authentication.

Figure 13: The OOWT Security Fault

Note: Security fault are usually captured on the next form submission as the unauthorized user will not be able to be assigned to an existing session.

4.3.4. Performance Slowdown

To inject performance slowdown, the size of the connection pool to the database server will be reduced, so that newly connected users will have to wait for other visitors to release the connection to the pool. There is no alteration to the code to implement this feature, just a database configuration.

- MHDG Technique: The MHDG cannot represent the average time a request is taking, and therefore will be unable to discover this fault.
- UMLT Technique: The dynamic and static testing method proposed by this paper will fail to detect the fault as it is neither navigational nor logical. Its discovery is impossible by any test data generated by this technique.
- AFT Technique: Empirix, the company behind the tool used by the authors of this paper, e-Load product from Empirix is an enterprise load and performance testing solution that allows users to stress .NET Web applications and Web services while analyzing application

performance and scalability. By reusing scripts created in e-Tester, another Empirix product, users can simulate hundreds or thousands of virtual users interacting with a web server, therefore uncovering this fault.

- TDN Technique: This technique does not present any method for testing web server loading; therefore it will fail to uncover this fault.
- OOWT Technique: Neither of the object or behavior perspective can uncover this fault. All the states will be correct, and this technique will always render correct graphs.

Note: The slowdown issue might be detected if the web application generates timeouts like the session timeouts in web mail and Internet banking application. Otherwise, it will remain unseen.

4.3.5. Database Faults

To inject database faults, the query in the cart page will be altered to ignore the quantity. Every summing for the values of the cart content will be ignored no matter what quantity the user specified. To implement this change, the code in the file cart.aspx.cs will be altered as follows:

Line	Working Code	Faulty Code
70	<pre>string cmd = "SELECT store_product.id as prodid, store_product.product, store_order_items.id, store_order_items.type, store_order_items.quantity, store_order_items.price_unit, (store_order_items.price_unit * store_order_items.quantity) as subtotal FROM store_order INNER JOIN store_order_items ON store_order.id = store_order_items.orderid INNER JOIN store_product ON store_order_items.productid = store_product.id WHERE store_order.id="+orderid;</pre>	<pre>string cmd = "SELECT store_product.id as prodid, store_product.product, store_order_items.id, store_order_items.type, store_order_items.quantity, store_order_items.price_unit, (store_order_items.price_unit * 1) as subtotal FROM store_order INNER JOIN store_order_items ON store_order.id = store_order_items.orderid INNER JOIN store_product ON store_order_items.productid = store_product.id WHERE store_order.id="+orderid;</pre>

- MHDG Technique: This fault is not a navigational fault and therefore all the coverage-based testing will not uncover it. Event flow testing will also fail, because all the events involved in this page will be successfully triggered and handled. Finally, Data flow testing will fail, because the error is in the code-behind page, not in its transmission to

the presentation front-end page. This is why the MHDG technique will not uncover this fault.

- UMLT Technique: The meta model proposed by this technique does not provide any mechanism to vary the page content. The only variables in the diagram are those that are sent in an html form. Even the dynamic validations (page testing, hyperlink testing, definition-use testing and all- path testing) cannot uncover a database fault. This is why this technique fails to locate this fault.
- AFT Technique: Being a read query, which does not impact the content of the database, the SQL validation will not be useful. As for assertions, the fault does not modify any variable that can be asserted against a previously recorded value, which makes the fault undetected.
- TDN Technique: Any test case generated against a derivation including P41, such as the test case number 3 in table 3, will uncover this fault. The tester will notice that the quantities are always one, even if it is changed and saved to the database.
- OOWT Technique: Being neither an object, behavioral nor structural fault, this technique will not uncover this fault. None of the proposed diagram can detect such a fault, since the navigation will be correct. No dead ends, dead locks or dead links. The state of the cart as shown in figure 14 will always be correct.

Figure 14: The Cart OOWR state diagram

4.3.6. Dataflow Errors

To inject data flow error, the method `Page_Loading()` in the file `cart.aspx.cs` will be altered so that the call to the `IsPostBack()` function is commented. This will cause the cart page to be always rendered empty at any item, since the commented code is responsible of loading the items added by the user to his cart.

Line	Working Code	Faulty Code
45	<code>if (!IsPostBack)</code>	<code>//if (!IsPostBack)</code>
46	<code>PopulateShoppingCartList();</code>	<code>//PopulateShoppingCartList();</code>

- **MHDG Technique:** The update level testing will be used to uncover this fault. This test is used to ensure that the requested changes are applied. In this case, when adding an item to the cart, the visible effect will be the direct redirection to the cart page which will be populated with the items previously bought, if any, plus the newly added item. This fault was injected in order to display an empty cart although the database was correctly filled. Figure 15 shows the graph of the

modified cart page, and table 6 shows the trigger-v-effect chains (triggers with visual effect on the navigation) in the update level.

Table 6: Events of the MHDG and trigger-v-effect

EDG	Event(s)	Trigger-v-Effect Chains
Cart	Add a product to the cart from the category_product page.	<475,1,538>
	Reach the cart page with content that was not previously checked out.	<1,1,82>

Figure 15: The MHDG DataFlow Errors

- UMLT Technique: The content of the cart will always be empty, although the diagram shows that the path is navigable. This implies that the error is in the source of the cart program. The location of the error will be uncovered, Figure 16.

Figure 16: The UMLT DataFlow Errors

- AFT Technique: The `populateShoppingCartList` method invokes the query defined in the previous fault, which will make it also undetectable by this technique.
- TDN Technique: When applying test case 1 described in table 2, the fault will be uncovered.
- OOWT Technique: Many test cases can be generated from the Function Cluster Diagram (FCD) as shown in Figure 17, some of which are displayed in Table 7.

Table 7: Test cases generated from FCD

Test	Function Call Sequence	Detects Current Fault
1	F1,F9,F4	TRUE
2	F6,F7,F4	FALSE
3	F2	FALSE
4	F8	FALSE

Test case number 1 will detect a failure as the call to F4 will not be

executed.

Figure 17: The OOWT DataFlow Errors diagram

4.3.7. Event Flow Faults

To inject an event flow fault, the 'Check Out' button in the cart page will have its associated code disabled. The file cart.aspx.cs will be altered as follows:

Line	Working Code	Faulty Code
88	<code>Response.Redirect("checkout.aspx");</code>	<code>//Response.Redirect("checkout.aspx");</code>

- MHDG Technique: The MHDG show that the checkout area of the site is unreachable and that the 'Checkout' button in the cart page (node number 83) if clicked, will not do any action. Therefore, only coverage-based testing will be able to detect this fault. Since the checkout page is reached through a button, the All-input-GUI testing will uncover it. Figure 18 clearly shows this.

Figure 18: The MHDG Event Flow Faults

- UMLT Technique: By generating the UML diagram against the modified web site, the link between the checkout button and the checkout page will not be rendered as shown in Figure 19, therefore uncovering the fault.
- AFT Technique: When performing the SQL database validation, there will not be a single record of a successfully completed transaction, since the checkout function which is only available in the cart page has now been disabled. This technique would have found this fault.
- TDN Technique: Test case number 4 described in Table 3 will uncover this fault.
- OOWT Technique: From the test cases shown in Figure 5, test case number 3, Url-D will fail uncovering the fault. On the page navigation diagram, the link between the cart page and the categories page will not appear, Figure 20.

Figure 19: The UMLT Event Flow Faults

Figure 20: The OOWT Event Flow Faults

4.3.8. Architectural Errors

Most Architectural errors will make the website being tested inaccessible to a part or not all the clients trying to connect to it. A typical example is trying to run the website on a server that does not have the .NET framework installed. No code needs to be altered to apply this fault. All of the proposed testing technique will detect a major problem with the SJS.

- MHDG Technique: All the tests will fail as the server is not available, therefore uncovering a major fault.
- UMLT Technique: All the tests will fail as the server is not available, therefore uncovering a major fault.
- AFT Technique: All the tests will fail as the server is not available, therefore uncovering a major fault.
- TDN Technique: All the tests will fail as the server is not available, therefore uncovering a major fault.
- OOWT Technique: All the tests will fail as the server is not available, therefore uncovering a major fault.

4.3.9. Navigation Errors

To inject navigational behavior error, the cart page will be altered, so that the 'Continue Shopping' button will always go to the home page. The file cart.aspx.cs will be altered as follows:

Line	Working Code	Faulty Code
94	<code>Response.Redirect("category.aspx");</code>	<code>Response.Redirect("http://www.sjsalumni.org");</code>

- MHDG Technique: The 'Continue Shopping' (node number 81) will now point to the home page (node number 899). As this modification will not prevent the triggering of the event but rather the resulting effect, this fault is a navigation fault. The retrieve level testing will uncover this fault as the two tables 8 and 9 show.

Table 8: The trigger-v-effect chain before injecting the fault.

EDG	Event(s)	Trigger-Effect Chain
Cart	Continue Shopping	<1,274,81>

Table 9: The trigger-v-effect chain after injecting the fault.

EDG	Event(s)	Trigger-Effect Chain
-----	----------	----------------------

Cart	Continue Shopping	<1,899,81>
------	-------------------	------------

Before injecting the fault, the event triggered by node 81 would always take the user from node 1 to node 274. After injection of the fault, that same trigger will always take the user to node 899, which is the home page. Figure 21 shows the MHBG against the modified code base.

Figure 21: The MHDG Navigation Errors

- UMLT Technique: By generating the UML diagram against the modified web site, the category page will appear as not reachable from the cart page while the home page will be; therefore the fault will be uncovered by this technique, Figure 22.
- AFT Technique: The modification presented in this injection will not be detected because it only involves read queries from the database, and it does not modify any variable that can be asserted against a previously recorded value. This technique will be unable to uncover this fault.
- TDN Technique: Test case number 3 described in Table 3 will uncover this fault.
- OOWT Technique: Test case number 1 (Url-A, Url-B, Url-C) and 4 (Url-B, Url-C) described in Figure 5 will uncover the fault. The page

navigation diagram will show that the link between the cart page and the category page is missing. The page navigation diagram in Figure 23 shows the navigation map after injecting the fault.

Figure 22: The UMLT Navigation Errors

Figure 23: The OOWT Navigation Errors

CHAPTER V

COMPARISON OF TESTING TECHNIQUES

5.1 Discussion of Results

The table below shows a summary of the synthesis of the experimental results found in chapter 4.

Table 10: Summary of experimental results

#	Faults	MHDG	UMLT	AFT	TDN	OOWT
1	Functional 1	Found	Found	Found	Found	Not Found
	Functional 2	Not Found	Found	Found	Not Found	Not Found
2	Page Structure	Found	Found	Found	Found	Found
3	Security	Found	Found	Found	Found	Found
4	Performance	Not Found	Not Found	Found	Not Found	Not Found
5	Database	Not Found	Not Found	Not Found	Found	Not Found
6	Data Flow	Found	Found	Not Found	Found	Found
7	Event Flow	Found	Found	Found	Found	Found
8	Architecture	Found	Found	Found	Found	Found
9	Navigation	Found	Found	No Found	Found	Found
Total		8 / 10	8 / 10	7 / 10	8 / 10	6 / 10

Based on the first functional fault, the graph based techniques MHDG, UMLT and OOWT represent code-based static info that do not easily represent dynamic aspects like generating features by reading from the database.

5.1.1. Graph Based Techniques

The proposed techniques can be divided in two categories: Graph based, such as the MHDG, UMLT, OOWT, and non graph based, such as the other two, AFT and

TDN.

Graph based testing technique offers to test a web site by representing its structure and behavior as a graphical map. The HTTP protocol used to serve web content is stateless and session less. It means that from the web server point of view, every request the client makes is unique. There is no session that is kept for every user, although this behavior is simulated programmatically using a unique identifier that is presented by the user to the server on every request. This identifier is stored on the requester browser as a cookie. It is up to the web application to make the link between the user and the corresponding data on the server. Any error in this process will make the user go to the wrong page or see a content that was not meant for him. This is why web testers focus on navigational testing as it is by far the most common error encountered on web servers. Most of the fault injected in this paper (Navigation, Event Flow, Data Flow, Page Structure and even Security) can be seen as navigational. Since graph based testing techniques offer the best navigational testing techniques, they are highly adequate for web testing.

On the other hand, the success of the Internet and the technological advance in the development technique such as .NET and J2EE are the proliferation of web portal solution are pushing web designer to make richer and richer web pages by adding more elements to them.

Another serious limitation we faced when studying graph based testing techniques (MHDG, UMLT and OOWT) was the unavailability of software or algorithms that could help us evaluate the computing cost, such as the big O, the human effort required to implement these diagrams. Only the UMLT technique mentioned a couple of software, called ReWeb and TestWeb, but we were unable to acquire them or to have details about their implementation.

The MHDG technique achieved a good total in Table 10, but it took great effort to draw. The nodes being tied to line numbers, we had to start by renumbering the cart pages and all the pages it reaches directly. Our concern is that any change in the source code will require us to renumber, and therefore redraw the graph. The poor documentation and the unavailability of a precise computable algorithm (an algorithm that will allow us to build a program that will generate the graph from the .NET sources) mean that we have to generate three versions of the diagram before getting it right. But once generated, the MHDG shows all its power by clearly uncovering most of the faults we injected.

The UMLT is the only technique that proposed an algorithm in the form a UML diagram to generate the graph of a web site. This is why it is the only technique that managed to produce a software. One problem is that the ReWeb and TestWeb applications could not be found anywhere, as it would have allowed us to demonstrate how this technique will find or miss the faults we injected.

The OOWT technique is based on an ORD diagram. It proposes different graphs for testing different faults. The drawback of this approach is that many graphs need to be generated, and before deciding if a fault was uncovered or not, the tester will need to evaluate all of the diagrams.

5.1.2. Non Graph Based Techniques

The other non graph techniques, the AFT and TDN offer much simple solutions to old problems. The AFT solution is based on two software, the Empirix traffic recorder and a custom developed parser and assertion tool. The later seems to be a commercial application that was delivered to a single customer and does not seem to be available for sale, but a similar functionality could be offered by HTTP Unit.

The TDN offers a technique of splitting the code of a servlet into atomic section and then uses arguments from the regular expressions to get the different paths that could be traversed by an execution point. This technique is good at helping the tester in the generation of the test data, but it suffers a few serious limitations: It does not consider the code-behind feature of .NET applications, and it does not take into consideration the HTTP sessions mention earlier in this chapter. This testing handles every request independently of any session context. This will make this technique particularly weak at detecting any fault related to a context.

5.2. Computational Cost

An accurate evaluation of the computational costs is impossible without a software implementation and the algorithms used by the different techniques. Our evaluation of the cost relates to the effort that we had to put in to generate the different graphs and test cases for each of the techniques:

- The MHDG was the most difficult as its graph is very complex and tightly related to the code base.
- The OOWT requires five graphs (The Object Relation Diagram, ORD, the Page Navigation Diagram PND, the COSD state diagram, the function cluster

diagram, and finally the navigation test tree) to be generated. This is why this technique has a high computational cost.

- The UMLT has a medium cost as it only requires one graph to be generated, and provides an algorithm to do so. Its main drawback is that it requires a lot of user intervention, especially when using the TestWeb application where the tester is supposed to present the test data and validate the software's output.
- The TDN has a very low cost as it only provide the tester with means of generating test data.
- The AFT technique has medium cost as it uses an existing software to record the web session, but still need an assertion tool to analyze their content. The later is not available for sale and therefore needs to be developed.

5.3. Summary

This chapter provides a summary of the web application testing approaches discussed in the previous chapter. The summary is shown in Table 11. The table columns are described in the following. Approach: This column contains the name, or at least a brief description about each web application testing approach that was described in this paper. References: This column provides, for each approach, a reference in which the concepts of the approach were developed. Models Used: This column describes any models that were described for use in each web application testing approach. Empirical Work Reported: This column shows the empirical work detection of each web application testing approach, if an experimental analysis was performed in the reference defined in the same row. Faults Empirically Detected: This column specifies which faults each web application testing approach addresses. Computational Cost: This column shows how High, Medium or Low the technique is, in detecting a certain fault.

Table 11: Summary of Web Application Testing Techniques

Approach	References	Models Used	Empirical Work Reported	Faults Empirically Detected	Computational Cost
Formal Specifications	Jia and Liu (2001)	XML and formal Specification language	No	Functional, Page structure, Security, Performance	
Web Database Applications	Deng, Frankl and Wang (2004)	Web application graph	No	Database	
Event based Dependence Graphs (MHDG)	Mansour and Hourl (2005)	Dependence graphs	No	Page Structure, Security, Database, Data flow, Event flow, Architecture, Navigation	High
UML representation for static testing (UMLT)	Ricca and Tonella (2001)	UML diagram	No	Functional, Page Structure, Security, Data flow, Event flow, Architecture, Navigation	Medium
Architectural and Functional testing (AFT)	Sneed (2004)	None	Yes	Functional, Page Structure, Security, Performance, Event flow, Architecture	Medium
Testing for Dynamic Nature (TDN)	Wu and Offutt (2002)	Regular expressions	No	Page Structure, Security, Database, Data flow, Event flow, Architecture, Navigation	Low
Object Oriented Web Testing Model (OOWT)	Kung et.al (2000)	Object relation diagram, page navigation diagram, object state diagram, block branch diagram, function cluster diagram, interprocedural control flow graph	No	Page Structure, Security, Data flow, Event flow, Architecture, Navigation	High
Agent Framework	Kung (2004)	AOSE	No		

CHAPTER VI

CONCLUSION AND FURTHER WORK

In this document, we presented a number of testing techniques for testing web sites. After the detailed analysis of all the presented papers, one question will prevail: How would we select or recommend a method?

It all depends on three main factors:

- The size of the website in term of pages scripts and links.
- The time available for testing. Ideally, testing should last until we run out of time and resources.
- What needs to be tested, the content, the navigation, the architecture etc.

Event based dependency graphs take a lot of time to draw and will produce confusing representations for huge websites. Nevertheless they produce sharp representation of web surfing behaviors as series of event flow in petri-net like diagrams. It is ideal for small web sites with highly connected components. The UML representation for static testing can handle large web sites through the tools presented by its author, ReWeb and TestWeb. It is especially useful for path testing and quality control. The architectural and functional testing seemed to be the sharpest of all techniques, being able to give a precise description of the error and its location. Unfortunately, this technique introduces new challenges if it is to be applied, as it only becomes efficient with a large amount of history data. Ironically, an important number of log messages will require data mining skills and tools to be able to fetch error patterns. The testing of dynamic nature of web sites is not graph based and it extracts all the possible execution paths from the source code. It is quite efficient at detecting flow of control errors and is a good mean in producing test data. Finally, the object oriented web testing model will produce a set of graphs which are referred to as perspectives that can be navigational, behavioral or structural. Its implementation is time consuming as a lot of graph needs to be produced in order to unveil all types of faults.

But the techniques presented above are just part of the solution. Their software implementation, where feasible, will surely imply a different judgment.

Further work can be done as follows:

a. There is a lot of room for improvement, especially concerning the MHDG technique. We believe that the following improvement could highly benefit this new technique: Adding support for fork. The UMLT technique allows the

conditional links from one node to another based on a condition. These are referred to as conditional edges. The difference with other edges is that they contain the description of the condition to be fulfilled, for example: $c = (x1 = \text{"books"})$ will cause the edge below to be visited only if the variable $x1$ is equal to "books".

We propose to introduce a similar feature into the MHDG to allow faults such as the functional fault introduced in chapter 4. The conditional edges will emanate from a diamond shaped node which we will refer to as a fork node. It is similar to the if statement in a flowchart diagram, but can have many edges leaving it. The fork node is labeled with the condition to be tested, and the corners where the vertexes are emanating are labeled with conditional values.

b. One thing missing from the MHDG is an algorithm that will allow us to draw a correct diagram from the source code of the web site. If available, it will make this approach much more attractive especially when dealing with complex or constantly changing web sites.

c. If an algorithm can be defined for the generation of a MHDG, then next logical step will be to write a software that implements it. If the software is capable of generating an interactive graph, then the tester will be able to test a web site just by clicking on the correct nodes and seeing how the event will be triggered and how the token will be moving around the graph, similarly to an interactive Petri-net.

d. Our study only injected one fault of each type to assess the efficiency of each testing technique. In order to do deep assessment, further work could inject many more errors of each kind and see which techniques are the most efficient.

No method can be set as the most efficient of all. Web testing is still a young science and we are just starting to uncover its strengths. There is much space here for improvement and innovation. Web applications are getting richer every day and new technologies are appearing in the web browser window at an unprecedented rate. Web testing should not be seen as an optional practice but an integrated part of any web development process.

Figure 24: The conditional edges as proposed by the UMLT technique

REFERENCES

- Bedi, S., Schroeder, P. (2004). Observations on the Implementation and Testing of Scripted Web Applications. *In the Proceedings of the Sixth IEEE International Workshop on the Web Site Evolution (WSE'04)*, Wisconsin, Milwaukee, pp. 20-27.
- Beizer, B. (1990). *Software Testing Techniques*, New York, Van Nostrand Reinhold.
- Deng, Y., Frankl, P. & Wang, J. (2004). Testing Web Database Applications. *ACM SIGSOFT Software Engineering Notes*, 29(5).
- Dyck, S., Sloane, D, (1999). *Software Testing*.
<http://sern.ucalgary.ca/~sdyck/courses/seng621/webdoc.html#Unit>.
- Glossary of Internet Terms, (2004). <http://www.fmnetdesign.com/glossary.htm>
- Hieatt, E. & Mee, R. (2002). Going Faster: Testing the Web Application. *IEEE Software*, 19(22), pp. 60-65.
- Howden, W. (1986). A functional approach to program testing and analysis. *IEEE Trans. On Software Engineering*, 12(10), pp. 997-1005.
- HttpUnit.(2004). <http://www.javaworld.com/javaworld/jw-04-2004/jw-0419-httpunit>
- Internet and Web Glossary, (2000).
<http://www.webliminal.com/essentials/glossary.htm>
- Jia, X. & Liu, H. (2001). Rigorous and Automatic Testing of Web Applications. *Proceedings of the DePaul CTI Research Symposium (CTIRS)*, San Francisco, California.
- Kung, D. (2004). An agent-based framework for testing Web applications, *In Proceeding of First International Workshop on Quality Assurance and Testing of Web Applications, in conjunction with COMPSAC'04*, Hong Kong, September 30, 2004, IEEE Computer Society Press, 2004.
- Kung, C., Liu, C.H. & Hsia, P. (2000). An Object-Oriented Web Test Model for Testing Web Applications. *In Proceedings of the IEEE 24th International Computer Software and Application Conference (COMPSAC 2000)*, Taipei, Taiwan, pp. 537-542.
- Mansour, N., Hourri M. (2005). Testing Web Applications. *To appear in Information and Software Technology Journal*.

- Mansour, N., and Salame, M. (2004). Data generation for path testing. *Software Quality Journal*, 12: pp. 121-136.
- Microsoft's Web Application Stress Tool. (2000). <http://www.westwind.com/presentations/webstress/webstress.htm>
- Offutt, A. J., Jin, Z., and Pan, J. (1999). The dynamic domain reduction procedure for test data generation, *Software Practice and Experience*, 29(2): 167-193.
- Pressman, R.S. (2005). Software Engineering: A Practitioner's Approach, R.S. Pressman and Associates, *Testing for Web Apps*, Chapter 20.
- Puffin. (2002). <http://www-106.ibm.com/developerworks/library/os-puffin.htm>
- Rapps, S. and Weyuker, E. (1985). Selecting software test data using data flow information. *IEEE Trans. on Software Engineering*, April: 367-375.
- Ricca, F., Tonella, P. (2001). Analysis and Testing of Web Applications. *In the Proceedings of the International Conference on Software Engineering (ICSE'2001)*, Toronto, Canada, pp. 25-34.
- Sneed, H. (2004). Testing a Web Application. *In the Proceedings of the Sixth IEEE International Workshop on Web Site Evolution (WSE'04)*, Chicago, Illinois, pp. 3-10.
- Software Testing, (2005). http://www.en.wikipedia.org/wiki/Software_testing.
- Splaine, S., Jaskiel, S. P. (2001). The Web Testing Handbook, STQE Publishing, *Emerging Technologies*.
- WAPT. (2004). <http://www.loadtestingtool.com>
- Web Application, (2005). http://www.en.wikipedia.org/wiki/Web_application.
- Wu, Y. and Offutt, J. (2002). Modeling and Testing Web-based Applications. *In GMU ISE Technica ISE-TR-02-08*.
- Xiong, P., Probert, R.L. (2003). A Multi-Approach Testing Methodology in TTCN for Web Applications. Fast Abstract ISSRE 2003.

Appendix A - SJS Shopping Cart Code

cart.aspx

```
1 <%@ Register TagPrefix="uc" TagName="store_category" Src="../../ascx/store/store_category.ascx" %>
2 <%@ Register TagPrefix="uc" TagName="menu_side" Src="../../ascx/menu/menu_side.ascx" %>
3 <%@ Register TagPrefix="uc" TagName="login" Src="../../secure/login.ascx" %>
4 <%@ Register TagPrefix="uc" TagName="header" Src="../../ascx/header.ascx" %>
5 <%@ Register TagPrefix="uc" TagName="footer" Src="../../ascx/footer.ascx" %>
6 <%@ Page language="c#" Codebehind="cart.aspx.cs" AutoEventWireup="false" Inherits="sjsalumni.aspx.user.store.cart" %>
7 <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN" >
8 <HTML>
9 <HEAD>
10 <title>
11 <%=PageTitle%>
12 </title>
13 <LINK href="../../css/styles.css" type="text/css" rel="stylesheet">
14 <meta content="Microsoft Visual Studio 7.0" name="GENERATOR">
15 <meta content="C#" name="CODE_LANGUAGE">
16 <meta content="JavaScript" name="vs_defaultClientScript">
17 <meta content="http://schemas.microsoft.com/intellisense/ie5" name="vs_targetSchema">
18 </HEAD>
19 <body MS_POSITIONING="GridLayout">
20 <form id="cart" method="post" runat="server" class="mainTxt">
21 <table height="370" cellSpacing="0" cellPadding="0" width="760" align="center" border="0">
22 <TBODY>
23 <tr>
24 <td height="84"><uc:header id="header1" runat="server"></uc:header></td>
25 </tr>
26 <tr>
27 <td vAlign="top">
28 <table height="16" cellSpacing="0" cellPadding="0" width="760" border="0">
29 <tr>
30 <td vAlign="top" align="left" width="140" bgColor="#eaeef3" height="16"><IMG height="16" src="/image/PontG.gif" width="1">
31 <td vAlign="top" align="left" width="15" height="16"><IMG height="16" src="/image/PontccG.gif" width="15"></td>
32 <td width="385" background="/image/PontTile.gif" height="16"></td>
33 <td vAlign="top" align="left" width="14" height="16"><IMG height="16" src="/image/PontccD.gif" width="14"></td>
34 <td vAlign="top" align="right" width="206" bgColor="#eaeef3" height="16"><IMG height="16" src="/image/PontG.gif" width="1">
35 </tr>
36 </table>
37 <table cellSpacing="0" cellPadding="0" width="760" border="0">
38 <TBODY>
39 <tr>
40 <td vAlign="top" align="left" width="1" bgColor="#ababab"></td>
41 <td vAlign="top" align="middle" width="139" bgColor="#ffffff"><uc:login id="login1" runat="server"></uc:login><br>
42 <uc:menu_side id="menu_side1" runat="server"></uc:menu_side><br>
43 <br>
44 </td>
45 <td vAlign="top" width="414" bgColor="#ffffff">
46 <table cellPadding="3" width="414" bgColor="#ffffff" border="0">
47 <tr>
48 <td class="mainTxt">
49 <div class="heading">. Store :: Shopping Cart</div>
50 <br>
51 <asp:label id="msg" CssClass="mainTxt" Runat="server" ForeColor="#FF0000"></asp:label><br>
52 <asp:datagrid id="Recordgrid" runat="server" CssClass="mainTxt" AutoGenerateColumns="false" DataKeyField="quantity" CellPad
53 <ItemStyle CssClass="tdbgclr"></ItemStyle>
54 <HeaderStyle Font-Bold="True" CssClass="tablehead"></HeaderStyle>
55 <Columns>
56 <asp:HyperLinkColumn DataTextField="product" DataNavigateUrlField="prodid" DataNavigateUrlFormatString="product.aspx?ID={
57 <asp:TemplateColumn HeaderText="Qty" HeaderStyle-HorizontalAlign="Center" ItemStyle-HorizontalAlign="Center">
58 <ItemTemplate>
59 <asp:TextBox id="Quantity" runat="server" Columns="3" MaxLength="3" CssClass="mainTxt" Text="<%= # DataBinder.Eval(Contai
60 </ItemTemplate>
61 </asp:TemplateColumn>
62 <asp:TemplateColumn HeaderText="Amount" HeaderStyle-HorizontalAlign="Center" ItemStyle-HorizontalAlign="Center">
63 <ItemTemplate>
64 $
65 <asp:TextBox id="Amount" runat="server" Columns="5" CssClass="mainTxt" Text="<%= # DataBinder.Eval(Container.DataItem, "p
66 </ItemTemplate>
67 </asp:TemplateColumn>
68 <asp:BoundColumn HeaderText="Subtotal" DataField="subtotal" DataFormatString="{0:c}" ItemStyle-HorizontalAlign="Right" Hea
69 <asp:TemplateColumn HeaderText="Remove" HeaderStyle-HorizontalAlign="Center" ItemStyle-HorizontalAlign="Center">
70 <ItemTemplate>
71 <asp:CheckBox ID="Remove" runat="server" />
72 <asp:Label ID="ProductID" Runat="server" Visible="False" Text="<%= # DataBinder.Eval(Container.DataItem, "prodid") %>" />
73 <asp:Label ID="Type" Runat="server" Visible="False" Text="<%= # DataBinder.Eval(Container.DataItem, "type") %>" />
74 </ItemTemplate>
75 </asp:TemplateColumn>
76 </Columns>
77 </asp:datagrid><br>
```



```

78 <strong>Total: </strong> $<asp:label id="ftotal" CssClass="mainTxt" Runat="server"> </asp:label>
79 <br>
80 <br>
81 <div align="right"><asp:button id="butt_shop" CssClass="button" Runat="server" Text="Continue Shopping" OnCommand="Coni
82 <asp:button id="butt_update" CssClass="button" Runat="server" Text="Recalculate" OnCommand="Recalculate_Click"> </asp:bu
83 <asp:button id="butt_checkout" CssClass="button" Runat="server" Text="Check Out" OnCommand="Checkout_Click"> </asp:but
84 <br>
85 </td>
86 </tr>
87 </table>
88 </td>
89 <td class="tdbgclr" vAlign="top" align="right" width="205"><uc:store_category id="store_category1" runat="server"> </uc:store
90 </td>
91 <td vAlign="top" align="left" width="1" bgColor="#ababab"> </td>
92 </tr>
93 </TBODY>
94 </table>
95 </td>
96 </tr>
97 <tr>
98 <td><uc:footer id="footer1" runat="server"> </uc:footer> </td>
99 </tr>
100 </TBODY>
101 </table>
102 </TD></TR></TABLE></form>
103 </body>
104 </HTML>

```

cart.aspx.cs

```

105 using System;
106 using System.Collections;
107 using System.ComponentModel;
108 using System.Data;
109 using System.Data.SqlClient;
110 using System.Configuration;
111 using System.Drawing;
112 using System.Web;
113 using System.Web.SessionState;
114 using System.Web.UI;
115 using System.Web.UI.WebControls;
116 using System.Web.UI.HtmlControls;
117
118 namespace sjsalumni.aspx.user.store
119 {
120     /// <summary>
121     /// Summary description for cart.
122     /// </summary>
123     public class cart : secure.Security
124     {
125         protected System.Web.UI.WebControls.DataGrid Recordgrid;
126         protected System.Web.UI.WebControls.Label msg;
127         protected System.Web.UI.WebControls.Label ftotal;
128         protected System.Web.UI.WebControls.Button butt_shop;
129         protected System.Web.UI.WebControls.Button butt_update;
130         protected System.Web.UI.WebControls.Button butt_checkout;
131         protected ascx.menu.menu_side menu_side1;
132         int userid,orderid;
133         public string PageTitle;
134         int communityid,rolelv;
135
136         private void Page_Load(object sender, System.EventArgs e)
137         {
138             auth();
139
140             PageTitle = Application["PageTitle"].ToString();
141             communityid = int.Parse(Application["CommunityId"].ToString());
142
143             msg.Text = "";
144
145             // Get CartID (orderid)
146             classes.storecart cstorecart = new classes.storecart();
147             orderid = cstorecart.GetCartId(userid);
148
149             if(!IsPostBack)
150                 PopulateShoppingCartList();
151         }
152
153         public void auth()
154         {
155             CheckPageRole(Request.RawUrl.ToString(),Request.ApplicationPath.ToString());
156             UpdateLoginControls();
157             UpdateLogoutControls();

```



```

158
159 secure.CustomPrincipal cprUser = this.User as secure.CustomPrincipal;
160 if(cprUser != null)
161 {
162     secure.CustomIdentity cidIdentity = cprUser.Identity as secure.CustomIdentity;
163     userid = cidIdentity.SecureId;
164     rolelvl = cidIdentity.RoleLevel;
165 }
166 menu_side1.UID = userid;
167 menu_side1.RID = rolelvl;
168 }
169
170 public void PopulateShoppingCartList()
171 {
172     SqlConnection myConnection = new SqlConnection(ConfigurationSettings.AppSettings["ConnectionString"]);
173
174     string cmd = "SELECT store_product.id as prodid, store_product.product, store_order_items.id, store_order_items.type, store_order_items.orderid=" + orderid;
175
176     SqlDataAdapter da = new SqlDataAdapter(cmd, myConnection);
177     DataSet ds = new DataSet();
178     da.Fill(ds, "records");
179     DataView Source = ds.Tables["records"].DefaultView;
180     Recordgrid.DataSource=Source;
181     Recordgrid.DataBind();
182
183     myConnection.Close();
184
185     classes.storecart cstorecart = new classes.storecart();
186     ftotal.Text = cstorecart.GetTotal(orderid).ToString();
187 }
188
189 public void CheckOut_Click(object sender, CommandEventArgs e)
190 {
191     // Update the Shopping Cart and then Repopulate the List
192     Response.Redirect("checkout.aspx");
193 }
194
195 public void ContinueShopping_Click(object sender, CommandEventArgs e)
196 {
197     // Update the Shopping Cart and then Repopulate the List
198     Response.Redirect("category.aspx");
199 }
200
201 public void Recalculate_Click(object sender, CommandEventArgs e)
202 {
203     // Update the Shopping Cart and then Repopulate the List
204     UpdateShoppingCartDatabase();
205     PopulateShoppingCartList();
206 }
207
208 void UpdateShoppingCartDatabase()
209 {
210     // Iterate through all rows within shopping cart list
211     for (int i=0; i < Recordgrid.Items.Count; i++)
212     {
213         // Obtain references to row's controls
214         TextBox quantityTxt = (TextBox) Recordgrid.Items[i].FindControl("Quantity");
215         TextBox amountTxt = (TextBox) Recordgrid.Items[i].FindControl("Amount");
216         CheckBox remove = (CheckBox) Recordgrid.Items[i].FindControl("Remove");
217         Label lblProductID = (Label) Recordgrid.Items[i].FindControl("ProductID");
218         Label lblType = (Label) Recordgrid.Items[i].FindControl("Type");
219
220         int productid = int.Parse(lblProductID.Text);
221         char type = char.Parse(lblType.Text);
222         float quantity = float.Parse(quantityTxt.Text);
223         float amount = float.Parse(amountTxt.Text);
224
225         try
226         {
227             classes.storecart cstorecart = new classes.storecart();
228
229             if (quantity == 0 || remove.Checked == true)
230                 cstorecart.RemoveItem(orderid, productid);
231             else
232                 cstorecart.UpdateItem(orderid, productid, quantity, amount);
233         } // try
234         catch
235         {
236             msg.Text = "There has been a problem with one or more of your inputs.";
237         }
238     } // for

```

```

239 }
240
241 public bool Display(string input)
242 {
243     if (input.ToString() == "p") return true;
244     else return false;
245 }
246 public bool Enable(string input)
247 {
248     if (input.ToString() == "d") return true;
249     else return false;
250 }
251
252 #region Web Form Designer generated code
253 override protected void OnInit(EventArgs e)
254 {
255     //
256     // CODEGEN: This call is required by the ASP.NET Web Form Designer.
257     //
258     InitializeComponent();
259     base.OnInit(e);
260 }
261
262 /// <summary>
263 /// Required method for Designer support - do not modify
264 /// the contents of this method with the code editor.
265 /// </summary>
266 private void InitializeComponent()
267 {
268     this.Load += new System.EventHandler(this.Page_Load);
269 }
270
271 #endregion
272 }
273
category.aspx
274 <%@ Register TagPrefix="uc" TagName="footer" Src="../../ascx/footer.ascx" %>
275 <%@ Register TagPrefix="uc" TagName="header" Src="../../ascx/header.ascx" %>
276 <%@ Register TagPrefix="uc" TagName="login" Src="../../secure/login.ascx" %>
277 <%@ Register TagPrefix="uc" TagName="menu_side" Src="../../ascx/menu/menu_side.ascx" %>
278 <%@ Register TagPrefix="uc" TagName="store_menu" Src="../../ascx/store/store_menu.ascx" %>
279 <%@ Page language="c#" Codebehind="category.aspx.cs" AutoEventWireup="false" Inherits="sjsalumni.aspx.user.store.category" %>
280 <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN" >
281 <HTML>
282 <HEAD>
283 <title>
284 <%=PageTitle%>
285 </title>
286 <LINK href="../../css/styles.css" type="text/css" rel="stylesheet">
287 <meta name="GENERATOR" Content="Microsoft Visual Studio 7.0">
288 <meta name="CODE_LANGUAGE" Content="C#">
289 <meta name="vs_defaultClientScript" content="JavaScript">
290 <meta name="vs_targetSchema" content="http://schemas.microsoft.com/intellisense/ie5">
291 </HEAD>
292 <body MS_POSITIONING="GridLayout">
293 <form id="category" method="post" runat="server">
294 <table height="370" cellSpacing="0" cellPadding="0" width="760" align="center" border="0">
295 <tr>
296 <td height="84"><uc:header id="header1" runat="server"></uc:header></td>
297 </tr>
298 <tr>
299 <td vAlign="top">
300 <table height="16" cellSpacing="0" cellPadding="0" width="760" border="0">
301 <tr>
302 <td vAlign="top" align="left" width="140" bgColor="#eaeed3" height="16"><IMG height="16" src="/image/PontG.gif" width="1">
303 <td vAlign="top" align="left" width="15" height="16"><IMG height="16" src="/image/Pontccc.gif" width="15"></td>
304 <td width="385" background="/image/PontTile.gif" height="16"></td>
305 <td vAlign="top" align="left" width="14" height="16"><IMG height="16" src="/image/Pontccc.gif" width="14"></td>
306 <td vAlign="top" align="right" width="206" bgColor="#eaeed3" height="16"><IMG height="16" src="/image/PontG.gif" width="1">
307 </tr>
308 </table>
309 <table cellSpacing="0" cellPadding="0" width="760" border="0">
310 <tr>
311 <td vAlign="top" align="left" width="1" bgColor="#ababab"></td>
312 <td vAlign="top" align="middle" width="139" bgColor="#ffffff"><uc:login id="login1" runat="server"></uc:login><br>
313 <uc:menu_side id="menu_side1" runat="server"></uc:menu_side><br>
314 <br>
315 </td>
316 <td vAlign="top" width="414" bgColor="#ffffff">
317 <table cellPadding="3" width="414" bgColor="#ffffff" border="0">
318 <tr>

```

```

319 <td class="mainTxt">
320 <div class="heading">. Store :: Categories</div>
321 <br>
322 <ASP:DATALIST id="recordddatalist" DataKeyField="id" runat="server" RepeatColumns="2" CellSpacing="2" CellPadding="5" Item
323 <ItemTemplate>
324 <table class="Feature" cellPadding="0" width="185" border="0" cellspacing="0">
325 <tr valign="top">
326 <td colspan="3" class="mainTxt" valign="top"><A class="mainTxt" href="category_products.aspx?ID=<%# DataBinder.Eval(Con
327 </tr>
328 <tr>
329 <td valign="center" bgColor="#b1c6da" colSpan="3" height="1"><IMG height="1" src="../../image/underline.gif" width="100%"
330 </tr>
331 <tr>
332 <td colspan="3" height="5"></td>
333 </tr>
334 <tr valign="top">
335 <td width="60"><A class="mainTxt" href="category_products.aspx?ID=<%# DataBinder.Eval(Container.DataItem, "id") %>"><
336 "photo").ToString())%>></asp:Image></a></td>
337 <td width=<%# Spacing(DataBinder.Eval(Container.DataItem, "photo").ToString())%>><%# DataBinder.Eval(Container.DataItem, "description") %></td>
339 </tr>
340 </table>
341 </ASP:DATALIST></td>
342 </tr>
343 </table>
344 </td>
345 <td class="tdbgclr" valign="top" align="right" width="205"><uc:store_menu id="store_menu1" runat="server"></uc:store_menu
346 </td>
347 <td valign="top" align="left" width="1" bgColor="#ababab"></td>
348 </tr>
349 </table>
350 </td>
351 </tr>
352 <tr>
353 <td><uc:footer id="footer1" runat="server"></uc:footer></td>
354 </tr>
355 </table>
356 </TD></TR></TABLE>
357 </form>
358 </body>
359 </HTML>

```

category.aspx.cs

```

360 using System;
361 using System.Collections;
362 using System.ComponentModel;
363 using System.Data;
364 using System.Data.SqlClient;
365 using System.Configuration;
366 using System.Drawing;
367 using System.Web;
368 using System.Web.SessionState;
369 using System.Web.UI;
370 using System.Web.UI.WebControls;
371 using System.Web.UI.HtmlControls;
372
373 namespace sjsalumni.aspx.user.store
374 {
375     /// <summary>
376     /// Summary description for category.
377     /// </summary>
378     public class category : secure.Security
379     {
380         protected System.Web.UI.WebControls.DataList recordddatalist;
381         protected ascx.menu.menu_side menu_side1;
382         protected ascx.store.store_menu store_menu1;
383         public string pathphoto, pathphoto1;
384         public int userid;
385         public string PageTitle;
386         int communityid, rolelvl;
387
388         private void Page_Load(object sender, System.EventArgs e)
389         {
390             auth();
391             store_menu1.UID = userid;
392
393             PageTitle = Application["PageTitle"].ToString();
394             communityid = int.Parse(Application["CommunityId"].ToString());
395
396             // call GetFieldPreference class
397

```

```

398 classes.functions gpath = new classes.functions();
399 pathphoto1 = gpath.GetFieldPreference(communityid,"path_photo_store_category");
400 pathphoto = Server.MapPath(pathphoto1);
401
402 if (!IsPostBack)
403     BindListRecord();
404 }
405
406 public void auth()
407 {
408     CheckPageRole(Request.RawUrl.ToString(),Request.ApplicationPath.ToString());
409     UpdateLoginControls();
410     UpdateLogoutControls();
411
412     secure.CustomPrincipal cprUser = this.User as secure.CustomPrincipal;
413     if(cprUser != null)
414     {
415         secure.CustomIdentity cidIdentity = cprUser.Identity as secure.CustomIdentity;
416         userid = cidIdentity.SecureId;
417         rolelvl = cidIdentity.RoleLevel;
418     }
419     menu_side1.UID = userid;
420     menu_side1.RID = rolelvl;
421 }
422
423 public void BindListRecord()
424 {
425     SqlConnection myConnection = new SqlConnection(ConfigurationSettings.AppSettings["ConnectionString"]);
426
427     string cmd="SELECT id,category,description,photo from store_category WHERE communityid="+communityid+" AND enable='1' O
428     SqlDataAdapter da = new SqlDataAdapter(cmd, myConnection);
429     DataSet ds = new DataSet();
430     da.Fill(ds, "records");
431     recorddatalist.DataSource = ds.Tables["records"].DefaultView;
432     recorddatalist.DataBind();
433     myConnection.Close();
434 }
435
436 public bool Display(string input)
437 {
438     if (input.ToString() != "")return true;
439     else return false;
440 }
441 public int Spacing(string input)
442 {
443     if (input.ToString() != "")return 3;
444     else return 0;
445 }
446 public string PhotoPath(string input)
447 {
448     string path = pathphoto1+input.ToString();
449     if (input == "") path = pathphoto1+"spacer.gif";
450     return path;
451 }
452
453 #region Web Form Designer generated code
454 override protected void OnInit(EventArgs e)
455 {
456     //
457     // CODEGEN: This call is required by the ASP.NET Web Form Designer.
458     //
459     InitializeComponent();
460     base.OnInit(e);
461 }
462
463 /// <summary>
464 /// Required method for Designer support - do not modify
465 /// the contents of this method with the code editor.
466 /// </summary>
467 private void InitializeComponent()
468 {
469     this.Load += new System.EventHandler(this.Page_Load);
470 }
471
472 #endregion
473 }
474 }

```

category_product.aspx

```

475 <%@ Page language="c#" Codebehind="category_products.aspx.cs" AutoEventWireup="false" Inherits="sjsalumni.user.store.cate
476 <%@ Register TagPrefix="uc" TagName="footer" Src="../../ascx/footer.ascx" %>
477 <%@ Register TagPrefix="uc" TagName="header" Src="../../ascx/header.ascx" %>

```



```

479 <%@ Register TagPrefix="uc" TagName="login" Src="../../secure/login.ascx" %>
480 <%@ Register TagPrefix="uc" TagName="menu_side" Src="../../ascx/menu/menu_side.ascx" %>
481 <%@ Register TagPrefix="uc" TagName="store_menu" Src="../../ascx/store/store_menu.ascx" %>
482 <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN" >
483 <HTML>
484 <HEAD>
485 <title>
486 <%=PageTitle%>
487 </title>
488 <LINK href="../../css/styles.css" type="text/css" rel="stylesheet"><meta name="GENERATOR" Content="Microsoft Visual Studi
489 <meta name="CODE_LANGUAGE" Content="C#">
490 <meta name="vs_defaultClientScript" content="JavaScript">
491 <meta name="vs_targetSchema" content="http://schemas.microsoft.com/intellisense/ie5">
492 </HEAD>
493 <body MS_POSITIONING="GridLayout">
494 <form id="category_products" method="post" runat="server">
495 <table height="370" cellSpacing="0" cellPadding="0" width="760" align="center" border="0">
496 <tr>
497 <td height="84"><uc:header id="header1" runat="server"></uc:header></td>
498 <td vAlign="top">
499 <table height="16" cellSpacing="0" cellPadding="0" width="760" border="0">
500 <tr>
501 <td vAlign="top" align="left" width="140" bgColor="#eaeef3" height="16"><IMG height="16" src="/image/PontG.gif" width="1">
502 <td vAlign="top" align="left" width="15" height="16"><IMG height="16" src="/image/PontccG.gif" width="15"></td>
503 <td width="385" background="/image/PontTile.gif" height="16"></td>
504 <td vAlign="top" align="left" width="14" height="16"><IMG height="16" src="/image/PontcccD.gif" width="14"></td>
505 <td vAlign="top" align="right" width="206" bgColor="#eaeef3" height="16"><IMG height="16" src="/image/PontG.gif" width="1">
506 </tr>
507 </table>
508 <table cellSpacing="0" cellPadding="0" width="760" border="0">
509 <tr>
510 <td vAlign="top" align="left" width="1" bgColor="#ababab"></td>
511 <td vAlign="top" align="middle" width="139" bgColor="#ffffff"><uc:login id="login1" runat="server"></uc:login><br>
512 <uc:menu_side id="menu_side1" runat="server"></uc:menu_side><br>
513 <br>
514 </td>
515 <td vAlign="top" width="414" bgColor="#ffffff">
516 <table cellPadding="3" width="414" bgColor="#ffffff" border="0">
517 <tr>
518 <td class="mainTxt">
519 <div class="heading">. Store :: Products</div>
520 <br>
521 <asp:label id="msg" ForeColor="#FF0000" Runat="server" CssClass="mainTxt"></asp:label>
522 <ASP:DATALIST id="recorddatalist" DataKeyField="id" runat="server" CellSpacing="2" CellPadding="5" ItemStyle-VerticalAlign="T
523 <ItemTemplate>
524 <table class="Feature" cellPadding="0" width="390" border="0" cellspacing="1">
525 <tr class="tablehead" height="15">
526 <td colspan="2" class="tablehead">&nbsp;<strong>::</strong> <A class="tablehead" href="product.aspx?ID=<%=# DataBinder.
527 </tr>
528 <tr valign="top" class="tdbgclr">
529 <td width="80"><A class="Feature" href="product.aspx?ID=<%=# DataBinder.Eval(Container.DataItem, "id") %>"><asp:Image ID=
530 <td class="Feature">
531 <strong>Description:</strong> <%=# DataBinder.Eval(Container.DataItem, "description") %><br>
532 <asp:Label ID="Type" Runat="server" Visible="False" Text="<%=# DataBinder.Eval(Container.DataItem, "type") %>' />
533 <asp:Panel ID="pan_product" Runat="server" Visible="<%=# DisplayProduct(DataBinder.Eval(Container.DataItem, "type").ToString
534 <strong>Price:</strong> <%=# DataBinder.Eval(Container.DataItem, "prix", "{0:c}") %> <i>/<%=# DataBinder.Eval(Container.Da
535 <asp:Label ID="UPrice" Runat="server" Visible="False" Text="<%=# DataBinder.Eval(Container.DataItem, "prix") %>' />
536 <asp:TextBox ID="Quantity" runat="server" Columns="3" MaxLength="3" CssClass="Feature" Text="1" />
537 <asp:linkbutton ID="butt_product" runat="server" commandname="additem" CssClass="Feature" text="Add To Cart" />
538 </asp:Panel>
539 <asp:Panel ID="pan_donation" Runat="server" Visible="<%=# DisplayDonation(DataBinder.Eval(Container.DataItem, "type").ToStri
540 <strong>Amount:</strong> <%=# DataBinder.Eval(Container.DataItem, "prix", "{0:c}") %> <i>/<%=# DataBinder.Eval(Container
541 <asp:TextBox ID="DonateAmount" runat="server" Columns="5" MaxLength="3" CssClass="Feature" />
542 <asp:linkbutton ID="butt_donation" runat="server" commandname="additem" CssClass="Feature" text="Donate" />
543 <div align="right"><b>Thank you!</b></div>
544 </asp:Panel>
545 </td>
546 <td>
547 </tr>
548 </table>
549 </ItemTemplate>
550 </ASP:DATALIST>
551 </td>
552 </tr>
553 </table>
554 <td>
555 <td class="tdbgclr" vAlign="top" align="right" width="205"><uc:store_menu id="store_menu1" runat="server"></uc:store_menu
556 </td>
557 <td vAlign="top" align="left" width="1" bgColor="#ababab"></td>
558 </tr>
559 </table>

```

```

560 </td>
561 </tr>
562 <tr>
563 <td><uc:footer id="footer1" runat="server"></uc:footer></td>
564 </tr>
565 </table>
566 </TD></TR></TABLE>
567 </form>
568 </body>
569 </HTML>

```

category_product.aspx.cs

```

570 using System;
571 using System.Collections;
572 using System.ComponentModel;
573 using System.Data;
574 using System.Data.SqlClient;
575 using System.Configuration;
576 using System.Drawing;
577 using System.Web;
578 using System.Web.SessionState;
579 using System.Web.UI;
580 using System.Web.UI.WebControls;
581 using System.Web.UI.HtmlControls;
582
583 namespace sjsalumni.user.store
584 {
585     /// <summary>
586     /// Summary description for category_products.
587     /// </summary>
588     public class category_products : secure.Security
589     {
590         protected System.Web.UI.WebControls.DataList recorddatalist;
591         protected System.Web.UI.WebControls.Label msg;
592         protected ascx.menu.menu_side menu_side1;
593         protected ascx.store.store_menu store_menu1;
594         public string pathphoto,pathphoto1;
595         int userid,categoryid;
596         public string PageTitle;
597         int communityid,rolelvl;
598
599         private void Page_Load(object sender, System.EventArgs e)
600         {
601             auth();
602             store_menu1.UID = userid;
603
604             PageTitle = Application["PageTitle"].ToString();
605             communityid = int.Parse(Application["CommunityId"].ToString());
606
607             categoryid = int.Parse(Request.Params["ID"].ToString());
608
609             // call GetFieldPreference class
610             classes.functions gpath = new classes.functions();
611             pathphoto1 = gpath.GetFieldPreference(communityid,"path_photo_store_product");
612             pathphoto = Server.MapPath(pathphoto1);
613
614             if (!IsPostBack)
615                 BindListRecord();
616         }
617
618         public void auth()
619         {
620             CheckPageRole(Request.RawUrl.ToString(),Request.ApplicationPath.ToString());
621             UpdateLoginControls();
622             UpdateLogoutControls();
623
624             secure.CustomPrincipal cprUser = this.User as secure.CustomPrincipal;
625             if(cprUser != null)
626             {
627                 secure.CustomIdentity cidIdentity = cprUser.Identity as secure.CustomIdentity;
628                 userid = cidIdentity.SecureId;
629                 rolelvl = cidIdentity.RoleLevel;
630             }
631             menu_side1.UID = userid;
632             menu_side1.RID = rolelvl;
633         }
634
635         public void BindListRecord()
636         {
637             SqlConnection myConnection = new SqlConnection(ConfigurationSettings.AppSettings["ConnectionString"]);
638
639             string cmd="SELECT store_product.id,store_product.product,store_product.description,store_product.type,store_product.photo,store

```

```

640 SqlDataAdapter da = new SqlDataAdapter(cmd, myConnection);
641 DataSet ds = new DataSet();
642 da.Fill(ds, "records");
643 recorddatalist.DataSource = ds.Tables["records"].DefaultView;
644 recorddatalist.DataBind();
645 myConnection.Close();
646 }
647
648 public void AddtoCart_Click(object sender, DataListCommandEventArgs e)
649 {
650     int productid = int.Parse(recorddatalist.DataKeys[e.Item.ItemIndex].ToString());
651
652     Label typeTxt = (Label) recorddatalist.Items[e.Item.ItemIndex].FindControl("Type");
653
654     try
655     {
656         int orderid = 0;
657         int itemid = 0;
658         char type = char.Parse(typeTxt.Text);
659         float amounttext, quantitytext;
660
661         if (type == 'p')
662         {
663             TextBox quantityTxt = (TextBox) recorddatalist.Items[e.Item.ItemIndex].FindControl("Quantity");
664             Label upriceTxt = (Label) recorddatalist.Items[e.Item.ItemIndex].FindControl("UPrice");
665             amounttext = float.Parse(upriceTxt.Text);
666             quantitytext = float.Parse(quantityTxt.Text);
667         }
668         else
669         {
670             TextBox amountTxt = (TextBox) recorddatalist.Items[e.Item.ItemIndex].FindControl("DonateAmount");
671             amounttext = float.Parse(amountTxt.Text);
672             quantitytext = 1;
673         }
674
675         if (amounttext > 0 && quantitytext > 0)
676         {
677             classes.storecart cstorecart = new classes.storecart();
678             cstorecart.AddtoCart(userid, orderid, productid, itemid, quantitytext, amounttext, type);
679             Response.Redirect("cart.aspx");
680         }
681     }
682     catch
683     {
684         msg.Text = "There has been a problem with your input.";
685     }
686 }
687
688 public bool DisplayProduct(string input)
689 {
690     if (input.ToString() == "p") return true;
691     else return false;
692 }
693
694 public bool DisplayDonation(string input)
695 {
696     if (input.ToString() == "d") return true;
697     else return false;
698 }
699
700 public bool Display(string input)
701 {
702     if (input.ToString() != "") return true;
703     else return false;
704 }
705
706 public string PhotoPath(string input)
707 {
708     string path = pathphoto1 + input.ToString();
709     if (input == "") path = pathphoto1 + "spacer.gif";
710     return path;
711 }
712
713 #region Web Form Designer generated code
714 override protected void OnInit(EventArgs e)
715 {
716     //
717     // CODEGEN: This call is required by the ASP.NET Web Form Designer.
718     //
719     InitializeComponent();
720     base.OnInit(e);
721 }
722
723 /// <summary>
724 /// Required method for Designer support - do not modify

```



```

722 /// the contents of this method with the code editor.
723 /// </summary>
724 private void InitializeComponent()
725 {
726     this.Load += new System.EventHandler(this.Page_Load);
727 }
728 }
729 #endregion
730 }
731 }

checkout.aspx
732 <%@ Register TagPrefix="uc" TagName="store_menu" Src="../../ascx/store/store_menu.ascx" %>
733 <%@ Register TagPrefix="uc" TagName="menu_side" Src="../../ascx/menu/menu_side.ascx" %>
734 <%@ Register TagPrefix="uc" TagName="login" Src="../../secure/login.ascx" %>
735 <%@ Register TagPrefix="uc" TagName="header" Src="../../ascx/header.ascx" %>
736 <%@ Register TagPrefix="uc" TagName="footer" Src="../../ascx/footer.ascx" %>
737 <%@ Page language="c#" Codebehind="checkout.aspx.cs" AutoEventWireup="false" Inherits="sjsalumni.aspx.user.store.checkou
738 <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN" >
739 <HTML>
740 <HEAD>
741 <title>
742 <%=PageTitle%>
743 </title>
744 <LINK href="../../css/styles.css" type="text/css" rel="stylesheet">
745 <meta content="Microsoft Visual Studio 7.0" name="GENERATOR">
746 <meta content="C#" name="CODE_LANGUAGE">
747 <meta content="JavaScript" name="vs_defaultClientScript">
748 <meta content="http://schemas.microsoft.com/intellisense/ie5" name="vs_targetSchema">
749 <script language="javascript">
750 function noPostBack(sNewFormAction)
751 {
752     document.forms[0].action = sNewFormAction;
753     document.forms[0].__VIEWSTATE.name = 'NOVIEWSTATE';
754 }
755 </script>
756 </HEAD>
757 <body MS_POSITIONING="GridLayout">
758 <form id="checkout" method="post" runat="server">
759 <table height="370" cellSpacing="0" cellPadding="0" width="760" align="center" border="0">
760 <tr>
761 <td height="84"><uc:header id="header1" runat="server"></uc:header></td>
762 </tr>
763 <tr>
764 <td vAlign="top">
765 <table height="16" cellSpacing="0" cellPadding="0" width="760" border="0">
766 <tr>
767 <td vAlign="top" align="left" width="140" bgColor="#eaeef3" height="16"><IMG height="16" src="/image/PontG.gif" width="1">
768 <td vAlign="top" align="left" width="15" height="16"><IMG height="16" src="/image/PontccG.gif" width="15"></td>
769 <td width="385" background="/image/PontTile.gif" height="16"></td>
770 <td vAlign="top" align="left" width="14" height="16"><IMG height="16" src="/image/PontccD.gif" width="14"></td>
771 <td vAlign="top" align="right" width="206" bgColor="#eaeef3" height="16"><IMG height="16" src="/image/PontG.gif" width="1"
772 </tr>
773 </table>
774 <table cellSpacing="0" cellPadding="0" width="760" border="0">
775 <tr>
776 <td vAlign="top" align="left" width="1" bgColor="#ababab"></td>
777 <td vAlign="top" align="middle" width="139" bgColor="#ffffff"><uc:login id="login1" runat="server"></uc:login><br>
778 <uc:menu_side id="menu_side1" runat="server"></uc:menu_side><br>
779 <br>
780 </td>
781 <td vAlign="top" width="414" bgColor="#ffffff">
782 <table cellPadding="3" width="414" bgColor="#ffffff" border="0">
783 <tr>
784 <td class="mainTxt">
785 <div class="heading">. Store :: Payment</div>
786 <br>
787 <br>
788 <div align="center">
789 To proceed with checkout, you will be taken to a separate
790 <br>
791 payment page which is a 100% secure.
792 <br>
793 <br>
794 <br>
795 <asp:Button ID="Submit" Runat="server" CssClass="button" Text="Continue"></asp:Button></div>
796 </td>
797 </tr>
798 </table>
799 </td>
800 <td class="tdbgclr" vAlign="top" align="right" width="205"><uc:store_menu id="store_menu1" runat="server"></uc:store_menu
801 </td>

```



```

802 <td vAlign="top" align="left" width="1" bgColor="#ababab"></td>
803 </tr>
804 </table>
805 </td>
806 </tr>
807 <tr>
808 <td><uc:footer id="footer1" runat="server"></uc:footer></td>
809 </tr>
810 </table>
811 </TD></TR></TABLE></form>
812 </body>
813 </HTML>

checkout.aspx.cs
814 using System;
815 using System.Collections;
816 using System.ComponentModel;
817 using System.Data;
818 using System.Data.SqlClient;
819 using System.Configuration;
820 using System.Drawing;
821 using System.Web;
822 using System.Web.SessionState;
823 using System.Web.UI;
824 using System.Web.UI.WebControls;
825 using System.Web.UI.HtmlControls;
826
827 namespace sjsalumni.aspx.user.store
828 {
829     /// <summary>
830     /// Summary description for checkout.
831     /// </summary>
832     public class checkout : secure.Security
833     {
834         protected System.Web.UI.WebControls.Button Submit;
835         protected ascx.menu.menu_side menu_side1;
836         protected ascx.store.store_menu store_menu1;
837         public int userid,orderid;
838         public string PageTitle;
839         int communityid,rolelvl;
840
841         private void Page_Load(object sender, System.EventArgs e)
842         {
843             auth();
844             store_menu1.UID = userid;
845
846             PageTitle = Application["PageTitle"].ToString();
847             communityid = int.Parse(Application["CommunityId"].ToString());
848
849             // Get CartID (orderid)
850             classes.storecart cstorecart = new classes.storecart();
851             orderid = cstorecart.GetCartId(userid);
852
853             char cfixd = 'Y'; // to disable "back" button (in 2CO page)
854             string sid = "210959";
855             string total = cstorecart.GetTotal(orderid).ToString();
856             string cart_order_id = orderid.ToString();
857             Submit.Attributes.Add("onclick", "noPostBack('https://www2.2checkout.com/2co/buyer/purchase?sid="+sid+"&total="+total+"&car
858             )
859
860             public void auth()
861             {
862                 CheckPageRole(Request.RawUrl.ToString(),Request.ApplicationPath.ToString());
863                 UpdateLoginControls();
864                 UpdateLogoutControls();
865
866                 secure.CustomPrincipal cprUser = this.User as secure.CustomPrincipal;
867                 if(cprUser != null)
868                 {
869                     secure.CustomIdentity cidIdentity = cprUser.Identity as secure.CustomIdentity;
870                     userid = cidIdentity.SecureId;
871                     rolelvl = cidIdentity.RoleLevel;
872                 }
873                 menu_side1.UID = userid;
874                 menu_side1.RID = rolelvl;
875             }
876
877             #region Web Form Designer generated code
878             override protected void OnInit(EventArgs e)
879             {
880                 //
881                 // CODEGEN: This call is required by the ASP.NET Web Form Designer.

```

```

882 //
883 InitializeComponent();
884 base.OnInit(e);
885 }
886
887 /// <summary>
888 /// Required method for Designer support - do not modify
889 /// the contents of this method with the code editor.
890 /// </summary>
891 private void InitializeComponent()
892 {
893     this.Load += new System.EventHandler(this.Page_Load);
894 }
895 }
896 #endregion
897 }
898 }
index.html
899 <HTML>
900 </HTML>
anypage.html
901 <HTML>
902 </HTML>

```

Figure 24: The conditional edges as proposed by the UMLT technique

Figure 25: The Fork Node in a MHDG Diagram

REFERENCES

Bedi, S., Schroeder, P. (2004). Observations on the Implementation and Testing of Scripted Web Applications. *In the Proceedings of the Sixth IEEE International Workshop on the Web Site Evolution (WSE'04)*, Wisconsin, Milwaukee, pp. 20-27.