AUTOMATED NAVIGATION MODEL EXTRACTION FOR WEB LOAD TESTING

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF INFORMATICS
OF
THE MIDDLE EAST TECHNICAL UNIVERSITY

BY

İSMİHAN REFİKA KARA

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE
IN
THE DEPARTMENT OF INFORMATION SYSTEMS

DECEMBER 2011

AUTOMATIC NAVIGATION MODEL EXTRACTION FOR WEB LOAD TESTING

Submitted by **İSMİHAN REFİKA KARA** in partial fulfillment of the requirements for the degree of **Master of Science in InformationSystems, Middle East Technical University** by,

Prof. Dr. Nazife BAYKAL _____
Director, **Informatics Institute**

Prof. Dr. Yasemin YARDIMCI ÇETİN _____
Head of Department, **Information Systems**

Assist. Prof. Dr. Aysu BETİN CAN _____
Supervisor, **Information Systems, METU**

Examining Committee Members

Prof. Dr. Semih BİLGEN _____
Electrical and Electronics Engineering, METU

Assist. Prof. Dr. Aysu BETİN CAN _____
Information Systems, METU

Assist. Prof. Dr. Bülent Gürsel EMİROĞLU _____
Computer Engineering, Başkent University

Assist. Prof. Dr. Erhan EREN _____
Information Systems, METU

Assist. Prof. Dr. Banu GÜNEL HACIHABİBOĞLU _____
Information Systems, METU

Date: 29.12.2011

I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.

Name, Last name : İsmihan Refika KARA

Signature : _____

ABSTRACT

AUTOMATIC NAVIGATION MODEL EXTRACTION FOR WEB LOAD TESTING

Kara, İsmihan Refika
M.S., Department of Information Systems
Supervisor: Assist. Prof. Dr. Aysu BETİN CAN

December 2011, 48 pages

Web pages serve a huge number of internet users in nearly every area. An adequate testing is needed to address the problems of web domains for more efficient and accurate services. We present an automated tool to test web applications against execution errors and the errors occured when many users connect the same server concurrently. Our tool, called NaMoX, attains the clickables of the web pages, creates a model exerting depth first search algorithm. NaMoX simulates a number of users, parses the developed model, and tests the model by branch coverage analysis. We have performed experiments on five web sites. We have reported the response times when a click operation is eventuated. We have found 188 errors in total. Quality metrics are extracted and this is applied to the case studies.

Keywords: Load Test, Web applications, Branch Coverage Analysis, Model Based Testing

ÖZ

WEB LOAD TESTLERİ İÇİN OTOMATİK OLARAK MODEL ÇIKARTILMASI

Kara, İsmihan Refika
Yüksek Lisans, Bilişim Sistemleri Bölümü
Tez Yöneticisi: Yrd. Doç. Dr. Aysu BETİN CAN

Aralık 2011, 48 sayfa

Web sayfaları pek çok internet kullanıcısına hitap ettiklerinden, bu uygulamaların yeterli seviyede test edilmeleri gerekmektedir. Bu sayede hata oranları azaltılarak, kullanıcılara daha verimli ve daha doğru hizmet verilebilir. Bu tezde öngörülmemiş çalışma hataları ve birden fazla kullanıcının aynı anda web sunucularına bağlantı kurmaları sonucu oluşan hataları içeren, web sayfalarını otomatik olarak test eden bir araç sunacağız. Aracımızın ismi NaMoX'tur. NaMoX, öncelikle web sayfalarının tıklanabilirlerine ulaşarak ve derinlik bilgisine dayanarak, derinlik öncelikli algoritma ile bir model oluşturur. Çıkartmış olduğu bu modeli kullanarak, birden fazla kullanıcıyı simüle eder ve dal kapsama analizi yaparak, web sayfasını yük testi için hazırlar. Bu calismada beş tane örnek web sayfası üzerinde deneyler yapıldı, simüle edilen her bir kullanıcı için web sitesinin tıklanabilirlerine erişilme yanıt süreleri, ve karşılaşılan hatalar bulundu. Toplamda 188 adet hata bulundu. Kalite metrikleri çıkartılarak, örnek sayfalara kalite karşılaştırması uygulandı.

Anahtar Kelimeler: Yük Testi, Web Uygulamaları, Dal Kapsama Analizi, Model Tabanlı Test

This thesis is dedicated to:

My Grand Parents…

ACKNOWLEDGEMENTS

I would like to thank my supervisor Assist. Prof. Dr. Aysu Betin Can for her guidance, patience and the discipline that she provided throughout my thesis.

I would like to present my thanks to the faculty and the staff of Informatics Institute, especially Ali Kantar, for their support to my education.

For providing scholarship, I would also thank the Scientific and Technological Research Council of Turkey (TÜBİTAK).

Finally, I would like to render my thanks to my father Himmet, my mother Canan, my sister Evdegül, my brother Yasin, and my fiancé Ali for their patience and apprehension.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF SYMBOLS

AJAX	Asynchronous JavaScript and XML
ASP	Active Server Pages
DOM	Document Object Model
GB	Giga Byte
GHz	Giga Hertz
HTML	HyperText Markup Language
HTTP	Hypertext Transfer Protocol
METU	Middle East Technical University
PHP	Hypertext Preprocessor
QoS	Quality of Service
RAM	Random Access Memory
UML	Unified Modeling Language
URL	Uniform Resource Locator
XML	EXtensible Markup Language
XPATH	XML Path Language
XSL	EXtensible Stylesheet Language

CHAPTER 1

INTRODUCTION

With the rapid growth of internet technologies, web applications become very pervasive nearly in every area such as banking, education or government agencies. The convenience gained the popularity of web applications. However, internet users encounter some unexpected errors during their business on the web. These errors attenuate the reliability of the web domain and the web site loses its users. Hence, an adequate testing is needed to address the problems of web domains for more efficient and accurate services.

There are several test techniques used to prevent the inaccuracies. Load testing is one of the test techniques that ensures the application can operate under a specific load. In web application context, it is considerably important because there is an intense data transfer and high number of users. Several works on load testing have focused on three topics [1]:

- User writes test scenario using scripts using XML or Jython (e.g. TestMaker[2], Grinder[3], LoadSim[4], JMeter[5]).

- User enters the scenario using a graphical objects representing some built-in interactions (like JMeter[5], LoadTest[6]).
- User interacts with a browser and the interactions are recorded with a capture tool. (Like DieselTest[7], OpenSTA[8], LoadTest[6]). Some applications also combine the last two items (LoadTest[6]).

The most consequential disadvantage of these three approaches is the coverage. The coverage analysis is left to the user, which begets to overlook some navigation and reduce the testing adequacy. These tools mostly focus on the number of the concurrent users. The content of the page is keeping in the background although it is a considerable topic in testing.

Furthermore, most of these tools do not address the scripts inside the HTML codes. These tools do not realize and handle the scripts. However, scripting languages are becoming widespread and used in most popular pages, such as Facebook or Google.

In this thesis, we present an automated tool, called NaMoX, to create a navigation model for load testing. NaMoX checks execution errors and the errors composed when many users connect the same server concurrently.

NaMoX addressed the coverage problem by generating the test cases using a model extracted from the clickables, defined as hyperlinks and elements that have OnClick events, of the web page using branch coverage analysis. Branch coverage ensures that each possible branch of the model is executed at least once. NaMoX also crawls text boxes, list boxes and checkboxes to give specific values to these input boxes, using a gray box approach.

Taking the gray box approach, NaMoX overcomes the script issue by executing the test scenarios through a browser. This approach enables NaMoX not to discriminate scripts from static contents. NaMoX uses Selenium [9] to open web browser and to perform click operations on that browser. Selenium is a functional and acceptance testing tool for Web applications that provides an API to use several web browsers such as FireFox[10]. NaMoX firstly extracts the clickables from the source code and builds a transition graph

of the web application. Selenium, which can test client-side functionality implemented in JavaScript [9], is used for performing click operations of published clickables.

We have performed experiments on five commercial web sites, in JavaScript, ASP and PHP pages. We have reported the response times when a click operation is eventuated. We have found 188 errors in total.

1.1 Overview

There are five chapters in this thesis. Chapter 1 introduces some test techniques and focuses on severity of load testing. The contributions of our tool are described. The case studies and the results are explained briefly.

Chapter 2 deals with the background knowledge and related works about load testing and crawling algorithms.

Chapter 3 proposes the phases of the project in detail. The methodology and system architecture is explained in this chapter. We give technical information about NaMoX and our model's perspective.

Chapter 4 analyses the experiments in our study. The results in five web sites are described. The test environments and the durations for modeling and running tests are explained.

The final chapter is the conclusion part of the work. This chapter summarizes the study and the outputs of the thesis. The future works are also told in this chapter.

CHAPTER 2

LITERATURE REVIEW AND BACKGROUND

This chapter puts forth the previous work done in the literature on the automatic navigation model based testing and web testing. Section 2.1 provides an overview of related works about web testing. Section 2.2 focuses on literature review on model based testing. Finally, Section 2.3 mentions the technologies used in NaMoX.

2.1 Web Testing

Web testing includes any activity evaluating an attribute or capability of a web application and determining that it meets its required results.

There are many testing techniques used to meet the requirements of a system including web systems. One of them is load testing which ensures the web site runs under the expected load. The response times and the failures should be tested at different load levels [11].

It is critical for an end user especially customers that web applications are fast and reliable; therefore load testing is an important task for making sure a web site meets the requirements for optimizing applications different components for end users. [12].

The Apache JMeter [5] is a load test tool which measures performance of a web application. The JMeter takes the URLs and input values for input boxes values manually. It caches (i.e. records) the test and replays it to take the test result. Test results can be analyzed and replayed offline. However, JMeter is not supported by real web browsers. Users cannot see the browsers while running or after the test. Furthermore, it does not support JavaScript codes which are really very popular item in web programming. On the other hand, NaMoX finds the clickables automatically by the program instead of recording. These clickables are used for creating a navigational model of the web application as a state machine. Based on that state machine, test cases are generated with a number of simulated users. NamoX also have a support on JavaScript.

BrowserMob [13] also provides web site load testing. It takes Selenium [14] (described in Section 2.3.1 in details) scripts, and runs the tests according to these scripts. Selenium records user actions and builds test scripts, automates browsers. It performs reusable scripts in many programming languages that can be later executed. BrowserMob also supports JavaScript, which is different than JMeter that cannot handle scripts. BrowserMob uses real browsers, and reports the network performance of the tested web page. Although there are many similarities, the main difference of NaMoX from BrowserMob is that NaMoX creates the Selenium scripts automatically. Therefore, NaMoX does not use record and replay algorithm.

Marchetto et al [15] proposed a case study based on comparison of web testing techniques applied to AJAX web applications. It aims to find the answers of these questions:

- What is the effectiveness in revealing faults of each Web testing technique?
- What is the effort required to apply each Web testing technique?

The results show that the state based techniques are more successful on detecting unexpected errors. But the effort is higher especially in preparation phases. Following the results of this study we chose the model in NaMoX to be a state based model to detect more errors.

Menascé [11] describes the quality of service factors of load testing and how it works. This study specified QoS factors as:

- Availability,
- Response time.

This study shows the relationship between the number of users and response times. As user number increases, response time also increases. When a certain number of users in the system is reached, there is an exact increase in response time.

The experiments (explained in Chapter 4 in details) done using NaMoX show the direct proportion between response time and number of users as explained in Menascé's study.

Mosberger et al. [16] have a study on measuring web server's performance, called httperf. httperf run on the client and produce an HTTP workload on the server composed of three parts:

- The core HTTP engine: provides connection
- Workload generation: ensures load constitution.
- Statistics collection: creates a performance graph by collecting statistics of loads.

NaMoX does not have the ability to measure the performance of the whole system, however it generates some load and collects the response times of accessing to each clickable as described in the later chapters of this thesis.

A popular product for industry-strength load testing is Mercury Interactive's LoadRunner [17]. It uses a script-driven approach and increases usability by a visual editor for end-user scripts. This end-user scripts run on a load engine that takes care of load balancing and monitoring automatically. Most current load testing tools operate in a

similar manner to LoadRunner. However, LoadRunner does not provide a model-based solution.

Dirk Draheim et al. [18] has a study on realism in the simulation of user behavior. According to this study, a load test is applicable when the virtual users' behaviors are similar to the actual users, otherwise virtual users' behaviours can generate inconsistent results. However, the manual test case implementation is time consuming and difficult. Most of the current load testing tools supports the composition of test cases consisting of a fixed sequence of operations.

NaMoX's created model is appropriate for realistic situations. This model is constructed according to a logical order. The given initial URL forms the starting point and this URL's clickables are extracted and they are traversed according to depth-first search algorithm.

Scott Barber [19] has a study on the performance of load test tools. In this study, the considerations are divided into three categories:

- User psychology
- System considerations
- Usage considerations

User psychology is the most often ignored consideration. However, Dirk Draheim et al. [18] mentioned, realistic use behavior plays a material role and is an critical evaluation criteria in performance of load test tools.

As it is mentioned, NaMoX's model is occurred according to a logical order. Therefore, this model is created suitable for user psychology.

System considerations [19] decide the performance that the system can handle within the given parameters. System considerations include the following:

• System hardware
• Network and/or Internet bandwidth of the system
•Geographical replication
• Software architecture

These system considerations are included in the NaMoX's future work told in Chapter 5 in details. The experiments should be done in a strong performance server. According to these considerations, NaMoX will decide the thread number of load generation.

Usage considerations changes for each web application. For instance, an application for reading news or an application accessing some data from a company database requires different performance. A graphical application may be slower and some applications selecting data may be faster.

NaMoX eliminates this problem by using the Selenium [14] library in its load test module. Selenium improves performance and optimizes the test cases. NaMoX uses XPath [20] as the unique identifier and Selenium also improves the performance of the XPath processing.

Daniel A. Menascé [11] predicts web applications' performance at any load levels.

- *Nvu* = number of virtual users.
- *Nc* = number of concurrent requests a Web site is processing.
- *Z* = average think time, in seconds.
- *R* = average response time for a request, in seconds.
- *Xo* = average throughput, in requests per second.

The study gets the following relationship:

Nvu = [R(Nc) + Z] × Xo (Nc).

(Equation 2.1)

The metrics [21] are also very important for testing tools, determines application's performance and provides specific information on system errors. Some metrics are explained in below:

Connections: This test measures the number of refused connections while the load test module is running. A failed connection may cause from a busy server that cannot handle new requests or memory may not be adequate. It also may mean that the user sent malformed data to the server.

NaMoX reports HTTPExceptions and TimeOutExceptions. Connection failures may be because of the some http errors such as HTTP 408 Request Timeout Error, HTTP 504 Gateway Timeout Error, HTTP 416 Requested Range Not Satisfiable Error or TimeOutExceptions.

Throughput: Throughput is the metric of sum of response data size divided by the number of seconds in the reporting interval. This is an important metric that controls the application and its server connection is working properly. As the load in the web application increases, the throughput also increases [21].

HTTP 500 Internal Server Error is one of the failures that NaMoX handles and can be used as a throughput metric error. When the number of virtual user is over loaded, this kind of failures may be occurred.

Hits per Second: As the hits per seconded is increased, the application will handle more request in a second [21]. Hits per second metric explains if there is a possible scalability issue with the application.

Pages per Second: Pages per second measures the number of pages requested from the application per second. The more the page per second, the more work the application is doing per second [21].

NaMoX uses Selenium [14] that automates user activities and creates scripts according to the activities. According to the Selenium's performance, the pages per second metric will be changed.

2.2 Model Based Testing

Model based testing develops test cases from an extracted model of a system under test. Utting et al.[22] investigate model based testing tools' approaches and understanding the issues of integrating model based testing into a software development process. This study classified the approaches in model based testing tools. According to these classifications, NaMoX has online test case generation including arc coverage algorithm.

A. Pretschner et al. [23] compares the coverage and number of detected errors of model based tests with hand crafted tests. The results showed that the tests using a model detect more failures than hand crafted tests. This work also adduce that there is strong correlation between coverage and failure detection. In NaMoX, we extracted a navigation model on clickables of the HTML source code. On this model, we applied the branch coverage techniques for detection of more exceptions according to this study.

Kung et al [24] generates tests based on multiple models of the web applications. These models are: Object Relation Diagrams, Object State Diagrams, a Script Cluster Diagram, and a Page Navigation Diagram. This study uses white box test technique, assuming the source code is available. NaMoX uses gray box technique for composing the state diagram.

There are several example studies for testing object oriented programs using finite state machines. For instance, Kung et al. [25] extract the model from the code using symbolic execution. On the other hand Turner and Robson [26] derive the FSM from the design of classes. NaMoX differs from these studies, composes the model from the clickables of the HTML source code.

Crawljax [27] is the first web crawler targeted for Ajax applications. Given URL and a depth, Crawljax firstly finds the defined clickables of the web page. The clickables include the links on the source code, and the elements that have ‘onClick’ and ‘onMouseOver’ events. Crawljax provides an automatic click of the clickable objects. At the time of automatically clicking, a graph with DOM states is being created dynamically.

The ATUSA [28] is used with the Crawljax [27] project as a test generator tool. It uses a dynamic analysis to construct a model of an application’s state graph. ATUSA has detected six types of failures: three of them are the generic plugins, and the rest three through the application-specific plugins. However, it does not ensure one hundred percent coverage of state machine.

2.3 Technologies Used

This section explains the technologies that are used in NaMoX's background. In Section 2.3.1, a browser automation tool Selenium [14] will be described. Section 2.3.2 explains HTML Agility Pack [29] which is used for parsing the html source code of a web site. Section 2.3.3 gives information about Xpath [20] that is used as a unique property of clickables.

2.3.1 Selenium

Selenium [14] is a recording tool which records user actions for building test scripts and automating browsers. It performs reusable scripts in one of many programming languages that can be later executed. It has support of multiple browser platforms such as Mozilla FireFox [10], Google Chrome [30] or Internet Explorer [31], and several languages such as Java, JavaScript, Ruby, PHP, Python, Perl, or C #. It also provides an infrastructure for calling Selenium inside the programming languages.

We use Selenium not only for executing test cases while performing load testing but also for collecting clickable items in a web page while building the navigation graph of a web site. There are four important method of Selenium library used in NaMoX:

- Selenium.Click(XpathOfTheClickable) : implements the click operation. This function is used in executing clickables in model extraction. Also, it is managed in running the load test module.
- Selenium.Open(UrlOfTheClickable) : implements opening the given URL. This function is also used in executing clickables in model extraction and in load test module as Selenium.Click.
- Selenium.GetHtmlSource(): returns the HTML source code of the URL in which the browser is.
- Selenium.GetLocation(): returns the URL in which the browser is.

2.3.2 HTML Agility Pack

HTML Agility Pack [29] is a .NET code library which models and parses HTML documents, supporting XPath.

NaMoX needs to get clickables and input boxes from the HTML source code. For this purpose, firstly the HTML source code is discovered from a given URL with Selenium, and it is parsed by HTML Agility Pack.

The originated HTML source code is firstly loaded on an HtmlDocument class in the HTML Agility Pack library as shown in below:

```
HtmlAgilityPack.HtmlDocument htmldoc = new
HtmlAgilityPack.HtmlDocument();
htmldoc.LoadHtml(htmlSource);
```

The loaded html document is decomposed in html nodes by SelectNodes(XPath) method presented as in the below example for input boxes:

```
foreach (HtmlAgilityPack.HtmlNode node in
htmldoc.DocumentNode.SelectNodes("//input"))
{
    Input input = new Input();
    input.InputClass = node.Attributes["class"].Value;
    input.InputId = node.Attributes["id"].Value;
    input.InputName = node.Attributes["name"].Value;
    input.InputType = node.Attributes["type"].Value;
    input.InputValue = node.Attributes["value"].Value;
    input.InputXPath = node.XPath;

}
```

2.3.3 XML Path Language

Clark et al. [32] defines XPath as a syntax and semantics for functionality between XSL Transformations and XPointer. XPath aims to handle the part of XML document and manipulates strings, numbers and booleans.

In NaMoX, we used XPath as a unique attribute of clickables and input boxes. Since HTML codes do not have to have identification (id) for each element, we use XPath as the primary key of these classes.

According to the example HTML source code in Figure 2.1, there are two clickables. link1 and button1.

```
<html> <body>
      Main Page
      <a href = "link1.htm"> link1 </a>
      <INPUT TYPE=BUTTON ID=button1
      OnClick="window.location='button1.htm'"/>
</body> </html>
```

Figure 2.1: Example HTML source code

The XPaths of the clickables in Figure 2.1 are:

- link1 : body[1]//a[1] (selects the first `a` of the `body`)
- button1 : body[1]//input[1] (selects the first `input` of the `body`)

CHAPTER 3

NaMoX

In order to solve problems specified in Chapter 1, we developed a tool, called NaMoX. We aim to extract a navigation model of a web application for web load testing. To the best of our knowledge, there is no open source load testing tool that takes test cases from a model. To show that our model extraction enables test case generation for load testing, we performed our own load test runner module. We also propose NaMoX as an automated tool to find unhandled exception errors during creation of the model and running the load test partition. NaMoX adapts depth-first search algorithm to compose the navigation model. Moreover, the created model's coverage is designated by the clickables, including hyperlinks and elements that have OnClick events. Test cases are automatically generated employing the model using branch coverage analysis technique. NaMoX uses gray box technique for composing the state diagram.

In this chapter, the study is presented in details. Firstly, the methodology and algorithm of the study is explained. Then, the design and implementation of NaMoX is describes in terms of system architecture.

3.1 Methodology

This section will be described in two parts:

- Creating the state graph
- Creating the test sequences

Section 3.1.1 describes the formation of the state graph. Section 3.1.2 deals with the composed test sequences for web load testing.

3.1.1 Creating the State Graph

NaMoX's primary work is creating a model from a seed URL of a web application using states and transitions. In this model, states are combination of the clickables and HTML source code that contain these clickables. The clickables are defined as hyperlinks and elements that have OnClick events. These clickables are detected from the HTML source code through a crawling process. Clickables' attributes are:

- Clickable Xpath
- Clickable URL
- Clickable Id
- Clickable Type
- Clickable Event

explained in Section 3.2.1.1 in details.

State graph's transitions consist of the current state, the clickable, and the created next state when the clickable performed in the current state. Transitions also include the go back clickables while returning back to the previous state.

Figure 3.1 shows a small sized example HTML code. In this source code, there are two clickables. One of them is link1 which is a hyperlink, second one is button1 which has an OnClick event.

```
<html> <body>

      Main Page

      <a href = "link1.htm"> link1 </a>

      <INPUT TYPE=BUTTON ID=button1
      OnClick="window.location='button1.htm'"/>

</body> </html>
```

Figure 3.1: Example HTML source code including clickables

Figure 3.2 shows the state flow graph visualization of the extracted model when depth equals to 1 for Figure 3.1. Graph's vertices include the states; and arcs are formed from transitions. We use XPath [20] to denote the clickables in a page because it is a unique attribute of the clickable.

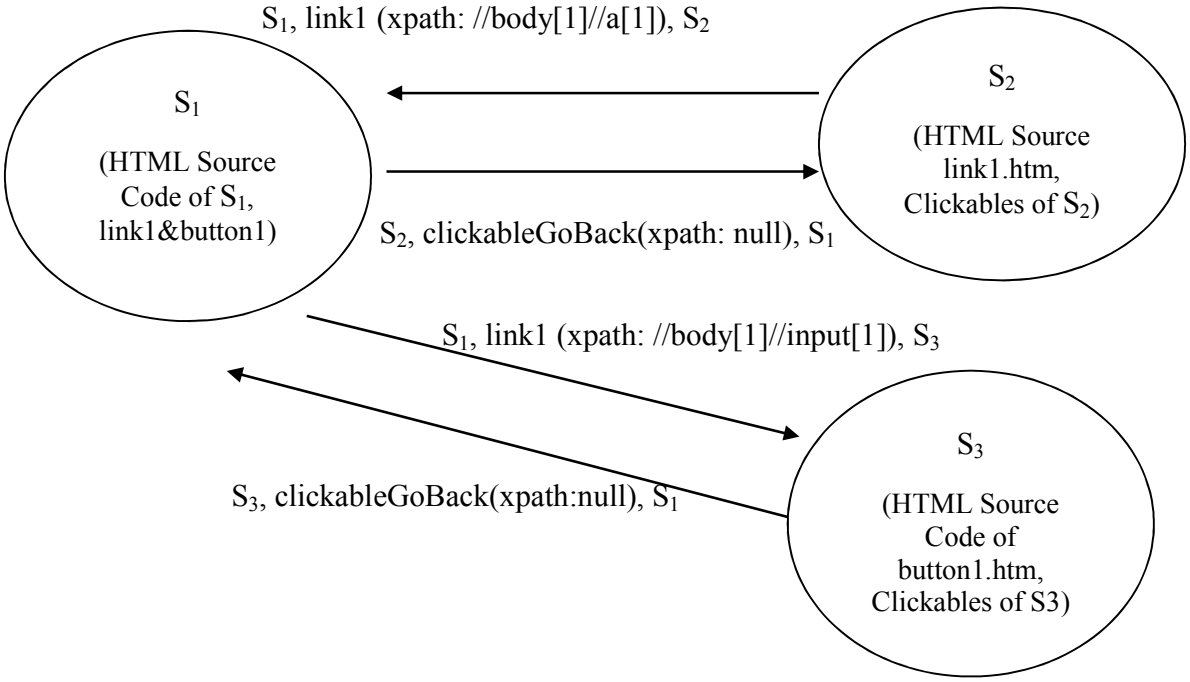

Figure 3.2: State flow graph visualization

We crawl the web pages in a depth first manner and while crawling we build the state graph as stated above. After this model is extracted, it is being preserved for reusing explained in Section 3.1.2 in details. The transitions of the graph are being written on a text file. Figure 3.3 shows an example of the graph file. To increase readability, the URL*id number of state* is written instead of state. NaMoX writes the states to another state files, giving the ids as file names, and matches this state files with the id number of state in the graph file.

```
FROM=> http://www.gucci.com/int/home *1*
WITH=>Xpath://body[1]//span[1]//div[2]//header[1]//h1[1]//a[1]
Event:href
TO=> http://www.gucci.com/int/home *2*
```

Figure 3.3: Example graph file

While composing the graph, there can be errors in execution level. As told in Chapter 4, we call this type of errors as 1st Level Errors. These error types are shown in Figure 3.4.

400	**Bad Request**
401	**Unauthorized**
402	**Payment Required**
403	**Forbidden**
404	**Not Found**
405	**Method Not Allowed**
406	**Not Acceptable**
407	**Proxy Authentication Required**
408	**Request Timeout**
409	**Conflict**
410	**Gone**
411	**Length Required**
412	**Precondition Failed**
413	**Request Entity Too Large**
414	**Request-URI Too Large**

415	**Unsupported Media Type**
416	**Requested Range Not Satisfiable**
417	**Expectation Failed**
500	**Internal Server Error**
501	**Not Implemented**
502	**Bad Gateway**
503	**Service Unavailable**
504	**Gateway Timeout**
505	**HTTP Version not supported**

Figure 3.4: HTTP Errors [33]

3.1.2 Creating the Test Sequences for Load Testing

As stated in Section 3.1.1, the model build using crawling is preserved for reusing. Once the graph is generated, testers can use it to load test the application anytime. In the graph file text, the transitions are navigation actions as shown in Figure 3.3. These transitions are formed from the branches of the state machine. NaMoX reads this graph file text and runs a branch coverage test from that model with a number of threads concurrently. In order to perform that, we could add some other coverage criteria, such as switch coverage, path coverage or random testing and let the tester choose for different levels of test adequacy.

According to the specified coverage criteria, NaMoX creates test sequences. It firstly reads the graph file and loads the model. Based on the source code in the state file text, NaMoX crawls the textboxes, listboxes and checkboxes from the HTML source code. To provide the input values of these extracted input boxes, the tool expects the user to

provide an input values file. This input file includes the URL, the Xpath of the widget that takes the input, and the value of the input that the user wants to assign (see Figure 3.5).

```
http://localhost/test.html  //body[1]//input[2]   Test
```

Figure 3.5: Input Values File

Based on the thread count given by the user, the generated test sequences are executed concurrently simulating many concurrent users on a real browser. NaMoX captures the errors shown in Figure 3.4. NaMoX also records response times for each thread.

3.2 Design and Implementation

The system architecture of NaMoX is shown in Figure 3.6. NaMoX consists of four main components: Crawler, Load Generator, Selenium and HTML Agility Pack. Below we explain each of these components in detail. Selenium and HTML Agility Pack are the available free source libraries [14, 29]. Therefore their usage is explained in Crawler and Load Generator modules.

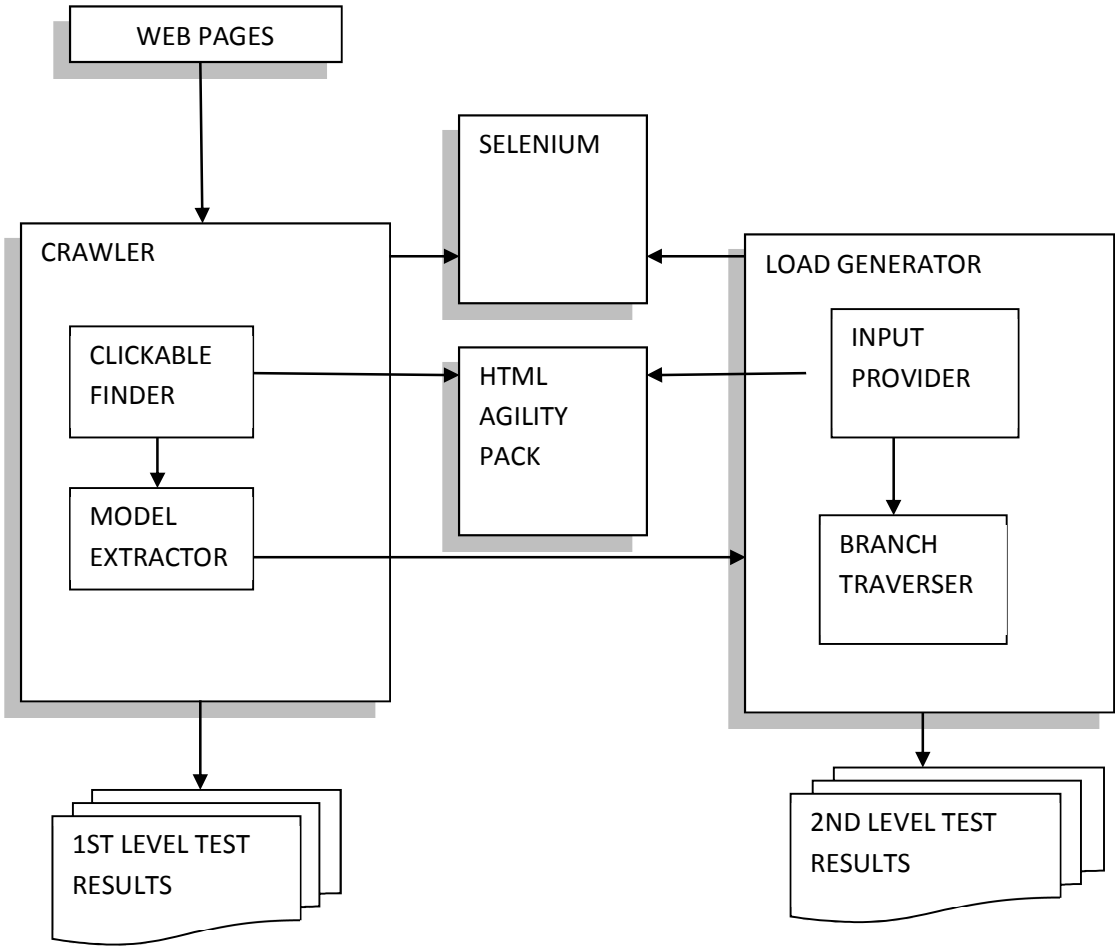

Figure 3.6: System architecture of NaMoX

3.2.1 Crawler

Crawler constitutes the main component of NaMoX. This component consists of two modules. The first module is Clickable Finder. Clickable Finder is responsible for finding the items in the document that has an OnClick event attached to it and the hyperlinks, i.e. the clickables. The second module is the model extractor. This component builds the model, its states and transitions according to a depth first search algorithm.

3.2.1.1 Clickable Finder

Clickable Finder is responsible for finding the defined clickables. We define clickables as hyperlinks and the elements having OnClick events. These specified clickables are extracted from the source code by using HTML Agility Pack [29] library which parses the source code. The input boxes such as textbox, listbox or checkbox are also extracted using this library (explained in Section 3.2.2.1 in details).

The clickables have five important attributes:

- Clickable Xpath: It describes the Xpath of the clickable. For instance; //body[1]//input[2] means the clickable in the first body tag, under the second input tag.
- Clickable URL: It defines the URL in which that clickable is.
- Clickable Id: It is the id of the clickable if it is determined in the source code. It may be null if it is not defined.
- Clickable Type: If the clickable is a hyperlink, then the type is “href”; else if the clickable is another element that has onClick event, then the clickable type refers to “OnClick”.
- Clickable Event: If the clickable is a hyperlink, then the type is “href”; else if the clickable is another element that has onClick event, then the clickable type refers to “click”.

Getting clickables algorithm is shown in below:

```
procedure GetClickables(htmlSourceCode)
       HtmlAgilityPack.HtmlDocument htmldoc ← htmlSourceCode
       foreach(HtmlAgilityPack.HtmlNode link in
htmldoc.SelectNodes("//*[@href]"))
              clickable.clickableXpath ← link.XPath
              clickable.clickableUrl ← selenium.GetLocation
              clickable.clickableId ← link.Id
              clickable.clickableType ← "href"
              clickable.clickableEvent ← "href"
       end foreach
       foreach(HtmlAgilityPack.HtmlNode click in
htmldoc.SelectNodes("//*[@onClick]"))
              clickable.clickableXpath ← click.XPath
              clickable.clickableUrl ← selenium.GetLocation()
              clickable.clickableId ← click.Id
              clickable.clickableType ← "OnClick"
              clickable.clickableEvent ← "click"
       end foreach
end procedure
```

According to this algorithm, Html Agility Pack [29] parses the HTML source code. The hrefs and onClicks are collected from the code and assigned to an HtmlNode element. The HtmlNode element's XPath [32] assigned to the clickableXPath and id assigned to the clickableId. The clickableUrl is taken from Selenium.GetLocation() method. This method returns the URL in which the browser is, described in detail in Section 2.3.1.

NaMoX uses static HTML source codes while getting clickables. Server side scripts poses no problemfor NaMoX because source codes generates them. As Selenium [14]

simulates the click operation of the clickable, the client side scripts also pose no problem.

3.2.1.2 Model Extractor

NaMoX's primary task is creating a model from the HTML source code of a web page. In this model, states are the combinations of the clickables and HTML source codes. On the other hand, transitions include the current state, the clickable, and the created next state when the clickable performed in the current state. Transitions also include the go back clickables while returning back to the previous state. NaMoX does not use browser's goBack function, because in dynamic pages, URL may not change although the content changes. NAMoX prevents this case by holding the previous state and returns back to that state during traversal.

While creating the model, there are two parameters derived from the user; the initial URL, and the depth. The entered initial URL defines the start point of the crawling. The entered crawl depth shows the levels to be followed. It means that in how many levels the clickables are continued to be detected. According to the depth, the web page's clickables are traversed recursively using Selenium library using Click and Open functions explained in Section 2.3.1.

Creating model algorithm is shown in below:

```
procedure InitGraph(URL, depth)
        currentState.htmlSource ← URL.GetHtmlSource()
        currentState.clickables ← GetClickables(currentState.htmlSource)
        AddVertex(currentState)
        CreateGraph(0, depth, currentState, URL)
end procedure
procedure CreateGraph(level, depth, currentState, currentUrl)
        level = 0
        if(level<depth)
```

```
            foreach(Clickable c in currentState.clickables)
                  previousState ← currentState
                  nextState ←Execute(c)
                  transition ← CreateTransition (currentState, c, nextState)
                  if(!IncludesEdge(transition))
                        AddTransition(transition)
                  end if
                  if(!IncludesVertex(nextState))
                        AddVertex(nextState)
                        CreateGraph(level + 1, depth, nextState,
                  currentUrl);
                  end if

      transition=CreateTransition(nextState,clickableGoBack,currentState)
                  if(!IncludesEdge(transition))
                        AddTransition(transition)
                  end if
                  GoTo(previousState)
            end foreach
      end if
end procedure
```

The execution errors may occur during the Execute function when composing the graph. These errors are hold in a file named ExecutionErrors.txt. We call them 1st level errors explained in Chapter 4 in detail.

3.2.2 Load Generator

This component uses a gray box testing approach and it generates test cases for load testing. It consists of two modules. Input provider is the first module crawling input boxes of the source code. The Branch Traverser runs the test according to the branch coverage algorithm.

3.2.2.1 Input Provider

Input Provider aims to crawl the input boxes of the source code in the graph file and assign a specified value determined by the user. The input boxes such as textbox, listbox or checkbox, are extracted from the source code by using HTML Agility Pack library which is useful for parsing source codes in given algorithm below:

```
procedure GetInputs(htmlSource, URL)

       foreach(HtmlAgilityPack.HtmlNode i in htmldoc.SelectNodes("[input]"))

              input.inputClass ← i.Class

              input.inputXpath ← i.XPath

              input.inputUrl ← URL

              input.inputId ← i.Id

              input.inputType ← i.Type

              input.inputName ← i.Name

       end foreach

end procedure
```

After the input boxes are crawled, the user should specify the values of these input boxes from the input values file. The file should be filled with the URL of the input box, xpath of the input box and finally the value of the input box. In this part of the NaMoX, the user should see the html source code; therefore the white box method will be used.

According to the input values file, the input boxes values are assigned to the related element with the below algorithm:

```
procedure FillInput(Input i, File InputValues)
       value ← InputValues.GetValue(i.inputUrl, i.inputXpath)
       i.inputValue ← value
end procedure
```

3.2.2.2 Branch Traverser

Branch Traverser module reads the graph file text. It originates the URL and the clickable's xpath. Then, it handles Selenium to automate the click operation from the xpath of the clickable, also automates opening the URL action from the URL of the clickable.

- Selenium.Click(XpathOfTheClickable) : implements the click operation.
- Selenium.Open(UrlOfTheClickable) : implements opening the given URL.

In order to make the graph file keeps the transactions, branch traverser runs the test according to graph file's order, causes branch coverage analysis. A number of threads are introduced to this process, to procure load testing analysis.

Recall that NaMoX does not use browser's goBack function. It saves the current state while traversing the model in a depth first manner and restores that state while backtracking. NaMoX need to do this state saving since in dynamic pages, URL may not change although the content changes and the goBack function of the browser may result in a wrong navigation.

Unexpected errors may occur during the execution of test sequences. We call them 2nd level errors explained in Chapter 4 in details.

3.3 Usage

In Figure 3.7, the user interface of NaMoX is shown. The user should specify the initial URL and the crawl depth. The entered initial URL defines the starting point of the crawling. The entered crawl depth shows the number of clicks for each path while exploring the side. After the initial URL and crawl depth information taken from the user, Submit button triggers the Selenium Remote Control [14] tool in Mozilla Firefox, and loads the URL as entered initial URL. The Firefox browser shows automatic transitions between web pages according to the clickables. When this process ends, the model is extracted and the graph file is ready for use.

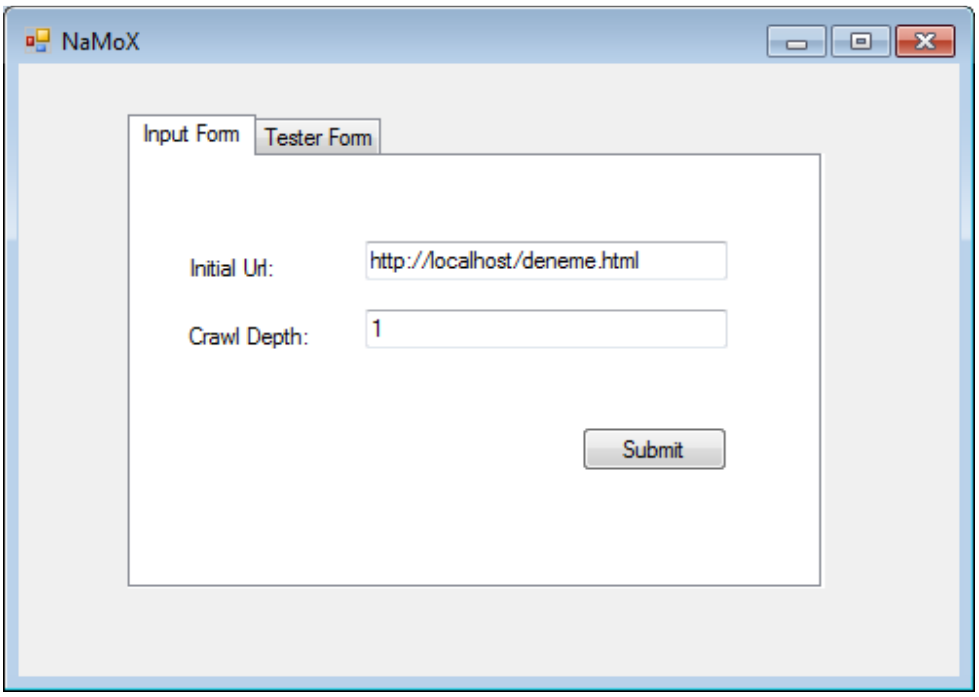

Figure 3.7: User Interface – Input Form

The server name, browser type and port that are entered by the user specify the server, browser type and port of the test environment as shown in Figure 3.8.

Before running the test, the user should prepare input values file as explained in Section 3.2.2.1. This input file includes the URL, the Xpath of the widget that takes the input, and the value of the input that the user wants to assign.

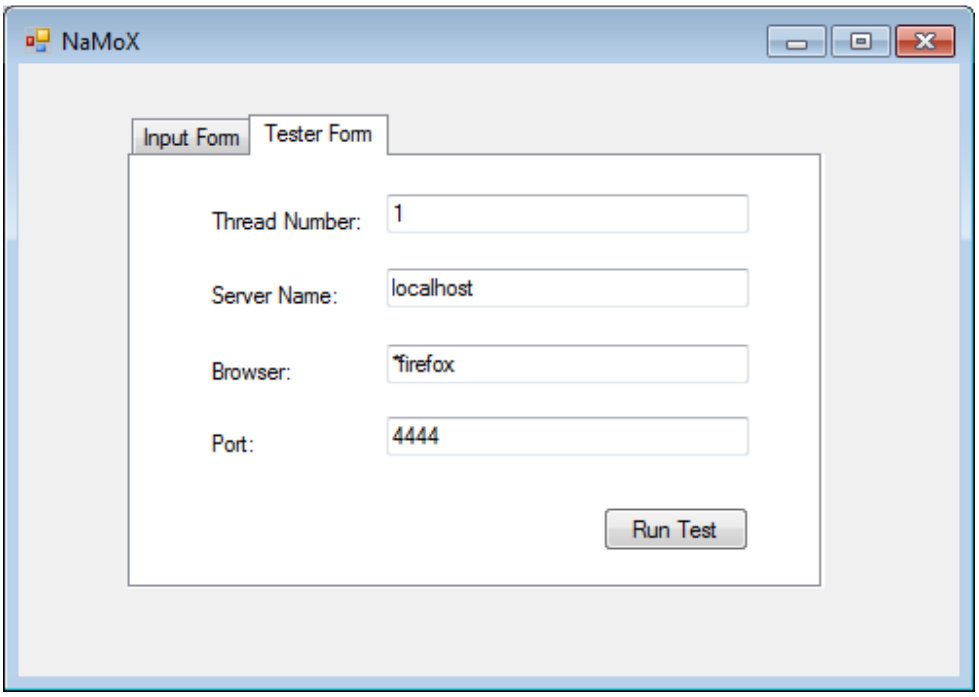

Figure 3.8: User Interface – Tester Form

CHAPTER 4

EXPERIMENTS

Our experiments verify the NaMoX under two topics:

- Accuracy
- Performance

This chapter includes three sections. First section describes the test environment. Section 4.2 focuses on the results derived in five web sites. Finally, Section 4.3 compares NaMoX with JMeter.

4.1 Test Environment

NaMoX has two progressive steps including navigational model extraction and test execution. The test execution module requires quite big memory and processing time for creating several threads concurrently. We reserved one of the METU Informatics

Institute's servers but they only allocated 3GHz processor and 2 GB RAM. Therefore, we experimented with maximum 30 threads simultaneously.

4.2 Test Results

We have used our tool NaMoX in five commercial web sites listed as:

- SALİNA - http://www.salina.com.tr
- GUCCI - http://www.gucci.com
- ERİNMEZ - http://www.erinmez.com.tr/
- MİLLİYET - http://www.milliyet.com.tr/
- ÖSYM - http://sonuc.osym.gov.tr/

SALİNA and ERİNMEZ are chosen as case studies because these web sites' all source codes can be accessed and it is easy for us to test NaMoX in these web pages.

The reason that ÖSYM is selected as a case study is the server breakdowns on this web page. ÖSYM is the web page of Turkey's educational system. In this web site, the students learn their exam results. When an important exam result is announced on the web; therefore, there are a great number of students trying to access the web server and it results with a server crash.

MİLLİYET is Turkey's very famous news web page. The site has a lot of clickables and this is an opportunity for NaMoX to make its test in such a comprehensive web domain.

GUCCI is a shopping web page that has also many clickables. The GUCCI's web site is applied as a case study in the Crawljax [27] study, therefore NaMoX selected the web site to form an example.

Table 4.1 shows the line of HTML source codes of each web site and summarizes the results of detected clickables in five web sites. In SALİNA's and ERİNMEZ's web sites, we recorded clickables in two depths; depth 2 and depth 3. This analysis shows the great expansion of number of clickables according to the depth.

Table 4.1: Detected Clickables

NAME	HTML LOC	DEPTH	NUMBER OF CLICKABLES DETECTED
SALİNA	100	2 3	143 1588
GUCCI	484	2	392
ERİNMEZ	61	2 3	86 566
MİLLİYET	243	2	3122
OSYM	113	2	40

As explained in Section 3.2.1.1, clickables are divided into two parts: The hyperlinks and the elements that have OnClick events. NaMoX reports these sections for each web application. Figure 4.2 shows the case studies' number of hyperlinks and OnClick event elements.

Table 4.2: Clickables Types

NAME	NUMBER OF HYPERLINKS	NUMBER OF ONCLICK ELEMENT EVENTS
SALİNA	129	14
GUCCI	336	56
ERİNMEZ	86	0

Table 4.2 (cont.)

MİLLİYET	2641	481
OSYM	29	11

We classified the detected errors in two parts:

First Level Errors:

While composing the graph, there can be errors in execution level. These errors are called as First Level Errors. The error list is given in the Figure 3.4. The errors which start with the digit "4" shows the client errors, the errors start with "5" define server errors. These errors are caught by using HttpException class in NaMoX.

Second Level Errors:

Second Level Errors are the caught unexpected errors while several users connect to the web server at the same time including first level errors and time out exceptions if existed.

Table 4.3 shows the statistics of detected errors. Since second level errors include first level errors, the second level error count is bigger than first level error count. The rest of the errors in second levels are the exceptions in class TimeOutException.

Table 4.3: Detected Error Numbers

NAME	**NUMBER OF FIRST LEVEL ERRORS DEPTH: 2**	**NUMBER OF SECOND LEVEL ERRORS THREAD NUMBER: 30**
SALİNA	21 (18: Hyperlink clickable error) (3: OnClick clickable error)	24 (21: Hyperlink clickable error) (3: OnClick clickable error)

Table 4.3 (cont.)

GUCCI	10 (8: Hyperlink clickable error) (2: OnClick clickable error)	16 (14: Hyperlink clickable error) (2: OnClick clickable error)
ERİNMEZ	14 (14: Hyperlink clickable error) (0: OnClick clickable error)	15 (15: Hyperlink clickable error) (0: OnClick clickable error)
MİLLİYET	32 (20: Hyperlink clickable error) (12: OnClick clickable error)	54 (40: Hyperlink clickable error) (14: OnClick clickable error)
OSYM	0	2 (2 : Hyperlink clickable error)

Table 4.4 shows the HTTP exceptions that NaMoX detected for each case study:

Table 4.4: Detected HTTP Exceptions for Each Case Study

NAME	**ERROR TYPE**	**REPETITION**
SALİNA	HTTP 403 – Forbidden	4
	HTTP 404 – Not Found	17
GUCCI	HTTP 408 – Request Timeout	6
	HTTP 404 - Not Found	4
ERİNMEZ	HTTP 408 – Request Timeout	8
	HTTP 503 – Service unavailable	2
	HTTP 404 - Not Found	4

Table 4.4 (cont.)

MİLLİYET	HTTP 408 – Request Timeout	12
	HTTP 404 - Not Found	20
OSYM	-	0

The caught errors are important for specifying the quality of a web application. NaMoX detected the failures in case studies as shown in Table 4.3, but the critical point in this experiment is measuring these failures' severities. If the initial URL of a web application is broken, the test will end and there will be no acquired useful results. Therefore, NaMoX also measures the depths of the errors. The lower the depth is, the severe is the error. Table 4.5 shows the depths of the failures in NaMoX's case studies:

Table 4.5: Error Severities

NAME	**ERROR TYPE**	**ERROR REPETITION**	**ERROR DEPTH**
SALİNA	HTTP 403 – Fobidden	4	2
	HTTP 404 - Not Found	17	2
GUCCI	HTTP 408 – Request Timeout	6	2
	HTTP 404 - Not Found	4	2
ERINMEZ	HTTP 408 – Request Timeout	8	2
	HTTP 503 – Service unavailable	2	2
	HTTP 404 - Not Found	4	2

Table 4.5 (cont.)

MİLLİYET	HTTP 408 – Request Timeout	12	2
	HTTP 404 - Not Found	14	2
	HTTP 404 - Not Found	6	1
OSYM	-	-	-

Table 4.5 explains that, in MİLLİYET's web page, HTTP 404 exceptions are occurred in first and second depth. As it is mentioned, as the depth decreases, error severity increases.

The clickable type is also important for NaMoX in defining the quality metrics. We divided clickables into two types: hyperlinks and OnClick events. According to these parameters, we weighted the errors to measure the quality. Equation 4.1 shows the relations between:

- Let *Q*: quality metric,
- C_0: number of hyperlink clickable,
- C_1: number of OnClick event clickable,
- *S*: error severity of the
- *n*: number of errors,
- W_0: weight of the hyperlink clickable,
- W_1: weight of the OnClick event clickable,
- *D*: error depth

$$\mathrm{Q} = \sum_{i=0}^{1} W_i * \left(\frac{\sum_{j=0}^{n} S_{ij} * (1/D_{ij})}{C_i}\right)$$

(Equation 4.1)

According to the Equation 4.1, as error depth increases, the quality metric (Q) of the web application decreases. If quality metric is closer to 0, it can be said that the quality of the web page increases. If we get Q as 0, it means that there is no error on the tested application.

We searched the literature to get the quality metrics for NaMoX. However, we could not find out a metric about number of errors. The existing metrics include such concerns:

- **Cohesiveness** [34]: Cohesiveness metric classifies the web pages according to their topics.
- **Number of Virtual Users** [11]: This metric measures the optimum number of virtual users according to the web application.
- **Centrality** [35]: Centrality metric defines hierarchies in the web applications.
- **Connections** [21]: This metric is about the connections that are refused when making the test.
- **Throughput** [21]: This metric is the sum of response data size divided by the number of seconds in the reporting duration.
- **Hits per Second** [21]: Hits per second metric will tell if there is a possible scalability issue with the application.
- **Pages per Second** [21]: Pages per second metric measures the number of pages requested from the application per second.
- **Web Page Search and Retrieval** [35]: This metric evaluates the performance of Web search and retrieval services.

According to the metric for NaMoX, we measured the quality metrics for each case study. Table 4.6 shows the quality metrics of the five case studies. According to the quality metric that we implemented, we assume the hyperlink clickables' weights as 0.8, OnClick event clickables' weight as 1. There are also no severity coefficients for HTTPExceptions and TimeoutExceptions in the literature; therefore we assume the following severities:

- HTTP 408 Exception: 0.6
- HTTP 414 Exception: 0.6

- HTTP 415 Exception: 0.6
- The rest HTTP 4XX Exception: 0.8
- HTTP 500 Exception: 1.2
- HTTP 503 Exception: 1.2
- The rest HTTP 5XX Exception: 1
- TimeOut Exception: 0.6

These assumptions are done according to the criticality levels of the failures. The HTTP 5XX Exceptions are occurring because of the server problems; therefore they are more important than HTTP 4XX Exceptions. If there is a server error or if the service is unavailable, the testing module will fail, so we specified highest severity coefficients for HTTP 500 and HTTP 503 exceptions. Time out exceptions including HTTP 408 and TimeOutExceptions are composing in order to thread numbers; hence we assumed a smaller coefficient. HTTP 414 and HTTP 415 exceptions' severity is also given as 0.6, because they have same severity as time out exceptions.

According to these assumptions, Equation 4.1 returns:

$$\mathrm{Q} = 0.8 * \left(\frac{\sum_{j=0}^{n} S_{0j} * (\frac{1}{D_{0j}})}{C_0} \right) + 1 * (\frac{\sum_{j=0}^{n} S_{1j} * (\frac{1}{D_{1j}})}{C_1})$$

(Equation 4.2)

Table 4.6 Quality Measurement

NAME	**QUALITY METRIC**
SALİNA	0,135
GUCCI	0,025
ERİNMEZ	0,056

Table 4.6 (cont.)

MİLLİYET	0,015
OSYM	0,016

As it is mentioned, as quality metric close to 0, the quality of the web page increases. According to Table 4.6, we can order the qualities of the case studies as:

1. MİLLİYET
2. ÖSYM
3. GUCCI
4. ERİNMEZ
5. SALİNA

NaMoX also saves the response times while test execution for each thread. For SALİNA's web site, Table 4.7 represents server response times for three threads for five clickable.

Table 4.7 Server Response Times

ACTION	**THREAD 1 (msec)**	**THREAD 2 (msec)**	**THREAD 3 (msec)**
open:default.aspx?ln=tr	190	313	342
goBack:http://www.salina.com.tr	186	355	377
open:default.aspx?ln=eng	385	552	627
goBack:http://www.salina.com.tr/	166	262	312
open:bilgi.aspx?ln=tr&id=11	241	313	424

Table 4.8 shows the durations of the model extraction and test execution models. Normally, it is expected the durations close to each other. However, as Daniel A. Menascé [11] mentioned; the response times are increases while the number of virtual users increasing.

Table 4.8: Durations of NaMoX

NAME	**MODEL EXTRACTION DEPTH: 2**	**TEST EXECUTION THREAD NUMBER: 30**
SALİNA	6 min.	8 min.
GUCCI	3 hours 24 min.	4 hours 52 min.
ERİNMEZ	46 min.	1 hour 3 min.
MİLLİYET	19 hours 38 min.	22 hours 29 min.
OSYM	4 min.	5 min.

4.1Comparison with JMeter

JMeter [5] allows load testing to measure performance of a web application. However, JMeter requires manual effort. The user should extract the links and input boxes, and create HTTP Requests for each hyperlink and input boxes. According to our results in NaMoX, the number of clickables range from 40 to 2549 in our experiments. Detecting these clickables and exercising each of them cannot be done manually in JMeter, or requires too much time to implement. However, since JMeter do not use real browsers, the response times may be less than NaMoX.

Table 4.9 shows the durations of threads' executions and spent manual effort in JMeter. In that example, we created a five click scenario with three threads. Firstly, the links are specified manually, and then these links are composed as a HTTP Request sample in JMeter. After that, a listener is added manually to see the logs.

Table 4.9: Results on JMeter

ACTION	**MANUAL EFFORT (min)**	**THREAD 1 (msec)**	**THREAD 2 (msec)**	**THREAD 3 (msec)**
Detecting the Links and Input Values	15	-	-	-
Adding HTTP Request Defaults	5	-	-	-
Adding HTTP Request of open:default.aspx?ln=tr	2	256	288	311
Adding HTTP Request of goBack:http://www.salina.com.tr	2	315	351	362
Adding HTTP Request of open:default.aspx?ln=eng	2	372	485	504
Adding HTTP Request of goBack:http://www.salina.com.tr/	2	298	301	319
Adding HTTP Request of open:bilgi.aspx?ln=tr&id=11	2	322	324	387
Adding a Listener	2	-	-	-

Table 4.9 shows the durations for only a simple scenario in JMeter. It takes time to add HTTP Request more than a minute. But, the most time consuming step in JMeter is designing a test scenario. As number of clickables is increased, detecting the links and input values durations will also be increased.

CHAPTER 5

CONCLUSION

This study is an automatic navigational model extractor for load testing on dynamic content web sites to improve testability, accessibility and accuracy. The main contributions of the paper are:

- A systematic process and algorithm to infer a state machine from a web domain. The states of the state machine are the collection of HTML source code and the clickables. The transitions are the onclick events on clickables or hyperlinks together with associated document item, which can also involve Go Back events. Challenges addressed include the identification of clickable elements, and the construction of the state machine;
- A systematic process to create test suites for load testing using branch coverage on the state machine model instead of using record and replay algorithms.
- Five case studies used to measure the effectiveness, accuracy, and performance of the proposed approach.

Future work consists of conducting more case studies to improve the ability of finding different clickables such as onMouseOver events, onSelectChange events. Testing will be done with different coverage techniques such as switch coverage, random tests to catch errors in unexpected user behavior. Furthermore, the results will extend to more graphical user interface and generating more detailed reports by combining this study with a load test tool. NaMoX will also add an error bar in its results which will measure the same web application in different days and reports the severity level for each measurement. In addition, the system considerations will be included in NaMoX and NaMoX will decide the thread number of load generation in a strong performance server.

REFERENCES

[1] Buret Julien, Droze Nicolas. An Overview of Load Test Tools (2003).

http://clif.objectweb.org/load_tools_overview.pdf , last visited: January 2012.

[2] Gerardo Canfora, Massimiliano Di Penta. Testing Services and Service-Centric Systems: Challenges and Opportunities. IT Professional, pages 10-17, 2006

[3] http://grinder.sourceforge.net, last visited: December 2011

[4] S.S. Foley, V. Pandey, M. Tang, F. Terkhorn, A. Venkatraman, Benchmarking Servers using Virtual Machines (2007). http://www.cs.indiana.edu/~mhtang/paper.pdf, last visited: January 2012.

[5] J. Grundy, J. Hosking, L. Li, and N. Liu. Performance Engineering of Service Compositions. *In Proceedings of the* 2006 International Workshop on Service–Oriented Software Engineering (SOSE 2006), pages 26–32, 2006

[6] http://www.xceptance-loadtest.com/products/xlt/what-is-xlt.html, Last visited December 2011

[7] http://sourceforge.net/projects/dieseltest, last visited: December 2011

[8] Anantha K. Bangalore, Arun K. Sood. Securing Web Servers Using Self Cleansing Intrusion Tolerance (SCIT). *In Proceedings of the* Second International Conferenceon Dependability , pages 60-65, 2009.

[9] Grig Gheorghi. A Look at Selenium. Software Quality Engineering, 7(8):38–44. 2005.

[10] Stefan Frei, Thomas Duebendorfer, Bernhard Plattner. Firefox (in) security update dynamics exposed. ACM SIGCOMM Computer Communication Review 39 (2009), no. 1, pages 16–22, 2009.

[11] Daniel A. Menascé. Load Testing of Web Sites. Internet Computing, IEEE, pages 70-74, 2002.

[12] G. Banga and P. Druschel. Measuring the Capacity of aWeb Server under Realistic Loads. *In Proceedings of the* World Wide Web, 2(1-2), pages 69–83, 1999.

[13] https://browsermob.com, last visited: December 2011

[14] A. Holmes and M. Kellogg. Automating Functional Tests Using Selenium. *In Proceedings of the* AGILE Conference '06, pages 270 - 275, 2000.

[15] A. Marchetto, F. Ricca, and P. Tonella. A case study-based comparison of web testing techniques applied to Ajax web applications. Internatıonal Journal on Software Tools for Technology Transfer, 10(6):477–492, 2008.

[16] David Mosberger, Tai Jin. Httperf—A Tool for Measuring Web Server Performance. In *Proceedings of the* Workshop on Internet Server Performance, 1998.

[17] Mercury Interactive Corporation. Load Testing to Predict Web Performance. Technical Report WP-1079-0604, Mercury Interactive Corporation, 2004.

[18] D. Darheim, J. Grundy, J. Hosking, C. Lutteroth, G. Weber. Realistic Load Testing of Web Applications. *In Proceedings of the* 10th European Conference on Software Maintenance and Reengineering, pages 11-70, 2006.

[19] Scott Barber. How fast does a website need to Be? (2010). http://www.perftestplus.com/resources/how_fast.pdf, last visited: January 2012

[20] Torsten Grust. Accelerating XPath Location Steps. *In Proceedings of the* 2002 ACM SIGMOD international conference on Management of data, pages 109-120, 2002.

[21] http://softwareqatestings.com, last visited: January 2012.

[22] Mark Utting, Alexander Pretschner, Bruno Legeard. A Taxonomy of Model-Based Testing. Technical report, Department of Computer Science, The University of Waikato (New Zealand), 2006.

[23] A. Pretschner, W. Prenninger, S. Wagner, C. Kuhnel, M. Baumgartner, B. Sostawa, R. Zolch, T. Stauner. One Evaluation of Model-Based Testing and its Automation. *In Proceedings of the* 27th international conference on Software engineering, 2005.

[24] D. Kung, C. H. Liu, and P. Hsia. A model-based approach for testing Web applications. *In Proceedings of the* Twelfth International Conference on Software Engineering and Knowledge Engineering, 2000.

[25] J. Z. Gao, D. Kung, P. Hsia, Y. Toyoshima, and C. Chen. Object state testing for object-oriented programs. *In Proceedings of the* 19th Computer Software and Applications Conference (COMPSAC 95), pages 232-238, 1995.

[26] C. D. Turner and D. J. Robson. The state-based testing of object-oriented programs. *In Proceedings of the* 1993 IEEE Conference on Software Maintenance (CSM-93), pages 302-310, 1993.

[27] Ali Mesbah, Engin Bozdag, Arie van Deursen. Crawling AJAX by Inferring User Interface State Changes. *In Proceedings of* the Eighth International Conference Web Engineering, 2008. ICWE '08., pages 122-134, 2008

[28] Arie van Deursen and Ali Mesbah. Research Issues in the Automated Testing of Ajax Applications. *In Proceedings of the* SOFSEM 2010 Theory and Practice of Computer Science, 2010.

[29] http://htmlagilitypack.codeplex.com, last visited: December 2011.

[30] Charles Reis, Adam Barth, and Collin Jackson. Browser security: lessons from Google chrome. Communications of ACM, pages 45–49, 2009.

[31] http://windows.microsoft.com/tr-TR/internet-explorer/products/ie/home, last visited: December 2011.

[32] J. Clark and S. DeRose. XML path language (XPath). W3C Recommendation, 1999.

[33] http://www.modemhelp.net/httperrors/httperrors.shtml, last visited: December 2011.

[34] Xiaolan Zhu, Susan Gauch. Incorporating quality metrics in centralized/distributed information retrieval on the World Wide Web. *In the Proceedings of the* 23rd annual international ACM SIGIR conference on Research and development in information retrieval, pages 288-295, 2000.

[35] Devanshu Dhyani, Wee Keong Ng, Sourav S. Bhowmick. A Survey of Web Metrics. ACM Computing Surveys, Vol. 34, No.4, pages 469-503, 2002.

TEZ FOTOKOPİSİ İZİN FORMU

ENSTİTÜ

Fen Bilimleri Enstitüsü ☐

Sosyal Bilimler Enstitüsü ☐

Uygulamalı Matematik Enstitüsü ☐

Enformatik Enstitüsü ☒

Deniz Bilimleri Enstitüsü ☐

YAZARIN

Soyadı: KARA
Adı: İsmihan Refika
Bölümü: Bilişim Sistemleri

TEZİN ADI (İngilizce): Automatic Navigation Model Extraction for Web Load Testing

TEZİN TÜRÜ: Yüksek Lisans ☒ Doktora ☐

1. Tezimin tamamından kaynak gösterilmek şartıyla fotokopi alınabilir. ☒

2. Tezimin içindekiler sayfası, özet, indeks sayfalarından ve/veya bir bölümünden kaynak gösterilmek şartıyla fotokopi alınabilir. ☐

3. Tezimden bir (1) yıl süreyle fotokopi alınamaz**.** ☐

TEZİN KÜTÜPHANEYE TESLİM TARİHİ:.....