FINDING MALFORMED HTML OUTPUTS AND UNHANDLED EXECUTION ERRORS OF ASP.NET APPLICATIONS

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

BY

MEHMET ERDAL ÖZKINACI

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

MAY 2011

Approval of the Graduate School of Informatics

Prof. Dr. Nazife Baykal
Director

I certify that this thesis satisfies all the requirements as a thesis for the degree of Master of Science.

Prof. Dr. Yasemin Yardımcı Çetin
Head of Department

This is to certify that we have read this thesis and that in our opinion it is fully adequate, in scope and quality, as a thesis for the degree of Master of Science.

Assist. Prof. Dr. Aysu Betin Can
Supervisor

Examining Committee Members

Assoc. Prof. Dr. Onur Demirörs (METU, II)

Assist. Prof. Dr. Aysu Betin Can (METU, II)

Aydın Nusret Güçlü, MSc. (METUTECH, Stratek)

Assoc. Prof. Dr. Altan Koçyiğit (METU, II)

Assoc. Prof. Dr. Halit OğuzTüzün (METU, CENG)

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: Mehmet Erdal Özkınacı

Signature :

ABSTRACT

FINDING MALFORMED HTML OUTPUTS AND UNHANDLED EXECUTION ERRORS OF ASP.NET APPLICATIONS

Özkınacı, Mehmet Erdal
M.Sc., Department of Information Systems
Supervisor : Assist. Prof. Dr. Aysu Betin Can

May 2011, 44 pages

As dynamic web applications are becoming widespread nearly in every area, ASP.NET is one of the popular development languages in this domain. The errors in these web applications can reduce the credibility of the site and cause possible loss of a number of clients. Therefore, testing these applications becomes significant. We present an automated tool to test ASP.NET web applications against execution errors and HTML errors that cause displaying inaccurate and incomplete information. Our tool, called Mamoste, adapts concolic testing technique which interleaves concrete and symbolic executions to generate test inputs dynamically. Mamoste also considers page events as inputs which cannot be handled with concolic testing. We have performed experiments on a subset of an heavily used ASP.NET application of a government office. We have found 366 HTML errors and a faulty component which is used almost every ASP.NET page in this application. In addition, Mamoste discovered that a common user control is misused in several generated pages.

Keywords: automated testing, concolic testing, dynamic web pages, asp.net

ÖZ

ASP.NET UYGULAMALARINDAKİ HATALI HTML ÇIKTILARININ VE
ÖNGÖRÜLEMEMİŞ ÇALIŞMA HATALARININ BULUNMASI

Özkınacı, Mehmet Erdal
Yüksek Lisans, Bilişim Sistemleri Bölümü
Tez Yöneticisi : Yard. Doç. Dr. Aysu Betin Can

Mayıs 2011, 44 sayfa

ASP.NET, dinamik web uygulamalarının geliştirilmesinde kullanılan popüler dillerden biridir. Web uygulamalarındaki hatalar, bu uygulamaların güvenilirliğini ve kullanıcı sayısını azaltabilir. Bundan dolayı, bu uygulamaların test edilmesi önem kazanmaktadır. Bu çalışmada, öngörülememiş çalışma hataları ve tarayıcılarda bozuk görüntülerin oluşmasına neden olan HTML hatalarını içeren ASP.NET uygulamalarını, otomatik olarak test eden bir araç sunacağız. Aracımızın ismi Mamoste'dir. Mamoste, test girdilerini dinamik olarak üretmek için somut ve sembolik çalışma yöntemlerini dönüşümlü olarak kullanan concolic testi ASP.NET uygulamalarına adapte eder. Mamoste, concolic test ile çözülemeyen web sayfalarındaki olayları da girdi olarak ele alır. Mamoste ile, bir kamu kurumu tarafından yoğun olarak kullanılan ASP.NET uygulamasının testini gerçekleştirdik. Mamoste, 366 HTML hatası ve bu uygulamanın hemen hemen her sayfasında kullanılan hatalı bir bileşen tespit etti. Bunun yanı sıra; Mamoste, üretilen sayfaların bir kısmında hatalı kullanılmış genel bir kontrol açığa çıkarttı.

Anahtar Kelimeler: otomatik test, concolic test, dinamik web sayfaları, asp.net

dedicated to my nephew, my uncle Abid, my family, and walnut tree

ACKNOWLEDGMENTS

I am deeply grateful to my supervisor, Assist. Prof. Dr. Aysu Betin Can, for her encouragement and guidance. I could not finish this study without her support.

I would like to thank all of those who supported me in any respect during this study especially my friends and the company which I work for, Stratek Strategic Technologies R-D.

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

Most importantly, I would like to thank whole my family, Şükriye ÖZKINACI, Feramuz ÖZKINACI, Nuriye ÖZKINACI, İbrahim Halil ÖZKINACI, and Emel ASLANTAŞ, for their endless love, so glad I have them.

TABLE OF CONTENTS

ABSTRACT iv

ÖZ v

DEDICATON vi

ACKNOWLEDGMENTS vii

TABLE OF CONTENTS viii

LIST OF TABLES x

LIST OF FIGURES xi

LIST OF SYMBOLS xii

CHAPTERS

1 INTRODUCTION 1

1.1 Problem Statement 3

1.2 Overview 4

2 BACKGROUND AND LITERATURE SURVEY 6

2.1 Dynamic Web Languages and ASP.NET 6

2.2 Dynamic Test Input Generation 9

2.3 Related Work 13

3 MAMOSTE 16

3.1 Methodology 16

3.2 Design And Implementation 21

3.2.1 System Architecture 21

3.2.2 Implementation 25

3.2.3 Instrumentation 28

3.3 Usage 30

3.3.1 Installation 30
3.3.2 Example 35
4 EXPERIMENTS 36
4.1 Applying Mamoste to SGB.net v2 37
4.2 Testing SGB.net v2 Manually 38
4.3 Applying Mamoste to SGB.net v1 38
5 CONCLUSION 41
REFERENCES 42

LIST OF TABLES

Table 2.1 Execution Results of Microsoft Pex 15

Table 3.1 Database Table of the Districts 21

Table 3.2 Code and Instrumented Code Samples 29

Table 3.3 Test Results of Given Example 33

Table 4.1 Comparison of Manual Testing and Mamoste 37

Table 4.2 HTML Faults found by Mamoste 39

Table 4.3 HTML Faults of Menu Component found by Mamoste 40

LIST OF FIGURES

Figure 2.1 HTML Interface File of DivideAndMultiply ASP.NET Web Form 8

Figure 2.2 Application Logic File of DivideAndMultiply ASP.NET Web Form 10

Figure 2.3 Code for illustrating Symbolic Execution [1] 11

Figure 2.4 Code for illustrating Defective of Microsoft Pex 15

Figure 3.1 Algorithm of Concolic Testing 17

Figure 3.2 Algorithm of Finding New Inputs in Concolic Testing 18

Figure 3.3 Code for illustrating Need of Manual Instrumentation on Web Applications Using Database 20

Figure 3.4 System Architecture of Mamoste 22

Figure 3.5 Class Diagram of Concolic Package of Mamoste 26

Figure 3.6 Class Diagram of Trace Listener Package of Mamoste 27

Figure 3.7 Class Diagram of GUI Package of Mamoste 28

Figure 3.8 Configuration File of Mamoste 31

Figure 3.9 Change in Configuration File of Tested Web Application for Trace Listener of Mamoste 32

Figure 3.10 Change in Global Application Class of Tested Web Application for Trace Listener of Mamoste 33

Figure 3.11 Instrumented Application Logic File of DivideAndMultiply ASP.NET Web Form 34

LIST OF SYMBOLS

ASMX	Active Server Methods Extended
ASP	Active Server Pages
ASPX	Active Server Pages Extended
CGI	Common Gateway Interface
DLL	Dynamic Link Library
GUI	Graphical User Interface
HTML	HyperText Markup Language
HTTP	HyperText Transfer Protocol
IIS	Internet Information Services
ISAPI	Internet Server Application Programming Interface
JSP	Java Server Pages
PHP	Hypertext Preprocessor
URL	Uniform Resource Locator
W3C	World Wide Web Consortium
XML	eXtensible Markup Language

CHAPTER 1

INTRODUCTION

Web applications with dynamic content are becoming very popular almost in every business area such as banking, entertainment and government agencies. One of the reasons of this popularity is that updating and maintenance of these applications do not require distribution and installation software. Besides, they are accessible by any computer with Internet access, which means there are potentially thousands of clients. In addition, in terms of user interaction these applications are starting to compete with the desktop applications due to dynamic web application technologies.

Information technologies (solutions, applications, programs) shift to Internet environment. Static web technologies cannot support this trend and there is an appearing demand. In order to meet upcoming needs, dynamic web languages are developed in the last decades such as the scripting languages ASP and PHP. After these technologies several complex frameworks are developed such as ASP.NET and JSP for dynamic web application projects. These frameworks are more preferable than ASP and PHP scripting languages because they have a number of features which ASP and PHP cannot cope with. They also support object oriented architecture that is the most popular view of late years while developing software projects.

ASP.NET is an example of dynamic web application development language. ASP.NET pages run typically on a server and generate HTML or XML pages that are sent to client browsers. A common practice in ASP.NET is separating application logic, such as event handlers, from the static HTML parts such as widgets to be displayed on the browser. In other words, the static HTML or XML parts reside in a separate file from the code for handling events and generating dynamic parts. Although this separation enables reuse of the code, it makes the development process error-prone.

Dynamic web applications even if developed with scripting languages have proliferated in almost all areas such as banking and communication. As a result of this tendency, accuracy and trusty of these kinds of applications have become increasingly important. Thus, there is a risen need to test and verify these applications. Researchers have been interested in improving testing and verification techniques, algorithms and tools for dynamic web applications. In other words, testing and verification society has broaden their direction to dynamic web technologies and applications. There are several techniques such as dynamic test generation, fuzz testing, symbolic execution and concolic execution (combination of symbolic and concrete execution). Researchers have worked on these techniques and have diversified them and their algorithms to test and verify dynamic web applications. For example, Halfond et al. [2] extract interfaces for Java based web applications to improve the effectiveness of test input generation. Moreover, Artzi et al. [3] target PHP applications and aim to catch faulty HTML outputs of such dynamic web sites.

Several works on testing and verification of dynamic web applications have focused on PHP and JSP. For example, Wassermann et al. [4] apply symbolic execution to discover SQL injections of PHP web applications. Halfond et al. [5] are interested in parameter mismatches in JSP web applications by using a static analysis based approach. Moreover, Artzi et al. [3] use concolic execution technique to find inaccurate HTML outputs and run-time errors of PHP. On the other hand, Fu et al. [6] are interested in SQL injection on ASP.NET web applications with the help of symbolic execution, but later they switch their focus on Java. As a result of all these researches, we have decided to work with ASP.NET dynamic web applications because we believe that ASP.NET dynamic web language is still virgin with respect to testing and verification.

In this thesis we present an automated tool, called Mamoste, to check web applications developed with ASP.NET whether there are execution errors crashing the applications and whether they produce malformed HTML outputs causing to display inaccurate and incomplete information.

Mamoste is based on concolic testing pioneered by DART [7] and CUTE [8]. In concolic testing an application is executed on concrete input values to variables and then using symbolic execution new concrete inputs are generated to maximize code coverage. By solving symbolic constraints that are derived from executed control flow, concolic testing generates new

concrete inputs to exercise unexplored paths. Mamoste applies this technique on ASP.NET applications. In addition, Mamoste triggers all the user implemented event handlers and supplies inputs. In a sense, we perform unit testing with dynamic input generation where a unit is an event handler.

We used our tool to detect HTML errors on a subset of an heavily used ASP.NET application owned by the Ministry of Finance of Turkey. Among numerous HTML errors and warnings discovered, Mamoste revealed errors on two important highly reused components of the application. We also tested older version of the same subset and compared the bugs found by the development team to bugs found by Mamoste. We found that manual testing has been ineffective compared to Mamoste. Firstly, Mamoste used less test inputs and collected more HTML outputs. Secondly, Mamoste increased code coverage by executing different branches of tested program.

1.1 Problem Statement

Since many Internet users interact with the dynamic web sites, the faults that crash the application or interrupt a transaction or cause to display inaccurate and incomplete information are not tolerable. These kinds of errors reduce the credibility of the site which can cause possible loss of a number of clients. Since dynamic web applications produce HTML pages at runtime depending on the interaction with the user, developers are more likely to make errors compared to developing sites with static web technologies. In the case of ASP.NET applications, all execution paths may not return correct HTML pages created dynamically. For example, code segment closing end tag of an HTML table may not be in execution path because of a condition when code segment opening begin tag of that HTML table is executed. In other words, an execution path of a program may produce malformed, wrong and incomplete HTML because conditions and input values of the program determine the paths executed. However, while using static web technologies, all HTML pages are developed manually in a straight-forward approach.

There are several researches analyzing ASP.NET web applications. SAFELI [6] detects SQL injection vulnerabilities at compile time by inspecting MSIL bytecode of ASP.NET dynamic web applications. Although it is an important problem, it is orthogonal to detecting mal-

formed HTML outputs. Microsoft Pex [9] and Moles [10] aim to generate unit tests for .NET framework which includes ASP.NET technology. However, Pex requires the developers to change the modifier of event handlers to public since Pex creates unit tests for only public methods. Also, unit tests use assertions which can detect executions errors; however, it is not clear how Pex can detect malformed HTML outputs.

There are specific topics worked on such as SQL injection, execution errors and security gaps related to dynamic web applications and these subjects are not less important than each other, but they have been enormously worked subjects. As a result, we are interested outputs of dynamic web applications. The outputs of these applications can be HTML and XML files and these output files can be malformed, wrong and non-standard, so browsers cannot display these kinds of output files. Malformed outputs of dynamic web applications may be tolerable up to some level by browsers but there is still a limit of tolerance that browsers can do. For this reason, this subject is challenging for us and we decide to deal with outputs of ASP.NET dynamic web applications.

Researchers have been rarely interested in ASP.NET dynamic web applications as aforementioned above. Therefore, this gap leads us to focus on ASP.NET dynamic web applications. Moreover, HTML outputs of ASP.NET applications have not studied yet. Thus, we have decided to automate detecting malformed HTML outputs and run-time execution errors of ASP.NET dynamic web applications.

To sum up, the problems we aim to solve in this study are;

- To discover malformed HTML outputs of ASP.NET dynamic web applications
- To discover unhandled execution errors of ASP.NET dynamic web applications

1.2 Overview

In this thesis study, we studied on concolic execution testing technique which was found by combining concrete and symbolic executions and we developed an automated tool named Mamoste based on concolic execution that tries to reveal run-time execution errors and defective HTML outputs generated by ASP.NET dynamic web applications.

There are five chapters in this thesis. First chapter introduces dynamic web application concept, concolic execution, related researches and our tool briefly. Then, problem statement section gives reasons why we choose this subject and indicates our motivation.

Second chapter presents background knowledge and literature survey about dynamic web languages and applications. After that dynamic test input generation techniques are addressed especially concolic execution, and finally works related to ours are explained.

In chapter three, we propose our solution in detail. We explain methodology of the study and system architecture of Mamoste. We give technical information about Mamoste and our concolic execution perspective. Lastly, explanation of how to apply Mamoste on ASP.NET dynamic web applications is given with a running example in depth.

Fourth chapter presents experimental results of the study on a subset of an ASP.NET dynamic web application, called SGB.net, that is used intensely by the Ministry of Finance of Turkey as well as by other governmental organizations. This chapter includes comparison of manual testing and Mamoste on this application. Moreover, current version of the SGB.net is compared with previous version. In this chapter, we also emphasize and make inferences on the results of these experiments.

The final chapter evaluates the study and summarizes outputs of the study. Limitations and future works are also explained in this chapter.

CHAPTER 2

BACKGROUND AND LITERATURE SURVEY

2.1 Dynamic Web Languages and ASP.NET

Web pages that are implemented by using dynamic web technologies have dynamic content in contrast to static web pages. Contents of dynamic web pages can change in time, according to user interaction, request parameters that are given to these pages and etc. However, static web pages always display same content, it does not change until these pages are changed.

The underlying working principle is actually simply the same for all of dynamic web technologies. This principle can be summarized as below.

- The client (mostly a browser like Internet Explorer, Google Chrome, Mozilla Firefox) prepares request data and sends it to the web server.
- The web server receives the request data and detects what file extension in the request data is.
- After comprehending the file extension, the web server redirects the request data to the related handler.
- The handler that the request data is sent prepares response which is asked by the client.
- The handler gives prepared response to the web server and the web server sends it to the client.

Relations between handlers such as ISAPI, CGI and file extensions like aspx, asmx are coordinated in the web servers during installation of programs or by developers. Thus, the web

servers know which handlers deal with which file extensions. There is no need to process static files since their contents are static. For this reason, the web servers send these files as they are to the clients.

There are many dynamic web languages such as PHP, ASP, Perl, ASP.NET, JSP to create dynamic web applications. There are differences between these languages. For example, ASP [11] and PHP [12] are scripting languages. However, ASP.NET [13] and JSP [14] are parts of huge frameworks. Moreover, ASP and PHP scripts are inlined into HTML tags, i.e. scripts and HTML tags are located in the same file. On the other hand, frameworks of ASP.NET and JSP can manage GUI parts and code parts as different files. In other words, the static HTML or XML parts reside in a separate file from the code for handling events and generating dynamic parts [15]. Also, ASP scripts can be written inside HTML tags in ASP.NET framework. ASP and PHP are interpreted languages but ASP.NET and JSP are compiled ones. In other words, source codes of ASP and PHP applications need to be copied to the web server that serves these applications. In contrast, there is no need to copy source code to the web server for ASP.NET and JSP. It is enough to copy compiled files to the web server in ASP.NET and JSP.

ASP is earlier server-side scripting language of Microsoft. Later, Microsoft has developed the ASP.NET dynamic web language, but ASP.NET is not newer version of ASP. It is an entirely new technology for server-side scripting. ASP.NET is a powerful tool for developing dynamic and interactive web pages in .NET framework of Microsoft. The web server that executes ASP.NET applications is IIS [16]. ASP.NET has many new features such as language support, new controls, XML-based components which are not supported by ASP. Since ASP.NET engine can run compiled code, the performance in responding to client request in the web server is increased. ASP.NET dynamic web applications use `CodeBehind` mode or `CodeFile` mode. The web applications that use `CodeBehind` mode need compiled files, called dll, in the web server but there is no need for source code in this mode. On the other side, source code has to be copied into the web server while using `CodeFile` mode.

ASP.NET pages are called web forms and they consist of HTML interface and application logic files [15]. The HTML interface file of a web form can be formed with standard HTML, web controls of ASP.NET framework, XML, and scripts. Scripts are executed on the web server by ASP.NET engine. Also, the web controls are executed and converted into

```
1  <%@ Page Language="C#" AutoEventWireup="true" Inherits="Example.DivideAndMultiply"
    CodeBehind="DivideAndMultiply.aspx.cs" %>
2  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
3  <html xmlns="http://www.w3.org/1999/xhtml">
4  <head runat="server"><title></title></head>
5  <body>
6      <form id="form1" runat="server"><div>
7          <div><asp:Label ID="lblResult" Font-Bold="true" Font-Size="Large"
            ForeColor="Red" runat="server"></asp:Label></div><br />
8          <div><asp:Label ID="lblNum1" Text="Number 1" runat="server"></asp:Label>
9          <asp:TextBox ID="txtNum1" runat="server"></asp:TextBox></div>
10         <div><asp:Label ID="lblNum2" Text="Number 2" runat="server"></asp:Label>
11         <asp:TextBox ID="txtNum2" runat="server"></asp:TextBox></div><br />
12         <div><asp:Button ID="btnDivide" OnClick="btnDivide_Click" Text="Divide"
            runat="server"></asp:Button>
13         <asp:Button ID="btnMultiply" OnClick="btnMultiply_Click" Text="Multiply"
            runat="server"></asp:Button></div>
14     </div></form></body>
15 </html>
```

Figure 2.1: HTML Interface File of DivideAndMultiply ASP.NET Web Form

HTML controls by ASP.NET engine before sending response to the client. The application logic file of a web form composes of event handlers and methods. Figure 2.1 and Figure 2.2 illustrate an example named `DivideAndMultiply.aspx` for ASP.NET web forms. The `DivideAndMultiply.aspx` web form is designed to divide or multiply the given two numbers by the users.

Figure 2.1 shows the HTML interface of `DivideAndMultiply.aspx` and it includes standard HTML tags and web controls of ASP.NET framework. In this file there are three labels, two text boxes and two buttons. It can be added events to any of these web controls and bound them with custom event handlers. For instance, in this example the `btnDivide_Click` event handler is bound to the `OnClick` attribute of the `btnDivide` button and the `btnMultiply_Click` event handler is bound to the `OnClick` attribute of the `btnMultiply` button. Note that the implementations of these event handlers are in the application logic file in Figure 2.2.

As the first line of Figure 2.1 says, the code behind the HTML interface is implemented in C# and it resides in the application logic file `DivideAndMultiply.aspx.cs` which is shown in Figure 2.2. The application logic file implements the event handlers that are bound to the web controls of the HTML interface file in Figure 2.1.

In this example there are three event handlers, `Page_Load`, `btnDivide_Click` and `btnMultiply`

`_Click`. The `Page_Load` event handler is the default event handler of every ASP.NET web form and this event handler initializes the text boxes and the label (lines 6-8) in Figure 2.2. The second event handler, `btnDivide_Click`, is a custom event handler for the `OnClick` event of the `btnDivide` button. This binding can be defined in the interface file as well as in the application logic file. In this example the event handler is bound to the button in the interface file. This event handler performs a division operation and puts the result into the `Text` property of the label `lblResult` in line 20. If one of the given numbers is negative, it puts an error message in boldface into the label in line 23. The third event handler, `btnMultiply_Click`, is a custom event handler for the `OnClick` event of the `btnMultiply` button. This event handler performs a multiplication operation and puts the result into the `Text` property of the label `lblResult` in line 30. If one of the given numbers is negative, it puts an error message in boldface into the label in line 33.

2.2 Dynamic Test Input Generation

Concolic testing is a hybrid software verification technique that interleaves concrete execution with symbolic execution to generate test inputs dynamically. The basic idea behind the concolic testing is to execute the application under test on some initial inputs. After that, additional inputs are obtained by solving constraints derived from previously executed paths. Execution continues with generated new inputs until no new inputs are obtained.

There is an increase in the number of tools that implement hybrid algorithm of concrete execution and symbolic execution (concolic testing) to generate test inputs dynamically because of success of concolic testing on code coverage. Some of these tools are DART [7], CUTE [8], EXE [17] and Pex [9]. The DART [7] works by code instrumentation on C programs to achieve symbolic evaluation and to generate new input values for possible program execution paths. The CUTE [8] is a follower of the DART. The CUTE performs concolic execution on automating unit testing with memory graphs. The Pex generates unit test for programs written with C#, Visual Basic and F# languages by using concolic execution.

Concrete execution is simply running a program with concrete values. On the other side, symbolic execution uses symbolic values for the program variables instead of concrete [18]. Symbolic execution is a technique which executes programs with symbolic input values, in-

```
1  using System;
2  namespace Example {
3      public partial class DivideAndMultiply : System.Web.UI.Page {
4          protected void Page_Load(object sender, EventArgs e) {
5              if (!IsPostBack) {
6                  txtNum1.Text = "Enter a number...";
7                  txtNum2.Text = "Enter a number...";
8                  lblResult.Text = "Result will be displayed here...";
9              }
10         }
11         protected void btnDivide_Click(object sender, EventArgs e) {
12             int num1 = 0;
13             int num2 = 0;
14             try {
15                 num1 = Convert.ToInt32(txtNum1.Text.Trim());
16                 num2 = Convert.ToInt32(txtNum2.Text.Trim());
17             }
18             catch (Exception) {}
19             if (num1 >= 0 && num2 >= 0) {
20                 lblResult.Text = "Division : " + (num1 / num2).ToString();
21             }
22             else {
23                 lblResult.Text = "<br><b>Please enter natural numbers...";
24             }
25         }
26         protected void btnMultiply_Click(object sender, EventArgs e) {
27             int num1 = Convert.ToInt32(txtNum1.Text.Trim());
28             int num2 = Convert.ToInt32(txtNum2.Text.Trim());
29             if (num1 >= 0 && num2 >= 0) {
30                 lblResult.Text = "Multiplication : " + (num1 * num2).ToString();
31             }
32             else {
33                 lblResult.Text = "<br><b>Please enter natural numbers...";
34             }
35         }
36     }
37 }
```

Figure 2.2: Application Logic File of DivideAndMultiply ASP.NET Web Form

```
1   public static int testMethod(int x, int y) {
2       if (x > 3) {
3           int z = x + 2;
4           if (z < y)
5               return z;
6           else
7               return y;
8       }
9       return 0;
10  }
```

Figure 2.3: Code for illustrating Symbolic Execution [1]

stead of concrete data to infer behavior of the programs. Also, symbolic execution expresses the values of program variables symbolically. Consequently, the outputs computed by a program are stated as a function of the symbolic inputs [19, 20, 21].

$x = x + y;$

$z = x * 3 + y;$

For example, above statements are executed concretely with 2 for x and 3 for y. For the first statement the result of x + y is 5 and the value 5 is assigned to x. Then, x * 3 + y part of the second statement is executed and 18 is found and it is assigned to z. The values of x, y and z variables are 5, 3 and 18, respectively. As it can be seen, the output values become concrete values due to concrete execution. For symbolic execution let x equals A and y equals B as symbolic values. The result of the first statement is A + B and A + B is assigned to x. For the second statement if x is replaced with A + B and y is replaced B, the result is (A + B) * 3 + B and it is assigned to z. The output values of symbolic execution for x, y and z variables are A + B, B and 3A + 4B that are symbolic values. It can be said that concrete execution ends up an instance of the possible results set. On the other side all entities of the possible results set are covered by symbolic execution. The other significant point is that path constraints are collected during symbolic execution. In concolic execution, they are used to create new test case inputs for the next runs.

Figure 2.3 illustrates a method just for explaining symbolic execution. It should be underlined that the method is a pointless method. All possible paths of the method in Figure 2.3 are provided below when symbolic execution is applied to the method [1]. A path constraint is one of the all possible execution traces of a program.

- x <= 3
- x > 3 & x + 2 < y
- x > 3 & x + 2 >= y

The algorithm steps of the concolic testing are basically as follows.

- Run the program with bottom element values such as null for all inputs
- Capture the path constraints exercised by the run
- Infer unexplored path constraints by inverting exercised path constraints
- Generate new concrete input values from the unexplored constraints
- Continue running the program with found input values
- Generate new inputs until all path constraints have been exercised or until the time limit is reached

Let us go back Figure 2.3 and apply concolic testing. Firstly, let x and y equal 0, so the condition x > 3 in line 2 is not satisfied. Thus, the statement in line 9 is executed and 0 value is returned by the method. Executed path constraint is x <= 3 and x > 3 is found by inverting the exercised path constraint. After that let 4 is generated for x by using found path constraint x > 3. The method is executed with 4 for x, 0 for y and the condition in line 2 is satisfied and in line 3 z is assigned to x + 2 that equals 6. The if condition in line 4 returns false and else branch is executed because 6 is greater than 0. Only second condition of found path constraint, x > 3 & x + 2 >= y, is reversed and new path constraint is x > 3 & x + 2 < y. After solving this path constraint, 4 is generated for x and 7 is generated for y. Next, the method is executed with 4 and 7 for x and y and the condition in line 2 returns true and in line 3 z is assigned to 6 which is calculated value of x + 2. The if condition in line 4 is satisfied because 6 is less than 7 and the method returns 6 that is value of z. Finally, there is no unexplored path constraint of the method, so algorithm of concolic testing is finished.

2.3 Related Work

There are several works that combine concrete and symbolic executions. Directed Automated Random Testing, DART, [7] seeks execution errors such as program crashes, non-termination of C programs and input values causing these errors. DART uses random testing and symbolic evaluation to generate new input values for possible program execution paths. Concolic Unit Testing Engine, CUTE, [8] is a subsequent work which has adjunct improvements in terms of the DART. CUTE implements both concrete and symbolic executions, called concolic execution, in order to solve the problem of automating unit testing with memory graphs as inputs. Another approach called SAGE (Scalable, Automated, Guided Execution) [22] implements a whitebox fuzz testing algorithm originated from symbolic execution and dynamic test generation. SAGE works on security testing of windows applications written with C, C++. Emmi et al. [23] generate automatic test inputs and database records for database applications with using concolic execution. Their implementation executes concrete and symbolic testing simultaneously similar to Mamoste.

Automated test input generation techniques has been applied to web applications domain. Halfond et al. [2] propose a specialized form of symbolic execution to identify interfaces for Java based web applications. They claim effectiveness of the testing techniques such as test input generation on the web applications increases with the help of their approach. Another work on Java based web applications is parameter mismatches in these applications by using a static analysis based approach [5]. Wassermann et al. [4] and Artzi et al. [3] apply concolic testing for automated test input generation to PHP dynamic web applications. Wassermann et al. target on SQL injection of PHP dynamic web applications which is orthogonal to our work. On the other hand, Artzi et al. develop a tool, Apollo, to detect crashes and malformed HTML outputs of PHP applications.

Although Mamoste and Apollo have similarities such as using an HTML validator as test oracle, they distinguish some specific points. First of all our target language ASP.NET is different from their target language PHP. ASP.NET is not a scripting language but PHP is a scripting language. Secondly, Mamoste composes HTTP requests as if a user enters inputs and fires events whereas Apollo simulates user interaction by transforming the PHP script. Third, Mamoste implements additional checking mechanisms about visited inputs, visited path constraints, and generated path constraints by concolic algorithm. Finally, our approach

is lightweight compared to Apollo since our tests are similar to unit testing where units are event handlers.

According to Di Lucca et al. [24] different components of the web applications such as web pages, forms or other web objects can be identified as a unit from the unit testing perspective. They divide the unit testing of the web applications into two categories: client page testing, server page testing. Mamoste detects malformed HTML outputs of the web applications. Thus, it can be said that HTML output is a unit of the client page testing category. An event of a web page can be considered a unit of the server page testing category.

There are several researchers working specifically on ASP.NET web applications. SAFELI, a static analysis framework, [6] identifies SQL injection attacks by inspecting MSIL bytecode of ASP.NET applications using symbolic execution. While SAFELI concentrates on SQL injection on ASP.NET web applications, Mamoste aims to detect execution errors and malformed HTML outputs of these applications. Another work Microsoft Pex [9] uses a hybrid technique that integrates concrete and symbolic executions so as to generate unit test for programs under test. Microsoft Pex creates unit tests for applications written with C#, Visual Basic and F# languages. Moles [10] is another component provided by Microsoft and it supports writing isolated unit tests. By working together Microsoft Pex and Moles can generate unit tests for ASP.NET applications since they are targeted on .NET framework. There is an important point that event handlers of ASP.NET pages are protected functions, but Microsoft Pex can implement unit tests only public behaviors of public classes. Therefore, in order to use Microsoft Pex event handlers of ASP.NET pages are required to convert into public functions that is a disadvantage of Microsoft Pex. On the other hand, Mamoste does not need this kind of conversions. In addition, it is not clear how assertions that are used in unit tests can capture errors in HTML outputs. In other words, Microsoft Pex does not deal with HTML validation. Microsoft Research Team[1] attempts to simulate IIS with Microsoft Pex and Moles to breed unit tests. On the other hand, Mamoste composes HTTP requests as if a user enters inputs and fires events. In a sense, Mamoste replaces users and browsers. In this case, IIS does its work by itself; i.e. Mamoste transmits requests to IIS and IIS responses the requests. Therefore, Mamoste manages concrete executions as they occur in real, concrete execution of Mamoste is not a simulation.

[1] http://research.microsoft.com

```
using System;
public class Program {
    // Pex for fun always selects the first comment line.
    public static int Puzzle(int x) {
        if (x > 0)
            x++;
        else
            x = -x;
        x = x - 3;
        if (x > 0)
            x = 1;
        else
            x = 0;
        return x;
    }
}
```

Figure 2.4: Code for illustrating Defective of Microsoft Pex

We have practiced Microsoft Pex with the code in Figure 2.4 and Microsoft Pex generates the results in Table 2.1. As it can be seen in the figure, there are four possible paths to execute. However, three of them are found by Microsoft Pex. It is possible to execute the fourth path with `x = 3`, but Microsoft Pex cannot catch this input. Therefore, it can be said that Microsoft Pex applies symbolic execution defectively. Experiments can be done on the web site of Microsoft Pex[2].

Table 2.1: Execution Results of Microsoft Pex

x	result	Output/Exception
0	0	
-1140850204	1	
1	0	

[2] http://www.pexforfun.com/default.aspx?language=CSharp&sample=_Template

CHAPTER 3

MAMOSTE

In order to solve problems specified in Section 1.1 we developed a tool, called Mamoste. We aimed to detect malformed HTML outputs of ASP.NET dynamic web applications and runtime execution errors of these applications while developing Mamoste. We propose Mamoste as an automated tool to find malformed HTML outputs generated by ASP.NET web applications and unhandled execution errors crashing these applications.

Mamoste adapts concolic testing technique which is a combination of concrete and symbolic executions to generate test inputs dynamically. The tool also considers page events as inputs which cannot be handled with concolic testing. In a sense, the tool performs unit testing with dynamic input generation where a unit is an event of tested page. Moreover, Mamoste receives help from an HTML validator to verify HTML outputs found by Mamoste.

In this chapter, we present our solution in detail. Firstly, we explain methodology and algorithm of the study. After that, we give design and implemetation of Mamoste in terms of system architecture and class diagrams. Then, we show how to write instrumentation on samples. Moreover, we explain installation and usage of the tool by giving an example.

3.1 Methodology

To reach high test coverage, we have adapted concolic testing which has been successfully applied on Java and C programs [7, 8]. Figure 3.1 shows the pseudo code of the concolic algorithm we implement. `Program P` and `Event E` under test are parameters of the procedure. Variable `R` contains results of previous executions. Execution results consist of path constraints, input values, outputs and bugs. Variable `toVisit` is used to keep input values for

```
parameters : Program P, Event E
result : Execution Results

1 R = ∅;
2 toVisit = EmptyQueue();
3 Enqueue(toVisit, EmptyInput());
4 repeat
5     input = Dequeue(toVisit);
6     {newPC, output, bug} = RunConcreteAndSymbolic(P, E, input);
7     if (newPC not in Visited Path Constraints of R)
8         merge {newPC, input, output, bug} into R;
9         newInputs = FindNewInputs(newPC, Path Constraints of R);
10        newInputs = newInputs - Inputs of R;
11        EnqueueAll(toVisit, newInputs);
12 until (Empty(toVisit) or TimeExpired())
13 return R;
```

Figure 3.1: Algorithm of Concolic Testing

next executions and an empty input is added to this variable in line 3 as a bottom element. Then, there is a loop between line 4 and 12. The program `P` and it's event `E` are executed concretely and symbolically in this loop. In line 5, a new input is taken from variable `toVisit` and concrete and symbolic executions are run with this input in line 6. There is a checking mechanism in line 7. This mechanism checks whether new path constraint captured by the current execution is in the visited path constraint list or not. If the path constraint is in the list, the algorithm passes to the next input. If it is not, results gathered by the current execution are attached to the variable `R` in line 8. Subroutine in Figure 3.2 is called to find new inputs in line 9. Then, the inputs which are explored before are eliminated in line 10. The rest of the inputs found are appended to variable `toVisit` in line 11. These steps continue until `toVisit` is empty or until time is expired.

Figure 3.2 shows the pseudo code of finding new inputs in concolic testing. The procedure gets two parameters, `pc` and `pcList`. The parameter `pc` denotes path constraint which is used to generate new path constraints and inputs. The parameter `pcList` keeps the path constraints that are visited and generated before. Variable `inputs` is used as a list that contains new found inputs. Variable `pc` is parsed to find each conjunct of it in line 2. There is a loop applied for each piece between line 3 and 9. In this loop, the algorithm negates the last conjunct for each prefix of the path constraint in line 4. Found new path constraint by negation is checked if it is in the `pcList` parameter or not in line 5. If it is, the algorithm passes the next prefix.

```
parameters : Path Constraint pc, Visited and Generated Path Constraints pcList
result : New Inputs

1 inputs = ∅;
2 c1 ∧ ··· ∧ cn = pc;
3 for i = 1...n do
4     newPC = c1 ∧ ··· ∧ c(i−1) ∧ ¬ci;
5     if (newPC not in pcList)
6        merge newPC into pcList;
7        newInput = Solve(newPC);
8        if (newInput not empty)
9           merge newInput into inputs;
10 return inputs;
```

Figure 3.2: Algorithm of Finding New Inputs in Concolic Testing

Otherwise, new path constraint is attached to the `pcList` in line 6. After that, the constraint solver is called to find a concrete input that satisfies the generated path constraint in line 7. If the constraint solver can find such a value, this new input is added to variable inputs that keeps return values of the procedure in line 9.

We consider three kinds of inputs in ASP.NET dynamic web pages: events, HTML controls, and values entered by clients into the controls. For events, we choose the `Page_Load` event as bottom element. First, we execute the page with the `Page_Load` event and nothing or `null` value for HTML controls. Then, we collect all HTML controls and events from generated HTML output. We put them into the event list and the HTML control list. This step is different from concolic testing approach described in Section 2.2 since it is not possible to capture events and controls of an HTML output by examining path constraints. If the execution fails, we get the error message and the path constraints up to the error. If not, we receive only the path constraints of the execution. We save data gathered from the execution into the indicated directory of the file system.

After parsing the captured path constraints, we place found variables and their values into the variable list used to keep variables and their domain set. Next, we apply finding new path constraints algorithm from the captured path constraints and find new path constraints. After solving new path constraints, we find new input values for the variables by using a constraint solver. Finally, we execute the page again with the new input values and the events. We continue these steps for all found events until all path constraints are exercised or until the

time limit is reached. After concolic execution is finished, we start to validate HTML outputs of the tested ASP.NET web page by using an HTML validator. It is emphasized that the point is not using HTML validator. The point is executing all possible branches of a web page and finding HTML outputs of those executions as well as run-time execution errors.

The variable list, which keeps variables and their domain set, grows as we explore path constraints. Initially, the variable list is empty which means running the page with `null` value for all variables. When we examine a path constraint, such as `num1 >= 0 && num2 >= 0`, the variables `num1` and `num2` are added to the list with domains set as integers. When a constraint contains a string variable, we add that variable into the list and set its domain to { `null`, “”, *value_in_constraint*, *random_string*}. Here *value_in_constraint* denotes the string constant used in the constraint. Such constants are added to the domain list as we encounter different string constants in the constraints. The *random_string* denotes a randomly generated string constant different than all the elements in the domain set. We keep a domain set for string variables so that the constraint solver can return a value satisfying a condition, for example, *city* ≠ “*ankara*”.

We have modified the concrete execution step in the general concolic execution algorithm as well. First, we do not perform the validation during concrete execution. Instead, we save the output of the concurrent execution along with the current path constraints and then check the correctness of the outputs, which are the generated HTML pages, separately. This separation of checking mechanism enables to run the executions and validation in parallel. Second, the inputs for concrete execution consists of events and user values. Here we trigger an event, such as page load event or click event of a button, and supply input parameters. This execution is similar to unit testing with dynamic input generation where a unit is an event.

Symbolic execution is used to drive path constraints from the executed control flow that is required for Mamoste. Symbolic execution parses the program and constructs a tree from the program. Usually, special statements are injected to the program tree so that symbolic execution understands the program. Such kind of statement injection is called instrumentation. There are two ways to perform instrumentation for symbolic execution: source code instrumentation and bytecode instrumentation. Source code or bytecode of the tested program is certainly required.

Mamoste is to be used on the development side as a white box testing tool [24]. We use source

```
1   protected void btnWriteDistricts_Click(object sender, EventArgs e) {
2       int provinceCode = Convert.ToInt32(txtProvinceCode.Text);
3       if (provinceCode > 0) {
4           IDataReader dr = GetDistricts(provinceCode);
5           while(dr.Read()) {
6               Response.Write("District Code : " + dr.GetValue(0).ToString() + "<br />");
7               Response.Write("District Name : " + dr.GetValue(1).ToString() + "<br />");
8           }
9       }
10      else
11          Response.Write("Wrong province code!");
12  }
```

Figure 3.3: Code for illustrating Need of Manual Instrumentation on Web Applications Using Database

code instrumentation while developing Mamoste in which the instrumentation is done manually by the users presently. We prefer the manual instrumentation to test dynamic web applications that use a database. We believe that a software application connected to a database cannot be tested fully self-driven since executed control flow is dependent on stored data in database. In other words, variables and conditions in the program are mostly attached to the database. Thus, we need to know about stored data in database to generate values for variables that are related to the database.

Figure 3.3 shows an event handler of a web page connected to a database. The event handler in Figure 3.3 gets province code from a web control and writes districts of that province to the screen if province code is greater than 0 and that province has any district. Otherwise, an error message is written to the screen.

Let's apply concolic execution to the code in Figure 3.3. For the first iteration, province code should be bottom element which is 0 for integers. Since province code is not greater than 0, `else` branch in line 10 is executed and `!(provinceCode > 0)` path constraint is catched. According to the algorithm of finding new inputs in Figure 3.2, the path constraint is inverted and `provinceCode > 0` path constraint is found. This new constraint is unexplored, so we give the new path constraint to the constraint solver. Suppose 100 is returned as a province code by the constraint solver. For province code is greater than 0, `if` branch in line 3 is executed in second iteration. However, there is no district belong to the current province (`provinceCode = 100`) in the database table of the district as seen in Table 3.1. Thus, in line 5 loop that writes districts to the screen is not executed. As province code is related to the database, we cannot execute inside of loop in line 5. If we instrument the path constraints manually, we have a chance to run the loop. For example, we can instrument

Table 3.1: Database Table of the Districts

ProvinceCode	DistrictCode	DistrictName
1	1	Seyhan
1	2	Ceyhan
63	1	Birecik
63	2	Suruç

`else` branch in line 10 with `!(provinceCode == 1 || provinceCode == 63)`. If we negate this constraint, we get `provinceCode == 1 || provinceCode == 63` as path constraint. The constraint solver gives 1 or 63 as a province code for this constraint. As a result, we can execute inside the loop with 1 or 63 inputs. Consequently, we do not have any solution for web applications that use database except manual instrumentation at least presently.

3.2 Design And Implementation

In this section, we explain system architecture of Mamoste and give technical information about Mamoste such as class diagrams. Lastly, explanation of how to do instrumentation for capturing path constraints is given with samples.

3.2.1 System Architecture

The system architecture of Mamoste is shown in Figure 3.4. Mamoste consists of three main components: Test Driver, Solver Adapter and Validation Driver. Below we explain each of these components in detail.

3.2.1.1 Test Driver

Test Driver is the main component of Mamoste. This component consists of three modules. The main module is Inspector. Inspector is responsible for running the test cases on ASP.NET page and capturing the path constraints during the execution. Constraint Converter is responsible to communicate solver Adapter component. Variable Domain Miner module is responsible for populating variable list that are used in the constraints as explained in Section 3.1.

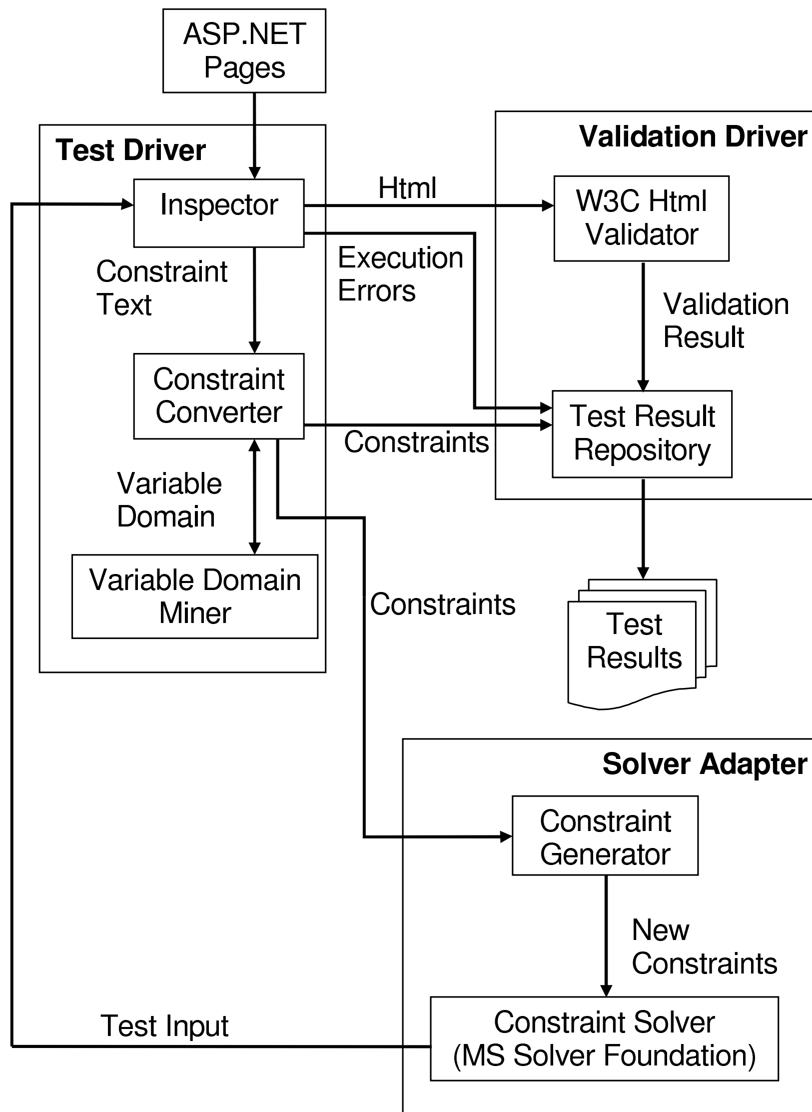

Figure 3.4: System Architecture of Mamoste

When Mamoste is loaded, Inspector takes the URL of an instrumented ASP.NET page. Then Inspector sends an HTTP GET request to the web server that results in loading the page under test. At the end of the execution the web server generates and sends an HTML output as a response to Inspector that is the client in view of the web server. After the first execution of the page, Inspector collects all the HTML controls and the events by parsing the HTML output as well as the future ones. Why Inspector applies parsing procedure each time to all HTML outputs is the probability of adding HTML controls to the page in run-time. It means that HTML controls might be added to the page dynamically during execution of an event. In order to catch these controls, Inspector has to take all HTML outputs into consideration. After finishing work on the HTML output, the module directs it to the Validation Driver component. The execution might be interrupted because of run-time error. In this case, Inspector sends error message to the Validation Driver component. Then, Inspector holds to execute the page with newly generated test case inputs until the input list is empty or time ends up. These steps are continued for all gathered events. There is a difference between the first execution and the others. Method of HTTP request has to be POST for all the events except page load. It has to be GET for page load event.

During an execution, Inspector module gathers the constraint texts of the executed branches via using the instrumentations described in Section 3.2.3. Inspector sends these captured constraint texts to the Constraint Converter module. This module transforms these texts into custom types of Mamoste. Also, path constraint tree is constructed during this transformation. While forming the constraint tree, Constraint Converter invokes the Variable Domain Miner module to expand the variable list and to populate the domain set of string variables as discussed in Section 3.1. Processed path constraints are sent to the Solver Adapter component to create new test case inputs for next executions. Also, these constraints are sent to the Validation Driver component to use during reporting test results.

The Inspector controls new test case inputs given by the Solver Adapter whether they are used in previous executions or not. If the Inspector has used an input before, that input is ignored and the Inspector passes to the next input. In addition, the path constraints that are captured during the executions are also checked by the Inspector whether they are visited before or not. If such a visited path constraint is met, that constraint is not taken into consideration. Even HTML output or execution error mining by that constraint is not given Validation Driver since there is a copy of that output in result list of Validation Driver. These two checking

mechanisms provide the Inspector and HTML Validator to work only when it is necessary.

3.2.1.2 Solver Adapter

This component is responsible for generating new test case inputs to explore new branches of the page under test. It gets formatted path constraints from the Test Driver component. Upon getting a path constraint, Constraint Generator module of this component creates new path constraints by employing a well established algorithm [7]. The new constraint generation is the same as other dynamic testing algorithms: Given $pc_1 \wedge pc_2 \wedge \cdots \wedge pc_n$, new constraints are $\neg pc_1, pc_1 \wedge \neg pc_2, \ldots, pc_1 \wedge pc_2 \wedge \cdots \wedge \neg pc_n$.

From the constraint set generated according to the algorithm in Figure 3.2, Constraint Generator selects the ones that have not been solved by the Constraint Solver before. If a constraint is not solved before, it is given to the Constraint Solver to produce new test case input for it. Otherwise, Constraint Generator ignores that constraint. Besides, Constraint Generator checks whether generated constraints are explored by the Test Driver or not. If Constraint Generator encounters such a visited constraint, it does not give that constraint to the Constraint Solver since new test case inputs found by the Solver will be a subset of the inputs generated before. These checking mechanisms prevent the Constraint Solver to generate unnecessarily new test case inputs same as before.

Mamoste uses MS Solver Foundation [25] for the generation of new test case inputs. Mamoste has to supply the domain of each variable that appears in the path constraint to the Solver, so the Solver can return values that make the given boolean formula (path constraint) true. As seen previous explanation, the generated values are used as new inputs by the Inspector.

Presently, Mamoste can handle variables of type integer, boolean, real and string. In the case of string, due to the Solver's abilities Mamoste can only support equality and inequality relations in constraints. Requests sent to the web server and responses the web server prepares are principally strings. We plan to support other custom types by linking them or their properties to the web controls of the tested page. This correlation provides us to deal with only primitive types. In other words, we need to capture correlation of the custom types and the controls of the web page in order to support all types.

3.2.1.3 Validation Driver and HTML Validation

Validation Driver component is responsible for communicating with HTML Validator and managing Test Result Repository. For validation Mamoste uses the HTML Validator of World Wide Web Consortium [26]. Each of the HTML outputs is sent to the Validator and the error/warning result messages are stored into Test Result Repository module.

Mamoste gives several choices about the HTML Validator. First one is that users can download and install W3C HTML Validator into their computers. Then, they can change the URL of the Validator with their local copy from the configuration file. Another opportunity is the ability to change validation mode: file upload and URL. This means that validation can be done by uploading the HTML output from file or by downloading it from URL. Result of the validation can be sent as HTML or XML files and Mamoste provides these two options. All these options are adjusted easily from the configuration file of Mamoste.

The Test Result Repository module holds HTML outputs and execution errors of the tested ASP.NET dynamic web pages along with the associated path constraints. Also, it saves the result of the Validator for each HTML output. This repository is used when the Test Results are displayed to the users.

3.2.2 Implementation

Our web application checker tool Mamoste is a desktop application and it is written in .NET Framework 3.5. However, it can test web applications implemented in .NET Framework 2.0 or later versions. It should be indicated that Mamoste is a white box testing tool which has to be used on the development side. Thus, it is interested source codes and client-side codes such as javascript and vbscript are out of its scope. Server-side codes are tested by Mamoste. In other words, Mamoste does not execute client-side codes because it works on the server-side in contrast to the web browsers.

Figure 3.5 displays the class diagram of the concolic package of Mamoste. The classes in this figure are responsible from the concolic testing of Mamoste. The `IDriver` interface provides required methods for the GUI of Mamoste. The `Driver` class is inherited from the `IDriver` interface. `Driver` class also includes other methods and fields to apply con-

Figure 3.5: Class Diagram of Concolic Package of Mamoste

Figure 3.6: Class Diagram of Trace Listener Package of Mamoste

colic testing. The path constraint texts captured by trace listener of Mamoste are read from the trace listener file by methods of this class. Then, these constraint texts are given the `Constraint` class to construct executed constraint tree and find constants, variables, operators, etc. During constructing constraint tree, the `Variable` class is used for variables and their domain values. Also, the `PathConstraint` class holds the executed path constraint and its variables. While the `ENUMConstraintType` enumeration is used to determine constraint types, `ENUMVarType` enumeration is used for variables. After generating new path constraints, the `Driver` class calls the `Solver` class to create new inputs for these constraints. The `HttpRequest` class executes the tested page with newly generated inputs. After each execution, the `HtmlParser` class parses detected HTML output to find new events and HTML controls of the page. During executions, the `TestLog` class is called for logging data of the executions. The `GeneralFunctions` and the `Constant` classes consist of required general functions and constants that are called by the other classes during testing. The `Validation` class is responsible to validate the found HTML outputs by using an HTML validator. The `TestResult` and the `FeedBack` classes is used to monitor results of the executions to the users.

Figure 3.6 displays the class diagram of the trace listener package of Mamoste. The class in this diagram is responsible to capture constraints of executed paths. The `TxtTraceListener` class writes gathered constraints into the trace listener file that is specified by the user explained in Section 3.3.1. Note that this trace listener file is used later by the `Driver` class in Figure 3.5.

Figure 3.7 displays the class diagram of the GUI package of Mamoste. The classes in this diagram are responsible to get users actions and show test results to the users. The `Program`, the `Settings` and the `Resources` classes belong to the .NET framework. The `Mamoste` class

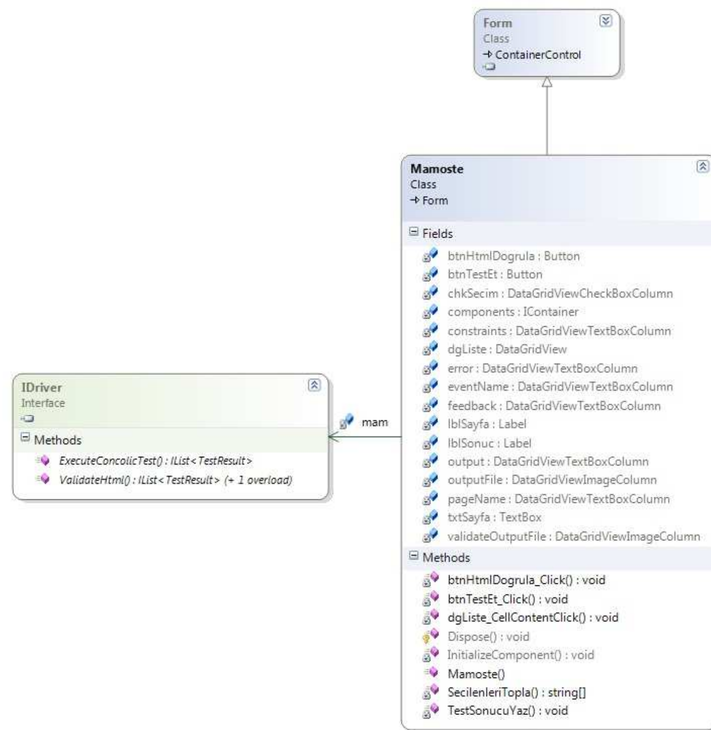

Figure 3.7: Class Diagram of GUI Package of Mamoste

provides interaction with the users and shows the test results on the screen.

3.2.3 Instrumentation

To collect the path constraints, Mamoste requires manual instrumentation. Manual instrumentation is done by using trace listener library of Mamoste, `Mamoste.TraceListener.dll`. This library has a class named `TxtTraceListener` and this class is inherited from the `TraceListener` class [27] of the `System.Diagnostics` namespace [28]. Users must instrument path constraints by calling `WriteLine` function of the trace listener class of Mamoste. Usage of this function is like `System.Diagnostics.Trace.WriteLine("a > 5")`. This instrumentation tells Mamoste that the current path is enabled when `a > 5` constraint holds. `!(a > 5)` constraint is negation of the `a > 5` constraint. To instrument this negated constraint, users have to call the function like `System.Diagnostics.Trace.WriteLine("!(a > 5)")` inside the related branch. Without these instrumentations Mamoste has to perform aliasing analysis to relate the variables with the controls of the page.

Table 3.2 illustrates several instrumentation examples. The first row is about instrumentation of integer variables. As seen at the first row of the table, original code does not contain `else` branch. However, instrumented code contains `else` branch since Mamoste needs to know

Table 3.2: Code and Instrumented Code Samples

Code	Instrumented Code

int a = Convert.ToInt32(txt.Text);
if (a > 5)
    Write("Variable a is greater than 5.");
	int a = Convert.ToInt32(txt.Text);
	if (a > 5)
	    System.Diagnostics.Trace.WriteLine("txt > 5");
	    Write("Variable a is greater than 5.");
	else
	    System.Diagnostics.Trace.WriteLine("!(txt > 5)");

string a = txt.Text;
if (!string.IsNullOrEmpty(a))
    Write("Variable a is not empty.");
else
    Write("Variable a is empty.");
	string a = txt.Text;
	if (!string.IsNullOrEmpty(a))
	    System.Diagnostics.Trace.WriteLine("txt != \"@@@\"");
	    Write("Variable a is not empty.");
	else
	    System.Diagnostics.Trace.WriteLine("!(txt != \"@@@\")");
	    Write("Variable a is empty.");

if (chk.Checked)
    Write("Checkbox is selected.");
else
    Write("Checkbox is not selected.");
	if (chk.Checked)
	    System.Diagnostics.Trace.WriteLine("chk == true");
	    Write("Checkbox is selected.");
	else
	    System.Diagnostics.Trace.WriteLine("!(chk == true)");
	    Write("Checkbox is not selected.");

int a = Convert.ToInt32(txt.Text);
for (int i = 0; i <= a; i++)
    if (i % 2 == 0)
        Write(i + " is even number less
            than or equal to " + a + ".");
    else
        Write(i + " is odd number less
            than or equal to " + a + ".");
	int a = Convert.ToInt32(txt.Text);
	if (a >= 0)
	    System.Diagnostics.Trace.WriteLine("txt >= 0");
	    for (int i = 0; i <= a; i++)
	        if (i % 2 == 0)
	            System.Diagnostics.Trace.WriteLine("txt % 2 == 0");
	            Write(i + " is even number less
	                than or equal to " + a + ".");
	        else
	            System.Diagnostics.Trace.WriteLine("!(txt % 2 == 0)");
	            Write(i + " is odd number less
	                than or equal to " + a + ".");
	else
	    System.Diagnostics.Trace.WriteLine("!(txt >= 0)");

which branch is executed each time. Otherwise, `if` branch of the code might not be executed by Mamoste. The second row shows instrumentation of string variables. The significant point of this example is using `@@@` characters instead of `empty` or `null` values of string variables. The third row illustrates instrumentation of boolean variables. Boolean variable can be `true` or `false`. The instrumentation of boolean varibles must be similar to `chk == true` or `chk == false` or negation of these. Constraint parser that Mamoste uses cannot handle `chk` and `!chk` as it expects equality or inequality that has left and right sides. Loop instrumentation is exemplified in the last row of the table. Loop code is placed in a `if` branch and condition of the `if` branch is derived from condition of the loop. In this example, `i <= a` condition is converted into `a >= 0` as the initial value of variable `i` is 0. As explained in the first row, `else` branch is also a requirement for Mamoste to be able to execute the loop. Note that the

conditions in the loop are instrumented as shown in the upper rows of the table.

3.3 Usage

In this section, we explain installations and configurations related to Mamoste. After installations and configurations, we describe how to use Mamoste on an ASP.NET web page.

3.3.1 Installation

Mamoste adopts MS Solver Foundation [25] as a constraint solver. Thus, users of Mamoste have to install MS Solver Foundation into their computers. Then, Mamoste requires several configurations such as in which local directory test results are saved and URL of the HTML Validator. Final one is several changes in configuration of the tested web application like adding the Trace Listener dll of Mamoste as a reference. The following sub-sections describe these requirements in detail.

3.3.1.1 Installation of MS Solver Foundation

MS Solver Foundation can be downloaded from the product web site [25]. Express edition of MS Solver Foundation is free to download, so it can be installed by the users. Installation of the component is easy nearly just clicking `Next` button several times. MS Solver Foundation supports 32-bit and 64-bit of Windows 7 and Windows Vista operating systems [29]. Users installing the component need administrator rights on the computer. Visual Studio 2008 or Visual Studio 2010 should be also installed on the computer. At least one of these programs might be already installed since target group of Mamoste consists of software developers.

3.3.1.2 Configuration of Mamoste

Files and directories provided by Mamoste can be copied into anywhere on the computer but they need to be all together. In other words, given package of Mamoste has to be saved same as given directory hierarchy. The package consists of below files and directories.

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <appSettings>
    <add key="OutputPath" value="C:\Mamoste\Output\" />
    <add key="TestLogFileName" value="TestLog.xml" />
    <add key="ValidateDirectoryName" value="Validate" />
    <add key="UrlOfValidator" value="http://validator.w3.org/check" />
    <add key="ValidationMode" value="FileUpload" />
    <add key="ValidationResultMode" value="Html" />
    <add key="TraceListenerFile" value="C:\inetpub\wwwroot\TestApplication\
        TraceListenerFile.txt" />
  </appSettings>
</configuration>
```

Figure 3.8: Configuration File of Mamoste

1. Output Directory : It is used for saving result files of the concolic executions.

2. Output\Validate Directory : It is used for saving result files of the HTML validations.

3. Output\Validate\images Directory : Files inside this directory are used to display in result files of the HTML validations.

4. Output\Validate\style Directory : Files inside this directory are used to display result files of the HTML validations in a well formatted.

5. Mamoste.Concolic.dll : This file is library for concolic testing of Mamoste.

6. Mamoste.exe : This windows application file is the graphical user interface of Mamoste.

7. Mamoste.exe.config : This file is configuration file of Mamoste.

8. RpnParser.dll [30] : This file is library for parsing captured constraint texts during the concolic executions.

9. Mamoste.TraceListener.dll : This file is library for listening execution traces of the tested web applications.

After saving the above files and directories, users have to open `Mamoste.exe.config` file with a text editor. This file seems like in Figure 3.8. Configurations inside this file have to be changed according to the directory users choose to save files and directories of Mamoste. Configuration keys of Mamoste that are seen in Figure 3.8 are explained below.

```
<system.diagnostics>
    <trace autoflush="true" indentsize="4">
        <listeners>
            <remove name="Default"/>
        </listeners>
    </trace>
</system.diagnostics>
```

Figure 3.9: Change in Configuration File of Tested Web Application for Trace Listener of Mamoste

- OutputPath Key : It denotes path in which result files of the concolic executions are saved.

- TestLogFileName Key : It denotes file that logs of the concolic executions are written. The extension of this file has to be `xml`. The directory in which this file is saved is represented by OutputPath Key.

- ValidateDirectoryName Key : It denotes directory name in which result files of the HTML validations are saved. The path in which this directory is located is represented by OutputPath Key.

- UrlOfValidator Key : It represents URL of the W3C HTML Validator. This key provides the users of Mamoste to install a copy of the W3C HTML Validator on their computer and use it locally. Thus, internet access requirement can be eliminated by this way.

- ValidationMode Key : Mamoste supports two validation modes of the W3C HTML Validator, file upload and URL validations. Consequently, this key can be set `FileUpload` and `URL` values.

- ValidationResultMode Key : Mamoste provides two output types for the validation result files, HTML and XML types. Thus, this key can be set `Html` and `Xml` values.

- TraceListenerFile Key : It denotes log file that keeps execution traces of the tested web applications. Execution errors and path constraint instrumentations are written into this file.

3.3.1.3 Configuration of Tested Web Application

Users of Mamoste have to set several configuration properties in their web applications for listening execution traces [27] and capturing path constraints and run-time errors. Firstly, they need to add Mamoste.TraceListener.dll trace listener library to references of their applications. After adding trace listener reference, Web.config, Web Configuration File [31], and Global.asax, Global Application Class [32], files must be also modified. There are two changes in Web.config file. First one is adding <trace enabled="true" writeToDiagnosticsTrace="true"/> config section row between the <system.web> and </system.web> tags. Then, users need to add config section [33] in Figure 3.9 between the <configuration> and </configuration> tags. Finally, Global.asax file must be configured. If the tested web applications do not have this file, users are supposed to add it in their projects. Code in Figure 3.10 have to be added in Global.asax file. The code in Figure 3.10 runs for every request, clears list of trace listener and adds the trace listener of Mamoste to the list. The path seen in Figure 3.10 represents path of trace listener file and this path must be same path of TraceListenerFile Key in Figure 3.8.

Table 3.3: Test Results of Given Example

Event	Path Constraint	Test Result	Description
Page_Load		Valid, 1 warning	No Character Encoding Found!
btnDivide	$txtNum1 >= 0$ && $txtNum2 >= 0$	Unhandled Execution Error	Attempted to divide by zero!
btnDivide	!($txtNum1 >= 0$ && $txtNum2 >= 0$)	Invalid, 2 errors, 1 warning	No Character Encoding Found! End tag for "br" omitted! End tag for "b" omitted!
btnMultiply		Unhandled Execution Error	Input string was not in a correct format!

```
protected void Application_BeginRequest(object sender, EventArgs e)
{
    System.Diagnostics.Trace.Listeners.Clear();
    System.Diagnostics.Trace.Listeners.Add(new
    Mamoste.TraceListener.TxtTraceListener("C:\\inetpub\\wwwroot\\
    TestApplication\\TraceListenerFile.txt"));
}
```

Figure 3.10: Change in Global Application Class of Tested Web Application for Trace Listener of Mamoste

```
1  using System;
2  namespace Example {
3      public partial class DivideAndMultiply : System.Web.UI.Page {
4          protected void Page_Load(object sender, EventArgs e) {
5              if (!IsPostBack) {
6                  txtNum1.Text = "Enter a number...";
7                  txtNum2.Text = "Enter a number...";
8                  lblResult.Text = "Result will be displayed here...";
9              }
10         }
11         protected void btnDivide_Click(object sender, EventArgs e) {
12             int num1 = 0;
13             int num2 = 0;
14             try {
15                 num1 = Convert.ToInt32(txtNum1.Text.Trim());
16                 num2 = Convert.ToInt32(txtNum2.Text.Trim());
17             }
18             catch (Exception) {}
19             if (num1 >= 0 && num2 >= 0) {
20                 System.Diagnostics.Trace.WriteLine("txtNum1 >= 0
                        && txtNum2 >= 0");
21                 lblResult.Text = "Division : " + (num1 / num2).ToString();
22             }
23             else {
24                 System.Diagnostics.Trace.WriteLine("!(txtNum1 >= 0
                        && txtNum2 >= 0)");
25                 lblResult.Text = "<br><b>Please enter natural numbers...";
26             }
27         }
28         protected void btnMultiply_Click(object sender, EventArgs e) {
29             int num1 = Convert.ToInt32(txtNum1.Text.Trim());
30             int num2 = Convert.ToInt32(txtNum2.Text.Trim());
31             if (num1 >= 0 && num2 >= 0) {
32                 System.Diagnostics.Trace.WriteLine("txtNum1 >= 0
                        && txtNum2 >= 0");
33                 lblResult.Text = "Multiplication : " + (num1 * num2).ToString();
34             }
35             else {
36                 System.Diagnostics.Trace.WriteLine("!(txtNum1 >= 0
                        && txtNum2 >= 0)");
37                 lblResult.Text = "<br><b>Please enter natural numbers...";
38             }
39         }
40     }
41 }
```

Figure 3.11: Instrumented Application Logic File of DivideAndMultiply ASP.NET Web Form

3.3.2 Example

Figure 3.11 illustrates the instrumented application logic file of the DivideAndMultiply ASP .NET web form in Figure 2.2. Statements in line 20, 24, 32 and 36 are instrumentations that users have to write in their program. As seen in Figure 3.11, the instrumentations are similar to the explanation in Section 3.2.3. These instrumentation statements assist Mamoste to gather path constraints executed.

After running Mamoste on the given example, test results are collected as shown in Table 3.3. According to the first row of the table HTML output of `Page_Load` execution is valid but character encoding is not found in the output. Second row shows that there is a run-time error during execution of `btnDivide`. The error attempting to divide by zero occurs in `txtNum1 >= 0 && txtNum2 >= 0` execution path in line 21 of Figure 3.11. Second execution of `btnDivide` event results in invalid HTML output with 2 errors and 1 warning as it can be seen in third row of the table. The errors are about missing end tag and warning is again no character encoding. Last row of the table displays an execution error which occurs in `btnMultiply` event. The error is incorrect input string which rises at the beginning of the event in line 29 or 30 of Figure 3.11. For there is no path constraint instrumentation up to that line, the path constraint column of the table is empty. Consequently, Mamoste detects 1 malformed HTML output, warnings in HTML outputs and 2 execution errors. These results indicate that Mamoste should be used to detect malformed HTML outputs and unhandled execution errors of ASP.NET dynamic web applications.

CHAPTER 4

EXPERIMENTS

We have used our tool Mamoste to check a subset ASP.NET web pages of the SGB.net system of Ministry of Finance of Turkey. This web application is used by several government organizations besides this ministry. Therefore, there are numerous active clients interacting with this system. In fact, only in Ministry of Finance there are nine to ten thousand clients accessing and performing several tasks in the SGB.net system. Removing faults in this kinds of systems plays an important role due to its huge number of clients.

We have applied Mamoste to a subset that includes five ASP.NET dynamic web pages of the SGB.net system. The names of pages are `OlcuBirim`, `Ambar`, `AmortismanSinir`, `AmortismanSure`, and `Bolge`. The numbers of ASP.NET web controls in these pages are respectively 10, 20, 10, 13, and 13. In general, the web applications that use database simply perform listing, saving, and deleting operations on data. These operations are also performed in our subset under test. Accordingly, each page of our subset has four events: listing of records, saving of records, deleting of records, and clearing the page. The sizes of these pages are provided in the second column of Table 4.1. The minimum size is 179 and the maximum size is 354.

We have done three experiments on the selected subset of the SGB.net system. The first experiment is applying Mamoste to the selected subset in the last version (v2) of the SGB.net. In the second experiment, manual testing is performed on the same subset of the SGB.net v2. Finally, we tested the previous version (v1) of the SGB.net via using Mamoste. While doing the first and the second experiments, we aimed to compare test results of Mamoste and manual testing for a view of effectiveness of Mamoste. The reason that we applied Mamoste to v1 and v2 is to find number of errors that the developers have corrected and to reveal the maintenance success rate of the developers and Mamoste.

Table 4.1: Comparison of Manual Testing and Mamoste

Web Page Name	LOC	Method	VC	TC	IG	H#
OlcuBirim	184	Mamoste	4	7	4	6
		Manual	3		11	3
Ambar	354	Mamoste	17	19	17	17
		Manual	9		38	10
AmortismanSinir	179	Mamoste	4	7	4	6
		Manual	3		10	4
AmortismanSure	214	Mamoste	8	10	10	10
		Manual	5		23	4
Bolge	276	Mamoste	11	13	8	9
		Manual	6		19	4

4.1 Applying Mamoste to SGB.net v2

Recall that Mamoste focuses on detecting execution errors and malformed HTML outputs. When we used Mamoste on the SGB.net v2, we have found no execution errors. This result was expected as the system has been used excessively by the government offices and numerous tasks have been performed daily; hence the system is being tested every day. On the other hand, Mamoste has found a number of faulty generated HTML outputs. In fact, the numbers of warnings and errors found are more than expected by the developers of the system. Mamoste found 319 HTML errors and 117 warnings, excluding 47 HTML errors and 21 warnings that repeat in every page because of a reused component.

Mamoste has surfaced two important errors in the system. The first one is as follows. There is an ASP.NET control used in almost every page of the SGB.net. Mamoste discovered that in some of the dynamically generated HTML outputs, this control is repeated more than once and all of the occurrences have the same properties. The second dramatic error is because of a menu component of the system. This component is used by nearly all the ASP.NET pages. Mamoste has found 47 HTML errors and 21 warnings only in this menu component. In other words, because of this menu component, there are at least 47 errors and 21 warnings in every single page.

4.2 Testing SGB.net v2 Manually

We have inspected and tested the same subset manually in order to compare the test results and reason about the effectiveness of Mamoste. Results of manual testing and testing with Mamoste are displayed in Table 4.1. The LOC column denotes the number of line of code in the page, the Method column denotes testing approach type, the VC column denotes the number of constraints explored, the TC column denotes the number of total constraints of page under test. The number of inputs generated by the constraint solver is shown in column IG. The column H# shows the number of HTML outputs generated. According to the table, manual testing found less HTML outputs while using more test inputs. On the other hand, Mamoste discovered more HTML outputs while using less test inputs. Moreover, Mamoste executed more path constraints than manual testing as it can be seen in VC column of Table 4.1. This means that code coverage is improved when using Mamoste significantly.

4.3 Applying Mamoste to SGB.net v1

As the third experiment, we ran Mamoste on the same subset of SGB.net v1. Our goal was to compare Mamoste with the maintenance team of the system in terms of HTML fault detection. Table 4.2 shows HTML faults of SGB.net v1 and v2 and these faults are categorized by the error type. The column labeled as App. Version shows the versions of the system under test. The Cnt column shows the number of occurrence of the error, the column labeled as W/E shows whether the row represents an error or warning. The last column provides the ratio of corrected error numbers to total error numbers of v1 while upgrading to v2. Our experiment revealed that in v2 there are 319 HTML errors and 117 warnings excluding the errors caused by the menu component in every page. Interestingly, Mamoste found 407 HTML errors and 146 warnings in v1. As seen in the last column of the table, only a small fraction of these errors in the earlier version had been either found and corrected or unconsciously corrected by the developers. The outcomes of this experiment support our approach. The first outcome is that Mamoste finds more HTML errors than the maintenance team of the system. The second one is that Mamoste increases the success ratio of error detection and correction.

Table 4.3 illustrates HTML faults of the menu component, which is reused by all pages in this system, both in v1 and v2. As it can be seen from the table, these errors are basically about

Table 4.2: HTML Faults found by Mamoste

Error Type	App. Version	Cnt	W/E	Maintnc Success Rate (%)
No attribute	v2 v1	9 12	E	25
Element not allowed	v2 v1	133 172	E	23
Cannot generate system identifier	v2 v1	54 67	W	19
No system identifier could be generated	v2 v1	54 67	E	19
Undefined entity	v2 v1	54 67	E	19
Reference not terminated by REFC delimiter	v2 v1	54 67	W	19
Missing attribute	v2 v1	38 47	E	19
Duplicate specification	v2 v1	22 30	E	27
End tag for unfinished element	v2 v1	9 12	E	25
Character not allowed	v2 v1	9 12	W	25

Table 4.3: HTML Faults of Menu Component found by Mamoste

Error Type	App. Vers.	Cnt	W/E
No Character encoding	v2 v1	2 1	W
No attribute	v2 v1	1 1	E
Element not allowed	v2 v1	34 10	E
Cannot generate system identifier	v2 v1	3 2	W
No system identifier could be generated	v2 v1	8 3	E
Undefined entity	v2 v1	3 2	E
Reference not terminated by REFC delimiter	v2 v1	8 3	W
Reference to external entity in attribute value	v2 v1	8 3	W
Missing end tag	v2 v1	1 1	E

missing, unfinished or wrong HTML elements and attributes. Unlike Table 4.2, number of errors in the menu component of v2 is more than those of v1. The underlying reason is that the menu component was implemented over again in a new manner. This case also supports usage of Mamoste since the errors in the new menu component are detected by Mamoste.

To conclude, we have reached several significant results through these experiments. First, Mamoste improves efficiency of test in terms of using less test inputs and collecting more HTML outputs and execution errors. Second, Mamoste increases the code coverage by executing different branches of the tested program. Finally, Mamoste increases the success ratio of HTML error detection in ASP.NET web applications.

CHAPTER 5

CONCLUSION

In this thesis we presented an automated tool called Mamoste to detect execution errors and malformed HTML outputs of ASP.NET dynamic web applications. Mamoste applies concolic testing in generating test cases and considers page events as test input as well.

Our experiments revealed numerous HTML bugs on a highly used ASP.NET application and including a faulty component which is used almost every page of this application. It also showed that some of generated HTML outputs have the same control more than once. In fact, Mamoste detected the errors that lived through the versions of this application and that showed effectiveness of Mamoste.

There are several novelties of this study. Firstly, Mamoste prepares HTTP requests same as the web browsers. While doing this, it injects inputs into the requests as if they are entered by the clients. Secondly, Mamoste finds events of the tested page and uses these events as unit test elements during concolic testing. This procedure cannot be handled by concolic testing. Thirdly, Mamoste performs several checking mechanisms to minimize input generation and HTML validation.

There are some limitations of Mamoste. Firstly, the instrumentation that Mamoste needs to catch branch conditions is manual. We plan to remove this manual instrumentation as a future work. Second limitation is that Mamoste can support only primitive variable types; integer, boolean, real and string. We are going to support other custom types by linking them or their properties to the web controls of the tested page. At present only equality and inequality relations of string variable type are supported due to the limitations of the constraint solver used. Hence, string operations are also a limitation of Mamoste. We plan to include string operations to be used in string constraints such as subset and prefix.

REFERENCES

[1] Gareth Lee, John Morris, Kris Parker, Gary A. Bundell, and Peng Lam. Using symbolic execution to guide test generation. *Software Testing, Verification and Reliability (STVR)*, 15(1):41–61, March 2005.

[2] William G. J. Halfond, Saswat Anand, and Alessandro Orso. Precise interface identification to improve testing and analysis of web applications. In *Proceedings of the 18th International Symposium on Software Testing and Analysis (ISSTA)*, pages 285–296, 2009.

[3] Shay Artzi, Adam Kieżun, Julian Dolby, Frank Tip, Danny Dig, Amit Paradkar, and Michael D. Ernst. Finding bugs in web applications using dynamic test generation and explicit state model checking. *IEEE Transactions on Software Engineering (TSE)*, 36(4):474–494, July/August 2010.

[4] Gary Wassermann, Dachuan Yu, Ajay Chander, Dinakar Dhurjati, Hiroshi Inamura, and Zhendong Su. Dynamic test input generation for web applications. In *Proceedings of International Symposium on Software Testing and Analysis (ISSTA)*, pages 249–260, 2008.

[5] William G. J. Halfond and Alessandro Orso. Automated identification of parameter mismatches in web applications. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, pages 181–191, 2008.

[6] Xiang Fu, Xin Lu, Boris Peltsverger, Shijun Chen, Kai Qian, and Lixin Tao. A static analysis framework for detecting sql injection vulnerabilities. In *Proceedings of the 31st Annual International Computer Software and Applications Conference (COMPSAC)*, pages 87–96, 2007.

[7] Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart: Directed automated random testing. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 213–223, 2005.

[8] Koushik Sen, Darko Marinov, and Gul Agha. Cute: A concolic unit testing engine for c. In *Proceedings of the 10th European Software Engineering Conference (ESEC) held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, pages 263–272, 2005.

[9] Nikolai Tillmann and Jonathan De Halleux. Pex: white box test generation for .net. In *Proceedings of the 2nd International Conference on Tests and Proofs (TAP)*, pages 134–153, 2008.

[10] Microsoft Research Team. Moles - isolation framework for .net. `http://research.microsoft.com/en-us/projects/moles`. Last visited: March 2011.

[11] W3Schools. Asp tutorial. http://www.w3schools.com/asp/default.asp. Last visited: February 2011.

[12] The PHP Group. What is php? http://www.php.net. Last visited: February 2011.

[13] Microsoft Corporation. Microsoft asp.net. http://www.asp.net. Last visited: April 2011.

[14] Oracle. Javaserver pages technology. http://www.oracle.com/technetwork/java/javaee/jsp/index.html. Last visited: March 2011.

[15] Paul D. Sheriff. Introduction to asp.net and web forms. http://msdn.microsoft.com/en-us/library/ms973868.aspx. Last visited: April 2011.

[16] Microsoft Corporation. The official microsoft iis site. http://www.iis.net. Last visited: April 2011.

[17] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. Exe: Automatically generating inputs of death. *ACM Transactions on Information and System Security (TISSEC)*, 12(2).

[18] James C. King. Symbolic execution and program testing. *Communications of the ACM (CACM)*, 19(7):385–394, July 1976.

[19] Corina S. Păsăreanu and Willem Visser. A survey of new trends in symbolic execution for software testing and analysis. *International Journal on Software Tools for Technology Transfer (STTT)*, 11(4):339–353.

[20] Christoph Csallner, Nikolai Tillmann, and Yannis Smaragdakis. Dysy: dynamic symbolic execution for invariant inference. In *Proceedings of the 30th International Conference on Software Engineering (ICSE)*, pages 281–290, 2008.

[21] Christophe Meudec. Atgen: automatic test data generation using constraint logic programming and symbolic execution. *Software Testing, Verification and Reliability (STVR)*, 11(2):81–96, June 2001.

[22] Patrice Godefroid, Michael Y. Levin, and David A Molnar. Automated whitebox fuzz testing. In *Proceedings of Network Distributed Security Symposium (NDSS)*. Internet Society, 2008.

[23] Michael Emmi, Rupak Majumdar, and Koushik Sen. Dynamic test input generation for database applications. In *Proceedings of International Symposium on Software Testing and Analysis (ISSTA)*, pages 151–162, 2007.

[24] Giuseppe A. Di Lucca and Anna Rita Fasolino. Testing web-based applications: The state of the art and future trends. *Information and Software Technology (IST)*, 48:1172–1186.

[25] Microsoft. Ms solver foundation. http://www.solverfoundation.com. Last visited: March 2011.

[26] World Wide Web Consortium. Html validator. http://validator.w3.org. Last visited: February 2011.

[27] Microsoft. Tracelistener class. http://msdn.microsoft.com/en-us/library/system.diagnostics.tracelistener.aspx. Last visited: April 2011.

[28] Microsoft. System.diagnostics namespace. http://msdn.microsoft.com/en-us/library/15t15zda.aspx. Last visited: April 2011.

[29] Microsoft. Installing solver foundation. http://msdn.microsoft.com/en-us/library/ff524499(v=vs.93).aspx. Last visited: April 2011.

[30] The Code Project Open License (CPOL). General expression parser and evaluator. http://www.codeproject.com/KB/recipes/GenExpParser.aspx. Last visited: March 2011.

[31] Microsoft. Editing asp.net configuration files. http://msdn.microsoft.com/en-us/library/ackhksh7(v=VS.100).aspx. Last visited: April 2011.

[32] Microsoft. Global.asax syntax. http://msdn.microsoft.com/en-us/library/2027ewzw.aspx. Last visited: April 2011.

[33] Microsoft. <system.diagnostics> element. http://msdn.microsoft.com/en-us/library/1txedc80.aspx. Last visited: April 2011.