DETECTING EXECUTION AND HTML ERRORS IN ASP.NET WEB APPLICATIONS[*]

Mehmet Erdal Özkınacı and Aysu Betin Can
Informatics Institute, Middle East Technical University, Ankara, Turkey

Keywords: Automated Testing, Concolic Testing, Dynamic Web Pages, ASP.NET.

Abstract: Dynamic web applications are becoming widespread nearly in every area. ASP.NET is one of the popular development technologies 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 execution 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 a heavily used ASP.NET application of a government office. We have found 366 HTML errors and a faulty component which is used almost every page in this application. In addition, Mamoste discovered that a common user control is misused in several dynamically generated pages.

1 INTRODUCTION

Web applications with dynamic content are becoming very popular almost in every business area such as banking, entertainment and government agencies. There are several reasons for this popularity. First, updating and maintenance of these applications do not require distribution and installation software. Second, they are accessible by any computer with Internet access, which means there are potentially millions of clients.

ASP.NET is an example of dynamic web application development technology. ASP.NET pages run typically on a server and generate HTML or XML pages to be 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. I.e., the static HTML or XML parts reside in a separate file from the code that handles events and generate dynamic parts. Although this separation enables code reuse, it makes the development process error-prone.

With the increasing amount of Internet users, the faults that crash the application, interrupt a transaction or cause to display inaccurate and incomplete information are not tolerable. Such errors reduce the credibility of the site and cause possible loss 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 web sites with static content. Not all execution paths may return correct HTML pages. For example, a code segment closing the end tag of a table may not be in the execution path because of a condition depending on input values of the application.

There are several studies analyzing ASP.NET web applications. SAFELI (Fu et al., 2007) detects SQL injection vulnerabilities statically by inspecting MSIL byte code of ASP.NET applications. Although it is an important problem, it is orthogonal to detecting malformed HTML outputs. Microsoft Pex and Moles (Pex) generate unit tests for .NET framework including ASP.NET applications. 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.

Recently researchers focus on automated testing of dynamic web applications. Halfond et al. extracts interfaces for Java based web applications to improve the effectiveness of test generation (Halfond et al., 2009). Artzi et al. target PHP applications and aim to catch faulty their HTML outputs (Artzi et al., 2010).

[*]This work is supported by STRATEK (www.stratek.com.tr) and TÜBITAK (www.tubitak.gov.tr)

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

Mamoste is based on concolic testing pioneered by DART (Godefroid et al., 2005) and CUTE (Sen et al., 2005). In concolic testing an application is executed using concrete input values. Then symbolic execution drives path constrains from the executed control flow. To perform symbolic execution, usually special statements are injected to either source code or byte code. By solving path constraints, concolic testing generates new concrete inputs to exercise unexplored paths. Mamoste applies this technique on ASP.NET applications. Hence, Mamoste is to be used on the development side as a white box testing tool. In addition to concolic testing, 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 a 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 versions of the same subset and compared the performance of development team to Mamoste. The experiment showed that Mamoste used less test inputs and collected more HTML outputs. Moreover, Mamoste increased code coverage by executing different branches of tested program.

The rest of the paper is organized as follows. Section 2 gives an ASP.NET example and introduces our instrumentations on it. Section 3 presents our methodology. Section 4 introduces Mamoste. Section 5 gives experimental results on a heavily used web application. Section 6 presents related work. Finally, Section 7 gives conclusions and future work.

2 ASP.NET EXAMPLE

ASP.NET is a web application framework to develop dynamic web sites and web services. (In this paper we only focus on dynamic web sites). The advantage of ASP.NET is that it allows the developers to use object oriented languages such as C# instead of scripting languages. This ability is enabled by the separation of the code from display items as opposed to interleaving them as in the use of PHP scripts.

Figure 1 and 2 show this separation. Figure 1 defines the widgets on the page. Here we have three labels, two text fields, and one button. We can add events to any of these widgets and bind them to custom event handlers. For instance, here `OnClick` of the `btnDivide` button is bound to `btnDivide_Click` handler which is implemented in the code file.

```
<\%@ Page Language="C#" AutoEventWireup="true"
    CodeBehind="Divide.aspx.cs" Inherits="Test.Divide" \%>
<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server"></head>
<body><form id="form1" runat="server"><div><div>
  <asp:Label ID="lblDivide" ForeColor="Red" runat="server">
  </asp:Label>  </div>
  <div>
  <asp:Label ID="lblNum1" Text="Number 1" runat="server">
  </asp:Label>
  <asp:TextBox ID="txtNum1" runat="server"></asp:TextBox>
  </div>
  <div>
  <asp:Label ID="lblNum2" Text="Number 2" runat="server">
  </asp:Label>
  <asp:TextBox ID="txtNum2" runat="server"></asp:TextBox>
  </div>
  <div><asp:Button ID="btnDivide" OnClick="btnDivide_Click"
  Text="Divide" runat="server"></asp:Button>  </div>
</div></form></body> </html>
```

Figure 1: Display file of the ASP.NET web page.

The first line in Figure 1 says that the code behind this page is implemented in C# and it resides in Divide.aspx.cs. This code file (Figure 2) contains the event handler implementations. In this example there are two event handlers: `Page_Load` and `btnDivide_Click`. `Page_Load` event is a default event of every web page. Here its handler initializes the textboxes (lines 5-7). The second one is a custom handler bound to `OnClick` event of the `btnDivide` button as discussed above. This handler performs a division and puts the result in label `lblDivide`. If numbers are negative, it puts an error message in boldface.

2.1 Instrumentation

Mamoste requires user instrumentations to gather event handlers and path conditions. To collect the event handlers, Mamoste needs the instrumentation `Mamoste.GUI.mam.FindEvents(this)` on the `Page_Load` function (see line 4 in Figure 2.) Page load is the first event of any ASP.NET web page and it is guaranteed to be executed. Through this command Mamoste is directed to collect all the events in the web page only when the page is first executed.

To collect path constraints, Mamoste requires a `Mamoste.GUI.mam.AddConstraint` instrumentation. Here, line 16 specifies that the current path is enabled when `txtNum1>=0 && txtNum2>=0` holds. Line

```
1 public partial class Divide:System.Web.UI.Page{
2  protected void Page_Load(object sender, EventArgs e){
3   if(!IsPostBack){
4     Mamoste.GUI.mam.FindEvents(this);//instrumentation
5     txtNum1.Text="Enter a number..";
6     txtNum2.Text="Enter a number..";
7     lblDivide.Text="Division will be displayed here..";
8    }}
9 protected void btnDivide_Click (Object s, EventArgs e){
10  int num1=0; int num2=0;
11  try{
12     num1=Convert.ToInt32(txtNum1.Text.Trim());
13     num2=Convert.ToInt32(txtNum2.Text.Trim());
14  }catch (Exception){}
15  if(num1>=0 && num2>=0){
16    Mamoste.GUI.mam.AddConstraint//instrumentation
                    ("txtNum1>=0 && txtNum2>=0");
17    lblDivide.Text="Division:"+(num1/num2).ToString();}
18  else{
19   Mamoste.GUI.mam.AddConstraint //instrumentation
                   ("txtNum1>=0 && txtNum2>=0",true);
20  lblDivide.Text="<br><b>Please enter natural numbers";}}}
```

Figure 2: Divide.aspx.cs, the code behind the example page.

19 specifies that the path is taken when the *negation* (indicated as `true` as the second parameter) of its condition holds. Without these instrumentations Mamoste has to perform aliasing analysis to relate the variables with the labels of the controls (We are working on pointsTo analysis to replace these instrumentations.) Mamoste can capture these instrumentations in the event handlers and the functions they call if they are about the same web page since instrumentations are about the web controls of tested page.

3 METHODOLOGY

We propose a methodology to generate test cases dynamically for executing every branch of an ASP.NET application in order to collect all possible different HTML outputs and run-time errors. To achieve coverage, we have adapted the concolic testing technique which is successfully applied on Java and C programs (Godefroid et al., 2005; Sen et al., 2005). This technique runs and test the program with bottom element for all inputs, captures the path constraints exercised by the run, infers unexplored path constraints, generates new concrete input values from these constraints with the help of a constraint solver, and loops until there are no new input values generated or timeout.

We consider two kinds of inputs in ASP.NET dynamic web sites: events and values entered by clients. For events, we choose the `Page_Load` event as bottom element. When the page is first executed we collect all the events and put them in our input stack. This step is different from the concolic testing approach described above since it is not possible to capture the events on a page by examining path constraints.

To handle the values entered by clients, we keep a variable list and their domain set. This list 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 `txtNum1>=0 && txtNum2>=0` in line 16 in Figure 2, the variables `txtNum1` and `txtNum2` are added to the list with domains set as integers. When a constraint has a string variable, we add that variable into the list and set its domain to $\{$`null`, “”, *const*, *rand*$\}$. Here *const* is the string constant used in the constraint. Such constants are added to the domain list as we encounter different string constants in the constraints. *rand* denotes a randomly generated string constant different than 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, $city \neq$ “*ankara*”.

We have modified the concrete execution step in the 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 constraint 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 event and user values. Here we trigger an event, such as page load or onclick events, and supply input parameters. This execution is similar to unit testing where a unit is an event handler.

4 MAMOSTE

Our web application checker Mamoste is an ASP.NET application. It is written in .NET Framework 3.5 but it can test web applications implemented in version 2.0 or later. Mamoste tests server side code. Client side code written for example in javascript is out of our scope. Figure 3 shows the system architecture. Next we explain the components in this figure.

4.1 Test Driver

This component is responsible for main execution loop. When Mamoste is loaded, the Inspector module of this component takes the URI of an instrumented ASP.NET page. Then Inspector creates an HTTP request to the web server that results in loading the page under test. During this page load event, Inspector collects all the event handlers in the page. Then Inspector

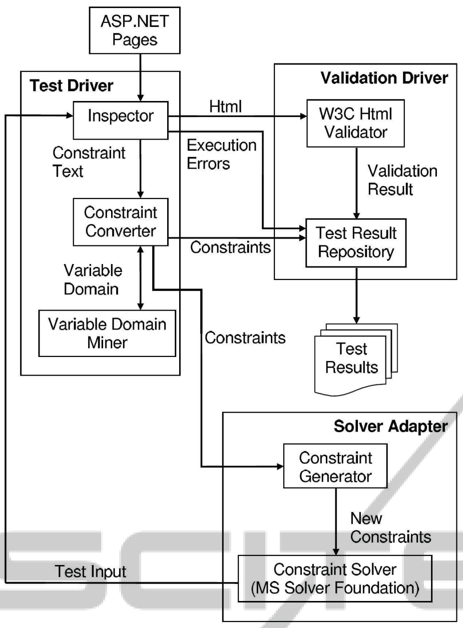

Figure 3: System Architecture of Mamoste.

creates HTTP requests to trigger each event handler with empty input values. With these event triggering and providing inputs to parameters, Mamoste simulates both a web browser and users in a sense.

During an execution, Inspector collects the constraints of the executed branches using the instrumentations. At the end of the execution, the server sends an HTML file as response. Inspector directs these HTML files or runtime error messages, if any, to the validator driver component. Then Inspector continues the execution loop with newly generated test inputs.

In this loop, path condition instrumentations are sent to Constraint Converter for transformation to the input format of the solver. We use MS Solver Foundation (MSS). While forming the constraints, this module 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.

4.2 Solver Adapter

This component is responsible for generating new inputs to explore new branches. Upon getting a path constraint from the Test Driver, Constraint Generator module creates new path constraints by employing a well established algorithm (Godefroid et al., 2005): Given $pc1 \wedge pc2 \wedge \ldots pc_n$, new constraints are $\neg pc_1$, $pc_1 \wedge \neg pc_2$, $\ldots$, and $pc_1 \wedge pc_2 \wedge \cdots \wedge \neg pc_n$. Out of these constraints, this module selects the ones that have not been explored before and passes them to MS Solver Foundation. This checking mechanism prevents the solver from unnecessary input generation. There is another checking mechanism after test input generation. The inputs are controlled whether they are used in concrete execution before or not.

We have to supply the solver the domain (value set) of each variable that appears in the constraints so that the solver can return the values that make the given boolean formula (the constraints) true. Currently, Mamoste can handle variables of primitive type and strings. Due to the solver's abilities, we can only support equality and inequality relations in string constraints. We plan to support custom types by linking them or their attributes to the web controls of the tested page. This relation provides us to deal with only primitive types. In other words, we need to capture the relation between custom types and web controls to handle no primitive types.

4.3 Validation Driver

Validation Driver is responsible for communicating with HTML validator and managing Test Result Repository. For validation we use the HTML validator of World Wide Web Consortium. The generated output files are sent to `http://validator.w3.org/` and the error or warning messages are stored in Test Result Repository. In this repository, each HTML file is stored along with the path constraints used in its generation and the validation results. This repository is used when the test results are displayed to users.

4.4 Back to Division Example

Let us explain how Mamoste works on the example given in Section 2. After getting the URI of the page, Mamoste sends an HTTP request to web server that triggers `Page_Load` handler with empty input. This handler first makes Mamoste to find the `Page_Load` and `btnDivide_Click` events through line 4 in Figure 2. Then, the handler initializes two text boxes and a label in lines 5, 6 and 7.

In the second run, `btnDivide_Click` event is triggered with empty input. Because of empty values for the textboxes `txtNum1` and `txtNum2`, the variables `num1` and `num2` equal 0. Thus, the first branch is taken in the if block (lines 15-17). Mamoste gets the path condition `txtNum1 >= 0 && txtNum2 >= 0`. Let this constraint be pc_1. Variable Domain miner module adds `txtNum1` and `txtNum2` into the variable list. Then Constraint generator module creates the path constraint `!(txtNum1 >= 0 && txtNum2 >= 0)`. Let this constraint be pc_2. MS Solver Foundation returns -1073741823 (the lower bound of integers) for both of the variables. These values become the next input values for Mamoste. The resulting HTML page is stored in the repository with the constraint pc_1.

In the third run, `btnDivide_Click` event is triggered with -1073741823 for both `txtNum1` and `txtNum2`. This case makes the else block (lines 18-20) execute. The instrumentation at line 19 says the path condition is the inverse of `txtNum1 >= 0 && txtNum2 >=0`, which is pc_2 in the previous run. Constraint generator creates a path condition $\neg pc_2$ which is exactly the same as pc_1. Since pc_1 is explored before, the solver is not invoked and no new input is generated. The resulting HTML page is stored in the repository with the constraint pc_2. Since no more inputs are left, the testing part ends.

Finally, the two resulting HTML outputs are validated by using W3C HTML validator. The validator finds the missing end tag `</br>` on the second file.

5 EXPERIMENTS

We have used Mamoste to check a subset of SGB.net system of Ministry of Finance of Turkey. This web application is used by several government organizations as well as the workers of the ministry. There are numerous users interacting with this system. There are nine to ten thousand users, only in Ministry of Finance, accessing and performing several tasks in the SGB.net system. Removing faults in this system plays an important role due to its number of users.

When we ran Mamoste on the SGB.net system, we 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 found a number of faulty generated HTML pages. In fact, the number of warnings and errors found are more than expected by the developers of the SGB.net. Mamoste found 319 HTML errors and 117 warnings, excluding 47 HTML errors and 21 warning that repeat in every page because of a reused component.

Mamoste surfaced two important errors in the system. The first one is about an ASP.NET control used in almost every page of the SGB.net. Mamoste discovered that in some of the dynamically generated HTML pages, 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. This component is used by nearly all the pages in the system. Mamoste has found 47 HTML errors and 21 warnings only in this menu component. Because of this menu component, there are at least 47 errors and 21 warnings in every single page.

We have inspected and tested the same subset manually to reason about the effectiveness of Mamoste. Table 1 shows the comparison of manual testing and Mamoste. The LOC column denotes the line of code in the page, 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 solver is shown in column IG. The column H# shows the number of HTML outputs generated. According to the table, compared to Mamoste, manual testing found less HTML outputs while using more test inputs. Moreover, Mamoste exercised more branching than manual testing as seen in VC column. This means that code coverage is improved with Mamoste significantly.

As a second experiment, we ran Mamoste on an older version of the same subset of SGB.net system. Our goal was to compare Mamoste with the maintenance team in terms of faulty HTML generation detection. Table 2 shows the number of HTML faults in SGB.net version1 (column V1) and SGB.net version2 (column V2) and their categorization (column labeled Error Type). The W/E column shows whether the fault is an error or a warning. The last column shows the ratio of the number of errors corrected to number of total errors while upgrading to version 2. Mamoste revealed that in version 2 there are 319 HTML errors and 117 warnings excluding the errors caused by the

Table 1: Comparison of Manual Testing and Mamoste.

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

Table 2: HTML faults found by Mamoste.

Error Type	W/E	V1	V2	%
No attribute	E	12	9	25
Element not allowed	E	172	133	23
Cannot generate system identifier	W	67	54	19
No system identifier could be generated	E	67	54	19
Undefined entity	E	67	54	19
Ref. not terminated by REFC delimiter	W	67	54	19
Missing attribute	E	47	38	19
Duplicate specification	E	30	22	27
End tag for unfinished element	E	12	9	25
Character not allowed	W	12	9	25

Table 3: Faults found in Menu component.

Error Type	W/E	V1	V2
No Character encoding	W	1	2
No attribute	E	1	1
Element not allowed	E	10	34
Cannot generate system identifier	W	2	3
No system identifier could be generated	E	3	8
Undefined entity	E	2	3
Reference not terminated by REFC delimiter	W	3	8
Reference to external entity in attribute value	W	3	8
Missing end tag	E	1	1

menu component. Interestingly, Mamoste found 407 HTML errors and 146 warnings in version 1. As seen in the last column, only a small fraction of these errors in the earlier version had been corrected by developers. The outcomes of this experiment are in favor of our approach. First, Mamoste found more HTML errors than found by maintenance team. Second, Mamoste increased the ratio of error detection.

Table 3 shows the faults in menu component. This component is used in all SGB.net pages in both versions. Unlike Table 2, the number of errors in version 2 is more than those of version 1. This increase is due to a reimplementation of the menu component.

These experiments show the followings. First, Mamoste improves efficiency of test in terms of using less test inputs and collecting more HTML outputs. Second, Mamoste increases code coverage by executing different branches of tested program than manual testing. Finally, Mamoste increases the success ratio of HTML error detection in ASP.NET sites.

6 RELATED WORK

Dynamic test generation is a promising research area. DART (Godefroid et al., 2005) employs a combination of symbolic execution and random testing for C programs. CUTE (Sen et al., 2005) employs concolic execution for unit testing Java programs with memory graphs as inputs. SAGE implements a whitebox fuzz testing based on symbolic execution and dynamic test generation. SAGE works on security testing of C and C++ windows applications. Emmi et al. generate test inputs for database applications using concolic execution (Emmi et al., 2007). Recently, automated test generation is being applied to web applications. Wassermann et al. (Wassermann et al., 2008) and Apollo (Artzi et al., 2010) use concolic testing for automated test generation to web applications in PHP. Wassermann et al. targets on SQL injection on PHP applications. Apollo works on crashes and malformed HTML outputs of PHP applications whereas we focus on ASP.NET applications which are heavyweight compared to PHP.

As for ASP.NET web applications, SAFELI, a static analysis framework (Fu et al., 2007), uses symbolic execution to identify SQL injections whereas Mamoste aims to detect execution errors and malformed HTML outputs of these applications. Microsoft Pex (Pex) uses concolic execution to generate unit test for C#, Visual Basic and F# applications. Working together with Moles for isolating units, Pex creates unit tests for ASP.NET applications. Since Pex works for public methods only, the modifier of event handlers needs to be changed for testing. For these tests, MS Research Team (MSR) simulates Internet Information Services (IIS) with Moles. In contrast, Mamoste composes http requests as if a user enters inputs and fires events; thus, working as both users and browsers and do not need to simulate IIS.

7 CONCLUSIONS

In this paper we presented an automated tool called Mamoste to detect execution errors and malformed HTML pages generated in ASP.NET web applications. Our experiments revealed numerous bugs on generated HTML files and a faulty component on a highly used ASP.NET application. Mamoste detected errors that lived through the versions of this application and showed its effectiveness.

Our limitations are as follows. First, Mamoste needs manual instrumentation to catch branch conditions and related to web inputs. We plan to remove this manual instrumentation in the future. Second, due to the constraint solver's limitation, currently, only equality and inequality of strings are supported. We plan to support other string operations in constraints, such as subset and prefix. Finally, Mamoste does not supports no primitive types, which could be avoided by linking custom types and their attributes the web controls of tested page.

REFERENCES

(Pex). http://research.microsoft.com/en-us/projects/pex/.

(MSS). http://www.solverfoundation.com.

(MSR). http://research.microsoft.com.

Artzi, S., Kieżun, A., Dolby, J., Tip, F., Dig, D., Paradkar, A., and Ernst, M. D. (2010). Finding bugs in web applications using dynamic test generation and explicit state model checking. *IEEE TSE*, 36(4):474–494.

Emmi, M., Majumdar, R., and Sen, K. (2007). Dynamic test input generation for database applications. In *Proc. of ISSTA*.

Fu, X., Lu, X., Peltsverger, B., and Chen, S. (2007). A static analysis framework for detecting sql injection vulnerabilities. In *Proc. of Computer Software and Applications Conference*, pages 87–96.

Godefroid, P., Klarlund, N., and Sen, K. (2005). Dart: Directed automated random testing. In *Proc. of PLDI*.

Halfond, W. G., Anand, S., and Orso, A. (2009). Precise interface identification to improve testing and analysis of web applications. In *Proc. of ISSTA*.

Sen, K., Marinov, D., and Agha, G. (2005). Cute: A concolic unit testing engine for c. In *Proc. of ESEC/FSE*.

Wassermann, G., Yu, D., Chander, A., Dhurjati, D., Inamura, H., and Su, Z. (2008). Dynamic test input generation for web applications. In *Proc. of ISSTA*.