

An Automated Tool for Data Flow Testing of ASP.NET Web Applications

Islam T. Elgendy^{1,*}, Moheb R. Girgis² and Adel A. Sewisy¹

¹Department of Computer Science, Faculty of Computers and Information, Assiut University, Egypt

²Department of Computer Science, Faculty of Science, Minia University, Egypt

Received: 1 Apr. 2020, Revised: 2 May 2020, Accepted: 22 Jun. 2020

Published online: 1 Jul. 2020

Abstract: The structure of ASP.NET web applications is very different from traditional ones. This paper discusses the adaptation of data flow testing techniques for such applications. In addition, it describes the data flow analysis of ASP.NET web applications, presents a proposed model for data flow testing of these applications, and describes the structure of a supporting automated tool. The tool generates the definition-use pairs for all variables in the application under test, then the application is executed with test data to cover these pairs, if possible. At the end of each test run, the tool shows the covered and uncovered pairs. The tool's report is generated in a highly organized format to improve the readability of the test results, and to reduce the time needed to review them. The paper also presents a case study to illustrate how the tool works. Finally, the paper presents the results of the experiments, which have been conducted to evaluate the viability of the data flow testing approach with the presented extensions for ASP.NET web applications, and the readability of the improved analysis and test report produced by the tool.

Keywords: Software testing, Data Flow Testing, Automated Testing Tool, Web Applications Testing

1 Introduction

Since the creation of the World Wide Web in the early 1990s [1], the use of Web applications (*WebApps*) has increased tremendously in our daily lives. WebApps are being used extensively in business, social, organizational, and governmental functions. In fact, the continual availability of WebApps has been one of the main reasons for their rapid spread among users without regard to their location or time limitations [2]. However, this demands high reliability of WebApps. Since inadequate testing poses huge risks including downtime and loss of users' trust and convenience, it is crucially important to ensure the quality, correctness, and security of WebApps.

Software testing has been widely used in industry as a quality assurance technique for the various software development stages. Nevertheless, software testing is very labor-intensive, time-consuming and expensive; almost 50% of the cost and 40% of the time of a software system development is spent on software testing [3, 4, 5]. An automated testing approach makes the process of testing much faster, less-expensive, more reliable, and

increases the quality of the software.

Random testing has shown its effectiveness in generating test data and covering several structural targets [6]. Still, through random testing, the chances of executing test targets, such as all branches and all variables definition-use associations, at random are very low. Therefore, to achieve such test targets, more intelligent techniques are required [7]. Another important testing issue is to improve the readability of the test results where testers spend a lot of time reviewing them, but some important parts might be missed because the final report involves much data. For instance, they might consider a test run as a "fail" when it is actually a "pass", or vice versa.

Meanwhile, webApps can be developed using various technologies and platforms, such as ASP.NET, PHP, JSP, etc. The study, presented in this paper, is concerned with testing WebApps developed using ASP.NET. We extend data flow testing techniques to WebApps, and present an approach to ASP.NET WebApps data flow testing. This study is an extension to the work done by Girgis et al. [8], where an approach for the construction of a WebApp data flow model to aid WebApp data flow

* Corresponding author e-mail: islam.elgendy@aun.edu.eg

analysis was suggested. Our work considers the data flow analysis of more ASP.NET server controls, and addresses the issues not considered in [8], such as (i) handling hyperlinks and the transfer between pages, (ii) defining the definitions of the used server controls' properties, (iii) handling the definitions of some of the unique features in WebApps, like session objects and query strings, and (iv) improving the readability of the test results to reduce the reviewing time by ordering and highlighting the key parts in the report.

The layout of the paper is, as follows: Section two presents the related work in WebApp testing. Section three, briefly, describes data flow analysis, and the related data flow testing criteria. Section four describes the suggested data flow model for WebApps. Section five presents a description of the automated testing tool. Section six presents a case study to illustrate the application of the automated testing tool. Section seven presents an empirical evaluation of the proposed data flow testing approach with the presented extensions for WebApps and the readability of the analysis and test report produced by the tool. Finally, Section eight is devoted for concluding remarks and future research.

2 Related Work

A comprehensive survey of Web testing advances was presented by Li et al. [5], where they discussed their goals, targets, techniques employed, inputs/outputs and stopping criteria. They stated that there are different testing goals, such as finding faults, ensuring testing adequacy, etc., and for each goal a different testing technique might be used to satisfy it. Lakshmi and Mallika [9] presented a comparative study of some of the prominent tools, techniques, and models for WebApp testing. Their work highlights the current research directions of some of the WebApp testing techniques.

This section presents a review of the published research work related to the data flow testing of WebApps.

Liu et al. [10] have presented a WebApps data-flow testing approach, which are implemented in XML and HTML languages, and include interpreted scripts and other types of executable constructs (such as Java beans, ActiveX controls, Java applets, etc.) at the client side and the server side of the application. The approach is based on a test model of WebApp, WATM, which includes an Object model, in which components are modeled as objects, and a Structural model that gets the data flow information of methods inside or across objects. In this approach, Liu et al. have derived test cases from three different views: intra-object, inter-object, and inter-client. They have defined 5 testing levels specifying different domains of the tests to be run. These different levels are: Function, Function Cluster, Object, Object Cluster, and Application level.

Ricca and Tonella [11, 12] have presented an approach

for a static WebApps white-box testing based on two test models: the Navigational model, which focuses on HTML pages and hyper links of the application, and the Control flow model, which represents the Web pages internal structure in terms of the followed execution flow. The Control flow model has been used to perform structural testing, as well. A test case is a sequence of pages to be visited, and the input values to be given to pages that contain forms. Test cases are designed using some control- and data-flow coverage criteria applicable on the two models.

Mansour and Hourri [13] have proposed white box techniques for testing .NET WebApps. These techniques focus on the WebApps distinguishing features, including their multi-tier nature, hyper-linked structure, and excessive use of events. First, they extended previous WebApps models by improving existing dependence graphs and presenting a dependence graph model based on events. Second, they applied data flow testing techniques to the dependence graphs and proposed a testing technique based on event flow. Third, they proposed some coverage testing approaches. Finally, they introduced mutation testing operators for evaluating WebApp tests adequacy.

Qi et al. [14] have proposed an agent-based approach to conduct WebApps data flow testing. They conducted the data flow testing by autonomous test agents at three levels: method, object, and object cluster levels.

Liu [15] has proposed Java Server Pages (JSP) data flow testing technique, which is an adaptation of conventional ones. Liu presented a test model that captures JSP pages data flow information considering various JSP action tags and implicit objects. Based on this test model, Liu presented an approach to obtain the intra-procedural, inter-procedural, and sessional data flow test paths for revealing the JSP pages data anomalies.

Girgis et al. [8] have presented an approach to data flow testing of WebApps. They presented an approach that comprises the construction of a WebApp data flow model to aid WebApps data flow analysis. In this approach, testing is conducted in four different levels: Function, Inter-procedural, Page, and Inter-Page levels. In each level, the definition-use (def-use) pairs of the variables are obtained. Then, the all-uses criterion can be satisfied by generating test data that cover these def-use pairs.

3 Data Flow Analysis

The most important task of the data flow testing is to analyze the variables. Each variable must be tracked to get its definitions (defs) and uses. The variable is said to be defined if it is assigned a value, and is said to be used if it is referenced in a statement. Two types of use exist: The first type is c-use where the variable is used in a computation. The second type is p-use where the

variable is used in a predicate.

Data flow techniques use a program control flow graph representation to obtain the def-use pairs. A def-use pair is a def-clear path between a variable def and its use with no new definitions in the path. A test data adequacy criterion is required to define the completion of the testing process [16]. Rapps and Weyuker [17] presented a family of data flow testing criteria. This work is based on one of these criteria, which is the *all-uses* criterion that requires a def-clear path from each definition of a variable to each use of that variable to be exercised.

4 Proposed Framework

This section presents the proposed data flow testing framework of ASP.NET WebApps. In WebApps, each web page is organized into two separate files. The first file (.aspx) is the presentation file, which contains XHTML tags and other server controls. The second file (.aspx.cs) is the code-behind class, which contains the event handlers, data items, and other functions required for the class to do certain actions.

4.1 Data flow testing model

In order to capture the data flow information of the WebApp under test, a data flow model of the WebApp must be constructed. The model is organised into four types of graphs: CFG (*Control Flow Graph*), ICFG (*Inter-procedural Control Flow Graph*), PCFG (*Page Control Flow Graph*), and CCFG (*Composite Control Flow Graph*).

The CFG represents the data flow of a single function. The ICFG represents the data flow of all functions and calling statements. The ICFG merges the CFGs of the calling and called functions into a single entry, single exit CFG. The PCFG represents the data flow between the presentation file and the code-behind file of a page. The PCFG merges the CFG of the presentation file with the ICFG of the code-behind file into a single entry, single exit CFG. Finally, the CCFG represents the data flow of all the web pages in the WebApp. The CCFG merges the PCFGs of all web pages of the application into a single entry, single exit CFG.

4.2 Data flow testing approach

Data flow analysis focuses on how data are used in a program. In traditional programs, data are stored in program variables. Whereas, in WebApps, data can be stored in ASP.NET server controls and session/state variables, as well as traditional program variables. Therefore, data flow analysis techniques have to be

extended to consider the distinct ASP.NET pages data elements.

In this subsection, the characterizing issues of ASP.NET pages data flow analysis are described. To properly test ASP.NET pages, the ASP.NET implicit objects, in addition to the traditional variables of the ASP.NET pages, need to be considered. Hence, in the proposed data flow testing approach, we consider the definitions and uses of five types of data objects found in WebApps [8]:

- Traditional program variables and arrays.
- Instance variables of the code-behind class.
- Simple and complex ASP.NET server controls and their properties.
- Implicit session/state objects, such as Query-string parameters, ViewState property, Cookies, and Session property.
- Objects of built-in classes, such as SQL server classes, ADO.NET classes, and custom button controls.

In order to determine the set of paths that satisfy the all-uses criterion, it is necessary to define the defs of every variable in the program and the uses that might be affected by these defs. The def-use pairs are computed as described in [18].

In the ASPX file, the server controls are treated as global variables, which can be defined or used in the code-behind class. Therefore, to perform WebApps data flow testing, it is essential to determine first when the def and use actions can occur to different server controls in the ASP.NET pages. Then, the def-use pairs for them are computed to derive suitable test cases. Table 1 shows the def and use actions for some ASP.NET server controls. The def and use actions for the SQL Server classes, and ADO.NET classes are defined in [8]. Table 1 is taken from [8] with some modifications to address the following issues:

- Because server control properties are implicitly referenced in the presentation file during page rendering, we consider any server control property defined in the code behind file to have a corresponding use at the last line of the presentation file. Also, because any used server control property is implicitly defined in the page load event, we add a def for it in the header of the page load event.
- The def of a query string is implicitly defined in the page load event. Hence, we add a def for it in the header of the page load event of the target page.
- The defs and uses of the navigation controls, such as Hyperlink, <a> element, and their properties.
- After a data source has been defined and set for the data control on the page, we must bind the data to this data source using the Control.DataBind() method. No data is rendered to the control until the DataBind() method is explicitly called, so we consider this call to be a def for the data control.

These modifications are shown in the shaded cells in Table 1. As an example to illustrate the first issue,

Table 1: The def and use actions of some ASP.NET server controls.

ASP.NET Data Item	Examples	Action	Statement	Location
Simple server control	TextBox, Button, Label, RadioButton, CheckBox	def	Definition element: <code>< asp : ControlTypeID = ControlName ></code>	ASPX file
		use	header of an event handler for the control	Code-behind class
			any statement refers to it.	
Simple server control properties	Text	def	the header of page load method	Code-behind class
			L.H.S. of an assignment statement	
		use	R.H.S. of an assignment or Conditional statement	ASPX file
			The last line of the presentation file, where the control is displayed	
Navigation control	Hyperlink, <a> element	def	Definition element: <code>< asp : HyperlinkID = ControlName ></code>	ASPX file
		use	The beginning of the class of the target page	Code-behind class
Navigation control properties	href, NavigateUrl	def	the header of page load method	Code-behind class
			L.H.S. of an assignment statement	
		use	R.H.S. of an assignment or Conditional statement	ASPX file
			The last line of the presentation file, where the control is displayed	
List control	ListBox, DropDownList, CheckBoxList, RadioButtonList, BulletedList	def	Definition element: <code>< asp : ListTypeID = ListName ></code>	ASPX file
		use	ListName.Items.Add();	Code-behind class
			header of an event handler for the list control any statement refers to it.	
List control properties	SelectedValue, SelectedItem, SelectedIndex	def	the header of page load method	Code-behind class
			Statement that assign a value to the List property	
		use	Statement that references the List property	ASPX file
			The last line of the presentation file, where the control is displayed	
Data control	DataGrid, GridView	def	Definition element: <code>< asp : datagridID = MyGrid ></code>	ASPX file
		use	A call to DataBind method	Code-behind class
header of an event handler for the data control				
Data control properties	Columns, DataSource	def	MyGrid.Columns.Add();	Code-behind class
			L.H.S. of an assignment statement	
		use	The last line of the presentation file, where the control is displayed	ASPX file

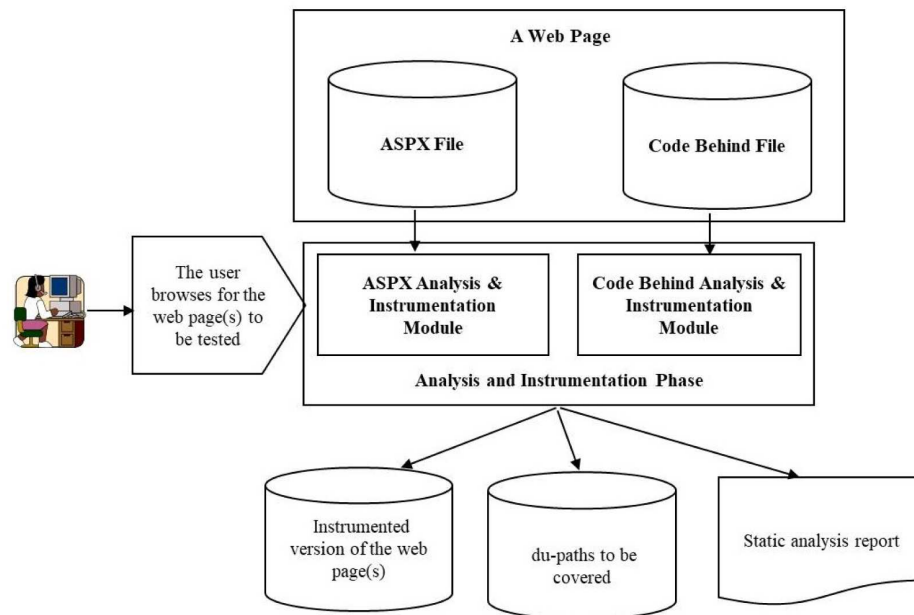


Fig. 1: The structure of the analysis and instrumentation phases of ASPDFT.

assume that a Label control, named LabelTotalText, and a Textbox control, named txtValue, are defined in the presentation file of some web page as follows:

```

<asp:Label ID="LabelTotalText" runat="server"
Text="Order Total: "></asp:Label>
<asp:TextBox ID="txtValue" runat="server">
</asp:TextBox>
    
```

Then, assume that, in the related code-behind class, the following code fragment is defined:

```

1. decimal cartTotal = int.Parse(txtValue.Text);
2. cartTotal = usersShoppingCart.GetTotal();
3. if (cartTotal > 0)
4. {
5.     LabelTotalText.Text = String.Format("0:c",
cartTotal);
6. }
    
```

In line 1, the "txtValue" text property is used and it is implicitly defined in the header of the page load, so a def action is set to it there. And in line 5, the "LabelTotalText" text property is defined and it is implicitly used in the last line of the presentation file, so a use action is set to it there.

Another example to illustrate the second issue, consider the following sample of a code behind file of some web page:

```

1. protected void Page_Load(object sender, EventArgs e)
2. {
3.     string rawId = Request.QueryString["ProductID"];
    
```

In line 3, the "Request.QueryString["ProductID"]" is a use to the QueryString object and it is implicitly defined

in the header of the page load (line 1), so a def action is set to it there.

One more example to illustrate the third issue, assume that a HyperLink control, named HomeHyperLink, is defined in the presentation file of some web page as follows:

```

<asp:HyperLink ID="HomeHyperLink" runat="server"
NavigateUrl="/WebForm1.aspx"> Home
</asp:HyperLink>
    
```

The "HomeHyperLink" is defined in this line, and it is implicitly used in the beginning of the code-behind class of the target page (WebForm1.aspx), so a use action is set to it there.

5 Tool Description

This section presents a description of an automated tool, named ASPDFT, for ASP.NET applications data flow testing. This tool goes through several analysis phases. First, it begins with an initial scan of the source code and loads it in memory, and determines the type of each statement.

Then, the tool reformats some statements in the source code to a standard format to facilitate their manipulation in the next analysis phase. The reformatting process for the presentation file involves getting rid of blank lines, combining tags written on multiple lines in a single line, and splitting multiple tags written on a single line such that each tag is on a separate line. The reformatting process for the code

behind file involves getting rid of blank lines, inserting opening and closing braces for control constructs (such as if, for, while, etc) if they are not present in the original code, placing each case of a switch block on a separate line if there are many cases in one line, and organizing the else-if block to be nested if blocks.

The next phase is the program instrumentation, in which output statements, referred to as probes, are inserted to track the line numbers of the executed path during each test run. As mentioned before, there are two files per web page, and both of these files are instrumented, but with different probes. Figure 1 illustrates the structure of the analysis and instrumentation phases of the proposed tool.

The goal of the proposed WebApps data flow testing approach is to perform data flow analysis of the WebApp to be tested, i.e. to capture its related data flow information. To do this, the ASPDFT tool goes through the following steps:

1. Static analysis.
2. Building control flow graph.
3. Constructing the def-use pairs.
4. Generating the static analysis report.
5. Dynamic analysis and generating the coverage report.

5.1 Static analysis

The source code is analyzed to get information on the statements, tags, blocks, and variables. The presentation file is analyzed to build its list of tags. Every line in the presentation file is read and parsed. Parsing a tag means to read the tag line and define its type, ID if any, the associated attributes and their values, and any related tags. Next, the code behind file is analyzed to build its list of statements. Every statement in the code behind file is read and parsed. Parsing a statement means reading the statement and determining its type, as well as giving each statement a line number, and a key code to represent its type. Then, the tool determines the blocks of the application. A block is any code structure that has an opening brace and a corresponding closing brace.

5.2 Building control flow graph

In the next step, ASPDFT builds the CFG of the WebApp. For each web page, there are two CFGs: One for the presentation file, and the other for the code behind file. These two graphs are then merged into one graph connecting the two graphs together with edges on certain nodes, as will be described later.

First, the CFG of the presentation file is constructed, where every line is considered a node. Second, the CFG of the code behind file is constructed, where every

statement is considered a node. However, the rules for constructing the CFG of the code behind class are much more complex than the presentation file [19]. Next, we construct the interprocedural CFG (ICFG) of the code behind file. An ICFG consists of the CFGs of calling and called methods. Next, a page CFG (PCFG) is constructed. A PCFG for a page consists of the CFG of the ASPX file and ICFG of the related code-behind class. Finally, a composite CFG (CCFG) is constructed to connect the entire WebApp. The CCFG is obtained by connecting the related PCFGs of the interacting Web pages together.

5.3 Constructing the def-use (du) pairs

The def-clear paths required to fulfill the all-uses criterion are called **def-use pairs** (du-paths). A list of du-paths is constructed as described in [18]. In this list, each du-path is represented by: a def-node, which contains a def of a variable; a use-node, which contains a use of that variable; and the set of nodes that must not appear in that path (nodes that contain other defs of that variable). These nodes are called **killing nodes** [18]. The construction of the def-use pairs can be done in two levels. The first level is only for one web page (presentation file and the related code-behind file). The second level is for multiple web pages. The ASPDFT tool provides both levels to the tester.

5.4 Generating the static analysis report

The static analysis report is organized into four main sections. The first section presents the list of all variables and their related information. The information includes, for each variable: type, name, scope, status, and a list of defs and uses. The variable which has defs without uses is colored in red to indicate an anomaly. The second section presents the def table records. The table is organized into four columns: Def number, Object, Variable, and Def node. The table is ordered by the def number. The third section presents the CCFG. The CCFG is organized into three columns: Edge number, Start node, and End node. The edges are ordered by the start node number. The important edges, such as those connecting functions and pages, are marked with different colors to facilitate their recognition. The fourth and final section presents the def-use table pairs. The table is organized into six columns: Variable name, Def number, Use number, Object, Function call node, and Killing nodes. In the killing node column, the value -1 means "no killing nodes" exist. In order to make the results more clear, the table is ordered by the variable name, followed by def number, and then by the use number.

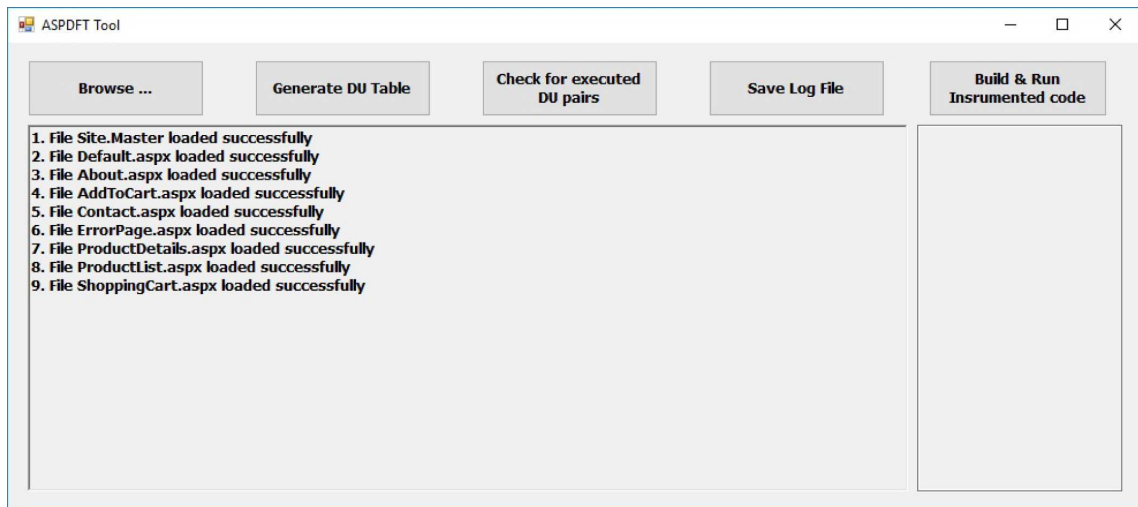


Fig. 2: The selected web pages to be tested by ASPDFT.

```

375 namespace WingtipToys
376 {
377     public partial class ErrorPage : System.Web.UI.Page
378     {
379         protected void Page_Load(object sender, EventArgs e)
380         {
381             // Create safe error messages.
382             string generalErrorMsg = "A problem has occurred on this web site.
Please try again. " + "If this error continues, please contact
support.";
383             string httpErrorMsg = "An HTTP error occurred. Page Not found. Please
try again.";
384             string unhandledErrorMsg = "The error was unhandled by application
code.";
385             // Display safe error message.
386             FriendlyErrorMsg.Text = generalErrorMsg;
387             // Determine where error was handled.
388             string errorHandler = Request.QueryString["handler"];
389             if (errorHandler == null)
390             {
391                 errorHandler = "Error Page";
392             }

```

Fig. 3: The original source code of the "ErrorPage" web page.

5.5 Dynamic analysis and generating the coverage report

After constructing the du-paths list for the WebApp under test, the tester executes the instrumented version of that WebApp with some data. After the completion of a test run, a check is made to see whether any of the constructed du-paths are covered by the traversed path. Finally, a report, showing the covered and uncovered du-paths, is produced to the tester. If a path contains a subpath that starts at the def-node of a du-path and ends at its use-node without passing through any of its killing nodes, then this path is said to cover that du-path

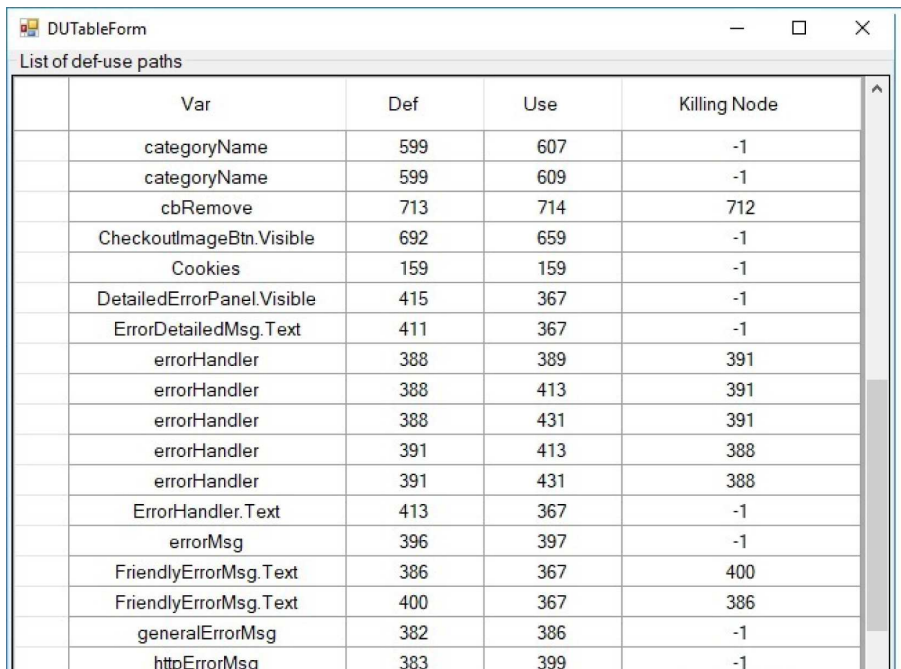
[18]. The tester then re-runs the application one or more times with different test data until all the remaining du-paths are covered. It should be noted that, sometimes a full coverage cannot be reached because of the existence of some infeasible paths that cannot be traversed by any test data. The generated coverage report is organised into three parts: First, the number of the test run is displayed along with the executed path. Then, the du-pairs fulfilled by the path are displayed. Finally, the unfulfilled du-pairs are displayed. In every successive run, the ASPDFT tool checks only the unfulfilled pairs. The tool shows only

```

public partial class ErrorPage : System.Web.UI.Page
{
    protected void Page_Load(object sender, EventArgs e)
    {
        if(!IsPostBack)
        {
            InstrumentedPath.sw.Write(377 + " ");
            InstrumentedPath.sw.Write(378 + " ");
        }
        InstrumentedPath.sw.Write(379 + " ");
        InstrumentedPath.sw.Write(380 + " ");
        // Create safe error messages.
        InstrumentedPath.sw.Write(381 + " ");
        string generalErrorMsg = "A problem has occurred on this web site.
        Please try again. " + "If this error continues, please contact
        support.";
        InstrumentedPath.sw.Write(382 + " ");
        string httpErrorMsg = "An HTTP error occurred. Page Not found. Please
        try again.";
        InstrumentedPath.sw.Write(383 + " ");
        string unhandledErrorMsg = "The error was unhandled by application
        code.";
        InstrumentedPath.sw.Write(384 + " ");
        // Display safe error message.
        InstrumentedPath.sw.Write(385 + " ");
        FriendlyErrorMsg.Text = generalErrorMsg;
    }
}

```

Fig. 4: The instrumented source code of the "ErrorPage" web page.



Var	Def	Use	Killing Node
categoryName	599	607	-1
categoryName	599	609	-1
cbRemove	713	714	712
CheckoutImageBtn.Visible	692	659	-1
Cookies	159	159	-1
DetailedErrorPanel.Visible	415	367	-1
ErrorDetailedMsg.Text	411	367	-1
errorHandler	388	389	391
errorHandler	388	413	391
errorHandler	388	431	391
errorHandler	391	413	388
errorHandler	391	431	388
ErrorHandler.Text	413	367	-1
errorMsg	396	397	-1
FriendlyErrorMsg.Text	386	367	400
FriendlyErrorMsg.Text	400	367	386
generalErrorMsg	382	386	-1
httpErrorMsg	383	399	-1

Fig. 5: Part of the Def-use pairs of the selected pages for WingtipToys webApp. (-1 means no killing nodes)

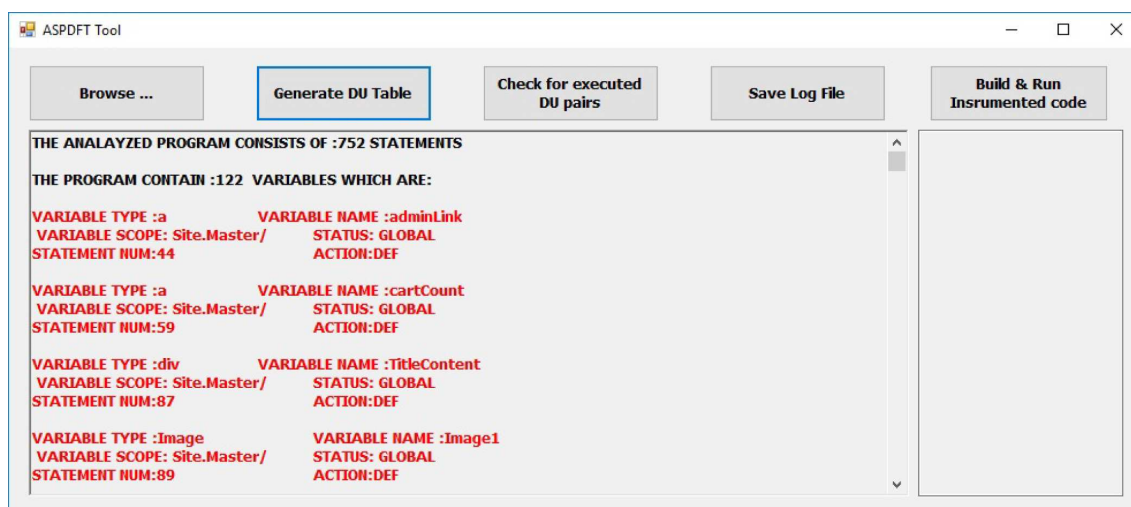


Fig. 6: Part of the static analysis report of the selected pages for WingtipToys webApp.

the newly fulfilled du-pairs and the remaining unfulfilled du-pairs.

6 Case Study

In this section, the ASPDFT tool is applied to an example WebApp, "*WingtipToys*"¹, to illustrate how it works. This application is a simplified example of a store front web site that sells items online. The "WingtipToys" application consists of a Master page and eight web pages. Five more web pages regarding checking out are in the "Checkout" subdirectory of the project. An admin web page to handle the requirements of the web site administrator is in the "Admin" subdirectory of the project. Also, 15 more web pages handling registering and managing account are in the "Account" subdirectory of the project. Additional classes can be found in the "AppStart", "Logic" and "Models" subdirectories of the project.

Figure 2 exhibits the interface of ASPDFT. The tool can be applied to the entire application or a set of selected web pages. Thus, the "Browse" button allows the tester to select a single web page or multiple web pages to be tested. After selecting the page(s), the tool analyses the page(s) and generates the corresponding instrumented version. Then, the tester can generate the du pairs of the selected page(s) by pressing the "Generate DU table" button. The tester is required to run the instrumented version with selected test data. To run the instrument version, a reconfiguration of the WebApp might be needed. A traversed path file is generated from the run in the same folder of the project. This file can be used to

automatically check for the fulfilled def-use pairs, when the tester presses the "Check for executed DU" button. All the results are shown in the main form of the tool in a rich text box. The tester can choose to save the results in a rich text document (.rtf) for later inspection, by pressing "Save log file" button. Figure 3 shows the source code of one of the selected web pages (*ErrorPage.aspx.cs*), and Figure 4 reveals its instrumented version.

The "WingtipToys" has a total of 30 web pages. For simplicity and in order to show the results, the nine main web pages are selected for analysis, as shown in Figure 2. Figure 5 shows part of the generated def-use pairs table. A part of the static report is shown in Figure 6 and part of the CFG is shown in Figure 7 with edges connecting the pages and functions are colored. The edges of the CFG are colored in "Black", the edges of the ICFG are colored in "Blue", the edges of the PCFG are colored in "Green", and the edges of the CCFG are colored in "Violet".

Then, the tester uploads the traversed path file, generated by the probes during the run of the instrumented version. Using this file, the ASPDFT automatically checks for the fulfilled def-use pairs covered by the path. A part of the coverage report is shown in Figure 8. The tester can choose to run the instrumented version with different test data to cover the unfulfilled def-use pairs, if possible.

7 Empirical Evaluation

This section presents the results of two experiments, which have been conducted to evaluate the viability of the proposed data flow testing approach, with the extensions presented in section 4.2, for ASP.NET web

¹ <https://code.msdn.microsoft.com/Getting-Started-with-221c01f5>

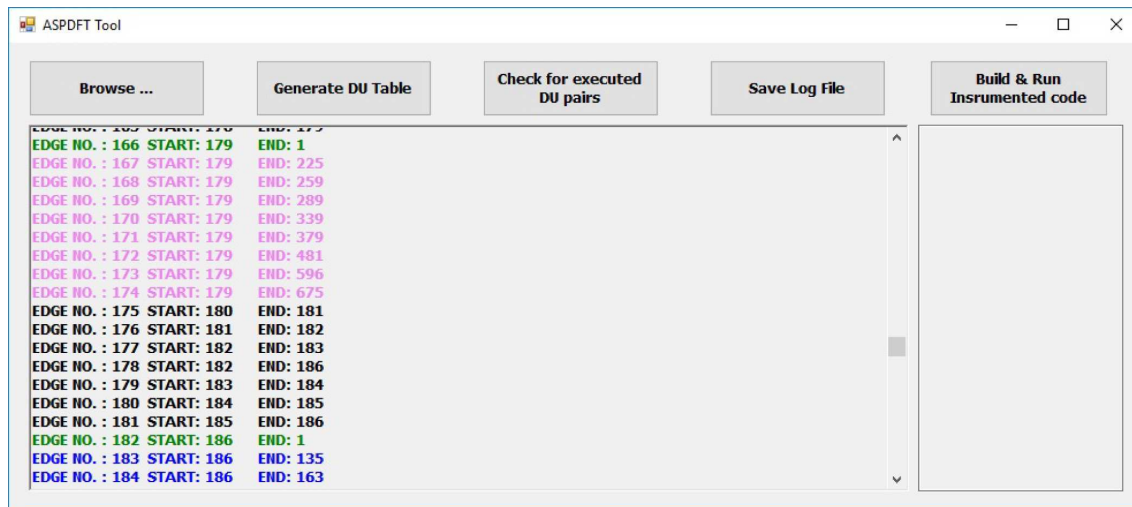


Fig. 7: Part of the CFG of the selected pages for WingtipToys webApp.

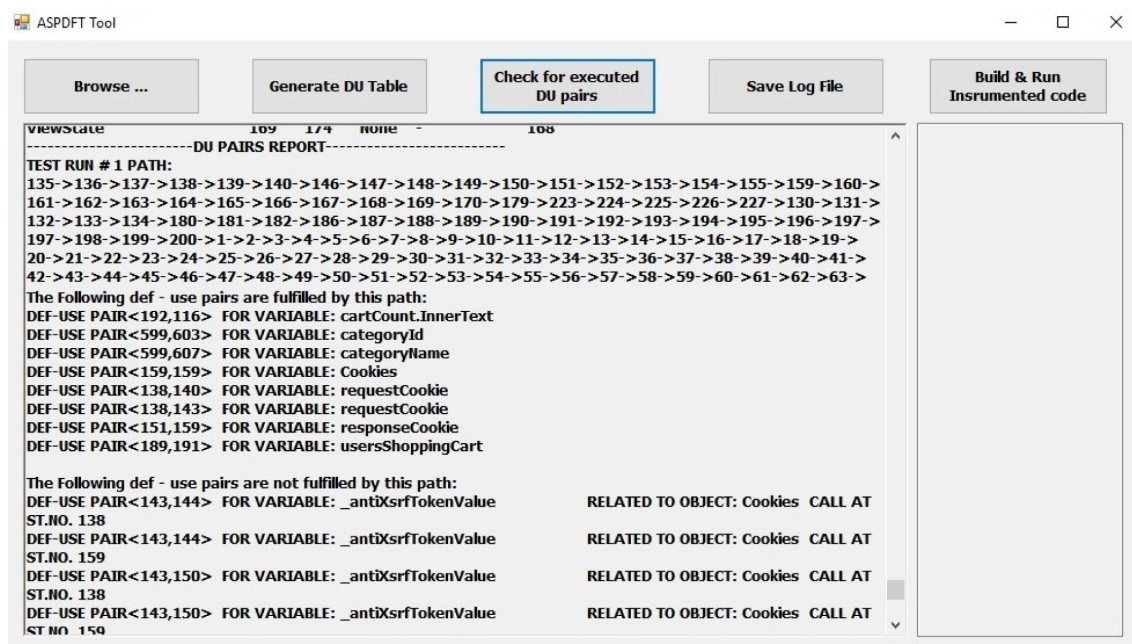


Fig. 8: Part of the dynamic coverage report generated after a test run of the selected pages for WingtipToys webApp.

applications, and the readability of the analysis and test report produced by the tool. In these experiments, the ASPDFT tool has been applied to three different ASP.NET web sites: The *Wingtip toys*, which is described in the previous section, the *to Do List*², and a *custom* site built by the authors. It consists of three web pages containing the most common ASP controls and Sql

connection to database³. In the first experiment, the number of anomalies detected; and du- pairs generated using Girgis et al. [8] approach and the proposed approach, were compared. There are three types of anomalies related to program variables: "D-D" means Def followed by Def, "D-NoU" means Def without corresponding use, and "NoD-U" means use without a corresponding def. Table 2 shows for each web site the number of dataflow anomalies, the number of correct

² <https://www.codeproject.com/Articles/1575/ASP-NET-To-Do-List-Application>

³ <https://github.com/islamelgendy/webApplication>

Table 2: A comparison between the number of anomalies detected, and du-pairs generated, by using the proposed approach and Girgis et al. approach [8].

Application	Modifications	# Dataflow Anomalies			# Correct du-pairs		# Incorrect/missing du-pairs	
		Type	Girgis et. al. [8]	Proposed approach	Girgis et. al. [8]	Proposed approach	Girgis et. al. [8]	Proposed approach
ToDo List	Add hyperlink uses	D-D	11	11	94	98	24	20
		D-NoU	20	16				
		NoD-U	5	5				
	Add Property uses & defs	D-D	11	11	94	110	24	8
		D-NoU	20	19				
		NoD-U	5	2				
	Add DataBind	D-D	11	12	94	95	24	23
		D-NoU	20	20				
		NoD-U	5	5				
	Add Query String def	D-D	11	11	94	97	24	21
		D-NoU	20	20				
		NoD-U	5	3				
	Add All	D-D	11	12	94	118	24	0
		D-NoU	20	15				
		NoD-U	5	0				
Custom	Add hyperlink uses	D-D	2	2	52	53	35	34
		D-NoU	40	39				
		NoD-U	18	18				
	Add Property uses & defs	D-D	2	2	52	83	35	4
		D-NoU	40	36				
		NoD-U	18	1				
	Add DataBind	D-D	2	3	52	53	35	34
		D-NoU	40	40				
		NoD-U	18	18				
	Add Query String def	D-D	2	2	52	54	35	33
		D-NoU	40	40				
		NoD-U	18	17				
	Add All	D-D	2	3	52	87	35	0
		D-NoU	40	35				
		NoD-U	18	0				
Wingtip	Add hyperlink uses	D-D	5	5	67	73	35	29
		D-NoU	88	82				
		NoD-U	6	6				
	Add Property uses & defs	D-D	5	5	67	92	35	10
		D-NoU	88	74				
		NoD-U	6	4				
	Add DataBind	D-D	5	6	67	68	35	34
		D-NoU	88	88				
		NoD-U	6	6				
	Add Query String def	D-D	5	5	67	70	35	32
		D-NoU	88	88				
		NoD-U	6	4				
	Add All	D-D	5	6	67	102	35	0
		D-NoU	88	68				
		NoD-U	6	2				

Table 3: A comparison between the time needed to review the reports (in *min:sec*) generated using the proposed approach and Girgis et al. approach [8].

Application	Inspection	Best time		Average time		Worst time	
		Girgis et. al. [8] (Uncolored)	Proposed approach (Colored)	Girgis et. al. [8] (Uncolored)	Proposed approach (Colored)	Girgis et. al. [8] (Uncolored)	Proposed approach (Colored)
ToDo List	Anomalies	1:51	0:30	2:31	0:36	3:40	0:43
	CFG	16:52	4:36	21:59	5:43	26:40	6:44
Custom	Anomalies	2:13	0:25	2:38	0:34	3:01	0:45
	CFG	5:34	1:52	6:40	2:06	7:55	2:23
Wingtip	Anomalies	2:57	1:02	3:36	1:33	4:10	2:02
	CFG	21:00	6:43	26:37	7:30	34:28	8:30

du-pairs, and the number of incorrect and missing du-pairs. The results have been recorded using Girgis et al. approach [8]. Then, the results were recorded after adding each modification alone. Finally, the results were recorded after adding all of the suggested modifications at once. It is clear that the number of detected data-flow anomalies and incorrect or missing du-pairs reduced, and more correct du-pairs were generated.

The purpose of the second experiment was to evaluate the readability of the report generated using the proposed approach, after applying the suggested ordering, coloring, and highlighting the key parts of the report, compared to the one generated using Girgis et al. approach [8]. In this experiment, two versions of the report were generated, one using Girgis et al. approach, and the other using the proposed approach. The reports were given to a set of 21 software engineering students who were asked to do two things: To record the consumed time for reviewing the detected anomalies, and to review correctness of the control flow graph with an emphasis on the edges of the ICFG, PCFG, and CCFG. Table 3 shows the best time, the average time, and the worst time in "*min:sec*" needed to review the results. It is clear that the suggested coloring and highlighting improve readability of the report, so the time and the effort of reviewing the results reduce.

8 Conclusion

In this paper, a tool for ASP.NET WebApps data flow testing has been proposed. The tool goes through several steps to analyse the WebApp. First, it starts by building an instrumented version of the WebApp to be used later in dynamic test runs. Second, it builds a data flow model for the WebApp under test. This model consists of four levels of control flow graphs, including CFG, ICFG, PCFG, and CCFG. Third, the def-use pairs are generated using the constructed data flow model. Finally, a static analysis report is generated. This report is organised in several sections; each section displays some of the collected information about the application under test in a highly readable format. For instance, the

important parts are highlighted to decrease their review time. For each test run of the instrumented version with manual test inputs, the tool checks and reports the fulfilment of the def-use pairs by the traversed path. Several runs can be done to cover all the def-use pairs, where each run checks only for the remaining unfulfilled def-use pairs. The paper also presented a case study to illustrate how the proposed tool works. Finally, an empirical evaluation was conducted on three ASP.NET web sites to assess the viability of the suggested data flow testing approach and the supporting tool. The results demonstrated that the number of detected data-flow anomalies and incorrect or missing du-pairs reduced, and more correct du-pairs were generated. Moreover, the suggested ordering, coloring, and highlighting improved readability of the report, so the time and effort required to review the results reduced.

Currently, we are working on evaluating the viability of the proposed tool in discovering different types of WebApp errors. Furthermore, we are working to modify the tool to automatically generate test data to cover the def-use pairs. This will automate the entire testing process, save a lot of work, and improve its accuracy.

References

- [1] T. Berners-Lee, "*The world-wide web*", Computer Networks and ISDN Systems, 25, 454-459, (1992).
- [2] N. Alshahwan, and M. Harman, "*Automated web application testing using search based software engineering*", in 26th IEEE/ACM International Conference on Automated Software Engineering ASE, 3-12, (2011).
- [3] B. Korel, "*Automated software test data generation*", IEEE Transactions on Software Engineering, 16, 870-879, (1990).
- [4] J. Edvardsson, "*A survey on automatic test data generation*", In Proceedings of the 2nd Conference on Computer Science and Engineering, 21-28, (1999).
- [5] Y.F. Li, P.K. Das, and D.L. Dowe, "*Two decades of Web application testing - A survey of recent advances*", Information Systems, 43, 20-54, (2014).

- [6] J.W. Duran and S.C. Ntafos. "An evaluation of random testing", IEEE Transactions on Software Engineering, 10, 438-444, (1984).
- [7] P. McMinn, M. Harman, K. Lakhoria, Y. Hassoun, and J. Wegener, "Input domain reduction through irrelevant variable removal and its effect on local, global, and hybrid search-based structural test data generation," IEEE Transactions on Software Engineering, 38, 453-477, (2011).
- [8] M.R. Girgis, A.I. El-Nashar, T.A. Abd El-Rahman, and M.A. Mohammed. "An ASP.NET Web applications data flow testing approach", International Journal of Computer Applications, 153, 8887, (2016).
- [9] D.R. Lakshmi, and S.S. Mallika. "A review on Web application testing and its current research directions", International Journal of Electrical and Computer Engineering, 7, 2132-2141, (2017).
- [10] C.H. Liu, D.C. Kung, P. Hsia, and C. Hsu, "Object-based data flow testing of Web applications", In Proceedings of the First Asia-Pacific Conference on Quality Software, 30-31, (2000).
- [11] F. Ricca, and P. Tonella, "Analysis and testing of Web applications", In Proceedings of the 23rd International Conference on Software Engineering, 25-34, (2001).
- [12] F. Ricca, and P. Tonella, "A 2-layer model for the white-box testing of Web applications", In Proceedings of the Sixth IEEE International Workshop on Web Site Evolution, 11-19, (2004).
- [13] N. Mansour, and M. Hourri. "Testing Web applications", Information and Software Technology, 48, 31-42, (2006).
- [14] Y. Qi, D. Kung, and E. Wong. "An agent-based data-flow testing approach for Web applications", Information and Software Technology, 48, 1159-1171, (2006).
- [15] C.H. Liu. "Data flow analysis and testing of JSP-based Web applications", Information and Software Technology, 48, 1137-1147, (2006).
- [16] P.G. Frankl, and S. Weiss. "An experimental comparison Of The effectiveness of branch testing and data flow testing", IEEE Transactions on Software Engineering, 19, 774-787, (1993).
- [17] S. Rapps, and E.J. Weyuker. "Selecting software test data using data flow information", IEEE Transactions on Software Engineering, 11, 367-375, (1985).
- [18] M.R. Girgis. "Using symbolic execution and data flow criteria to aid test data selection", Software Testing, Verification and Reliability Journal, 3, 101-112, (1993).
- [19] M.A. Mohamed, "A study of structural testing of Web applications". M.A. thesis, Faculty of Science, Minia University, Egypt, (2017).



Islam T. Elgendy received the BSc degree from Assiut University, and the M.S degree in Computer Science from the University of Assiut, Egypt in 2013. His research interests are in the areas of software engineering including Search-Based Testing and in particular automated test data generation. He is currently doing PhD in Computer Science from Assiut University, Egypt.



Moheb R. Girgis received the BSc degree from Mansoura University, Egypt, in 1974, the MSc degree from Assiut University, Egypt, in 1980, and the PhD degree from the University of Liverpool, England, in 1986. He is a professor of Computer Science, Minia University, and the director of Minia University Information Network. His research interests include software engineering, information retrieval, genetic algorithms, and networks. He is a member of the IEEE Computer Society.



Adel A. Sewisy received the BSc degree in Mathematics from Assiut University, Egypt, in 1984, the MSc degree in Mathematics from Assiut University, Egypt, in 1990, and the PhD degree in Computer Science from Assiut University, in 1997. He is a professor at Assiut University, Egypt. His research interests include artificial intelligence, image processing, arithmetic algorithms, and software engineering. He is a member of the Egyptian Computer Society and a member of the International Society of Applied Intelligence.