WHITE BOX TESTING OF WEB APPLICATIONS

NASHAT MANSOUR and MANAL HOURI
Computer Science and Mathematics Division, Lebanese American University, Mme Curie St., Beirut, Lebanon. E-mails: nmansour@lau.edu.lb, mhouri@idm.net.lb

ABSTRACT: We present new techniques for white box testing of web applications that focus on their distinctive features. We enhance previous dependence graphs for modeling of web applications and propose an event-based dependence graph model. We apply data flow testing techniques to these dependence graphs and introduce an event flow testing technique. Also, we present a few coverage testing approaches for web applications.

KEY WORDS: coverage testing, event flow testing, dependence graphs, web applications.

1. INTRODUCTION

The basic structure of web applications consists of three tiers: the client, the server and the data store. Web applications started simple and static and consisted mostly of HTML pages. Then, the integration of both HTML and other scripting languages yielded not only sophisticated web applications but also new issues that had to be addressed. The recent Microsoft's .NET platform lowered the barriers to web development [1]. In this paper, we focus on .NET web applications. ASP.NET supports event-driven programming. That is, objects on a web page can expose events that can be processed by Active Server Pages (ASP) code. Traditional web applications contain a mix of HyperText Markup Language (HTML) and scripts, making the code difficult to read, debug, and maintain. ASP.NET eliminates this problem by promoting the separation of code and content using the code-behind feature. The user interface and the user interface programming logic need not necessarily be written in a single page.

Testing is the process of revealing errors that is used to give confidence that the implementation of a program meets its specifications. Testing Techniques are usually classified as black-box and white-box. Black-box methods are specification based such as equivalence partitioning, boundary value analysis, random testing and functional analysis-based testing [2]. White-box testing methods are code based such as statement testing, branch testing, path testing, predicate testing, dataflow testing, mutation testing and domain testing [2-4]. Testing and maintaining web-based applications is both challenging and critical. It is challenging because traditional testing methods and tools are not sufficient for web-based applications, since they do not address their distinctive features. Examples of the new features of web applications are: extensive use of events, rich Graphical User Interface (GUI), and presence of server side scripting. Testing web-based applications is critical because failure may be very costly.

Research on web-based applications testing has been fairly limited. Some work has been recently proposed. Ricca and Tonnella [5] suggest a UML model of web applications and propose that all paths which satisfy a selected criterion are properly exercised. Ricca and Tonnella [6,7] investigate web application slicing and data flow testing of web applications. Di Lucca [8] employs an object-oriented model of a web application and proposes to test single units of a web application as well as integration testing. Wu and Offutt [9] define a generic analysis model that characterizes the typical behaviors of web-based applications independently of different technologies. Elbaum, Karre, and Rothermel [10] explore the notion that user session data gathered as users operate web applications can be successfully employed in the testing of those applications. In [11], data flow information of the web application using flow graphs is captured. Test cases devised for these flow graphs are based on the intra-object, inter-object, and inter-client perspectives.

In this paper, we present new techniques for testing web applications developed in the .NET environment. First, we extend previous work on modeling web applications by enhancing previous dependence graphs and proposing an event-based dependence graph model. Second, we apply data flow testing methods to the dependence graphs and propose an event flow testing technique. Also, we present a few coverage testing approaches.

This paper is organized as follows. Section 2 gives dependence graph models. Sections 3 and 4 present data flow testing, event flow testing, and coverage-based testing. Section 5 concludes the paper.

2. DEPENDENCE GRAPH MODELING FOR WEB APPLICATIONS

In this section, we present dependence graph models of .NET web applications. We extend previous control, data, and call dependence graphs with semantic dependences; we present a more elaborate call dependence graph and add an event dependence graph.

To illustrate our dependence graph modeling approach, we use a web application, "To Do List", for a simple personal agenda shown in Figure 1. This application contains only the essential elements of an agenda and was taken from [12]. It consists of two presentation ASP front ends (todolist.aspx, and edititem.aspx) and two C# code-behind classes (todolist.aspx.cs and edititem.aspx.cs) that are listed in the appendix. As depicted in Figure 1, the user views his/her unfinished tasks by priority order. Once finished with a task, the user can close it by clicking "Done". This removes the task from the open items and places it in the closed items. A user can also do several things such as "Edit a task", "Delete a task", "Add item to agenda", "Review item", etc.

Some work has been reported on slicing web applications in [6,7] based on a number of dependences. These dependences are: control, data, and call dependences, which are summarized below.

Definition: *a control dependence holds between two statements if one defines a scope which directly includes the other.* In the dependence graph, it is represented by a directed edge pointing to the dependent statement.
Definition: *a* data *dependence holds between two statements if one defines the value of a variable which is used by the other, and a definition clear path exists between the two.* In the dependence graph, it is represented by a curved directed edge.

Definition: *(1) a call dependence holds between each statement of type* call *and the server/client program or procedure invoked. (2) A* parameter-in *dependence holds between any actual parameter of a call and the respective formal parameter of the invoked program or procedure.* Both call dependences are represented by a dashed directed edge in the dependence graph.

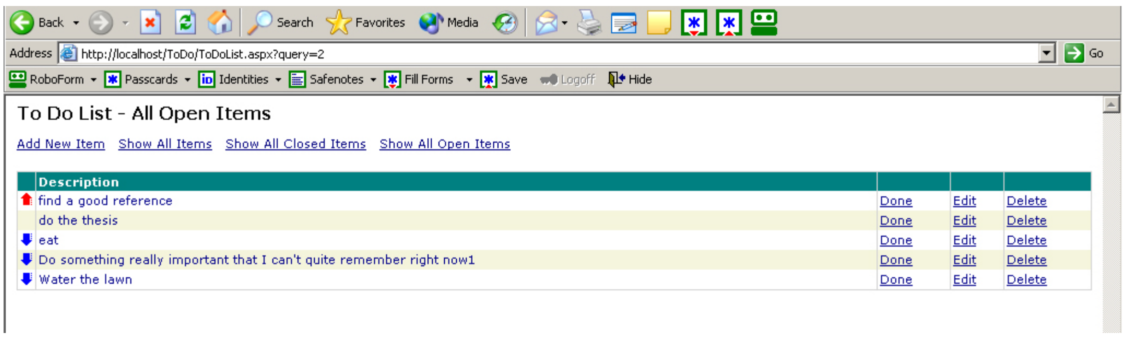
In the rest of this section, we present dependences of .NET web applications. Some of these dependences are directly applied from previous work. Others were customized to cover the unique features of ASP.NET. We start by presenting a study of all the dependences that are to be taken into consideration when slicing an ASP.NET application. Then, for every dependence, we construct its corresponding dependence graph; in this context, we introduce the Event-based Dependence Graph (EDG).

2.1. Call Dependences

In this subsection, we extend previous definitions and graph representations of call dependences. Specifically, we differentiate between three types of call dependence: internal, inheritance, and cascading; the description is intended to fit the .NET features.

2.1.1 Inheritance Call Dependence

Every ASP.NET web application has at least one presentation file. Therefore, it has at least one inheritance call dependence to the code behind class provided by the "inherits" keyword. Inheritance call dependences occur only at the root node level of any graph.
Definition: *An inheritance call dependence holds between a code behind class .aspx.cs and a presentation .aspx file if the keyword "inherits" of the .aspx file explicitly declares this inheritance.* In the dependence graph, it is represented by a dashed edge.



http://localhost/ToDo/ToDoList.aspx?query=2

To Do List - All Open Items

Add New Item   Show All Items   Show All Closed Items   Show All Open Items

	Description			
⬆	find a good reference	Done	Edit	Delete
	do the thesis	Done	Edit	Delete
⬇	eat	Done	Edit	Delete
⬇	Do something really important that I can't quite remember right now1	Done	Edit	Delete
⬇	Water the lawn	Done	Edit	Delete

Figure 1: "To Do List": A Simple Personal Agenda.

2.1.2 Internal Call Dependence

Besides the existing call dependence due to inheritance, internal call dependence may be present in ASP.NET code behind classes.
Definition: *An internal call dependence holds between a calling statement in a code behind class and a method, if both the calling statement and the method are internal to the class.* It is also represented by a dashed edge in the dependence graph.

2.1.3 Cascading Call Dependence

Besides the two previously presented types of call dependences, the unique feature of code-behind in .NET enforces another kind of dependence: "cascading call dependence". In fact, many of the main elements of a web page (i.e. data) are first defined on the presentation file (integer or string variables, datagrid, dropdown, text box, etc). Then, in the code behind class, they are used. Or these elements contribute to the definition of other data where the latter is used in the code.
Definition*: A cascading call dependence holds either (i) between a data definition in the presentation file and its use in the code behind or (ii) between a data definition in the presentation file and its use in the definition of another data in the code behind class.* It is represented by a dashed curved edge in the dependence graph.

Figure 2 illustrates control and data dependences, inheritance call, internal call, and cascading call dependences. As shown in Figure 2 for edititem.aspx, we have the definition of the variable "title" at line 118. Then in edititem.aspx.cs, title is used in the assignment statement of line 152. Therefore, we represent this cascading call dependence from one presentation file to a code behind class with a curved dashed line.

2.2 Semantic Dependences

Semantic dependences are important in .NET applications and are lacking in previous work. To model this type of dependences, we propose the addition of new notational elements to the SDG. Since ASP.NET may use both intermingled code and the code behind feature, numbering the statements and using their numbers as done in previous work does not make the representation very clear. Therefore, we introduce four new elements to the SDG: page, text, image, event and we keep the ellipse for statements of processing nature as suggested in [6]. Processing statements cover computation statements, definitions and uses of variables. It is worth mentioning that our SDG will not include all statements. Rather, it will skip the representation of statements that bring no additional information to the interactions between the different elements of the web application.

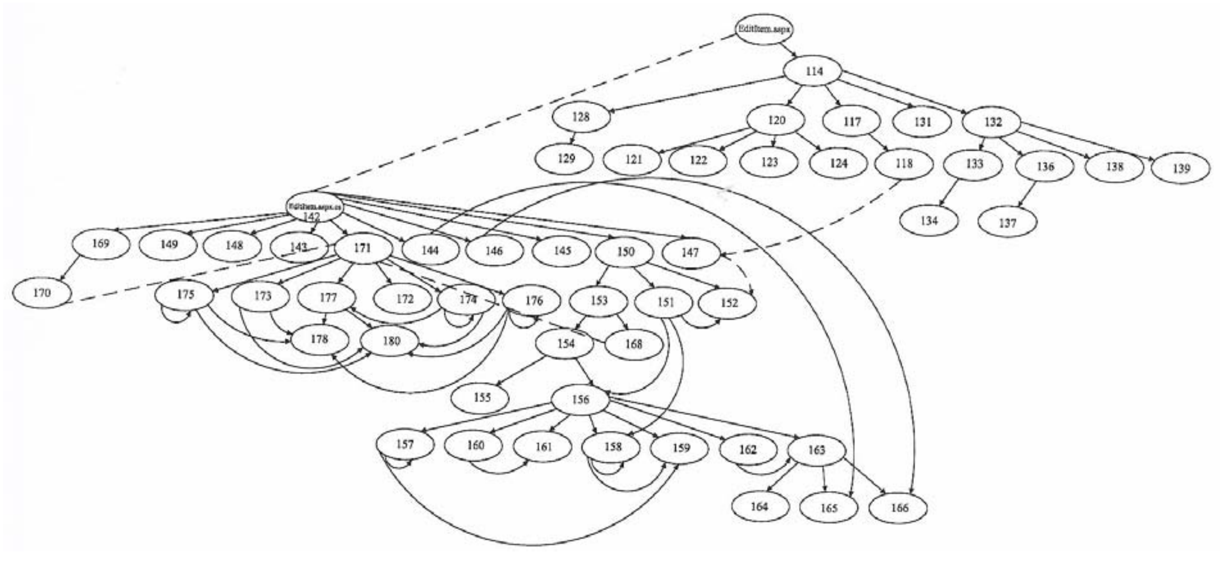


Figure 2: Control dependences (directed edges), data dependences (curved directed edges), inheritance call dependence, internal call dependences (dashed edges) and cascading call dependence (curved edge from one subgraph node to another) for a part of Edititem.aspx.cs and Edititem.aspx.

The sequential order of statements is almost meaningless for .NET web applications, since calls to code behind libraries and classes are made simple. Therefore, we suggest instead another representation style: that of defining a page and the elements holding the semantic dependences that it may contain. We use the following notation: square to represent a web page; parallelogram for a textual element (text included on a page); and a triangle for graphical element (button, text box, drop down, etc).

.NET web applications usually include a significant number of events. Therefore, it is essential to introduce a representation for events. In fact, they are the basis of the event-based dependences and the essential elements of the Event-based Dependence Graph (EDG). The representation notation of events that we propose is borrowed from the concept of Petri nets. The dynamics of a Petri net consists of a sequence of transition firing. Upon a transition firing, two things happen: (i) tokens are taken away from positions (which have arrows going from these positions) to the transition considered, and (ii) new tokens are placed on positions indicated by arrows that originate from the transition. In our model, places are web pages, transitions are events and tokens represent the interaction of the user with the event (usually this interaction occurs through the pressing of a button or a link). Places will not be represented as circles. Instead, they will be represented by any of the elements shown in Figure 3. As for tokens, we will use the solid dot to represent the "positive interaction" of the user with the event to be fired (i.e., the user did press the button), and a white dot to represent the "negative interaction" of the user with the event to be fired (i.e., the user didn't press the button). Figure 3 shows the two kinds of tokens.
Definition: *a semantic dependence holds between an informative object (graphical, textual, and processing) and a page or another informative object if the former provides information on the latter.*

2.3 Event-Based Dependences

In previous work, events are implicitly addressed through call dependences. We find it essential to add other types of dependences to satisfy a major feature of ASP.NET: events. These dependences are link, visible effect, and invisible effect dependences, which allow the construction of an Event-Based Dependence Graph (EDG). The three types of dependences are defined as follows:

(a) Link Dependence: upon clicking a button, the user may be taken to another page. This transfer is assured via the firing of an event that will fetch the requested page. A solid square arrow will point to the fetched page by the event. The part of Figure 3 that includes the elements 79, 17 and 114 illustrate this dependence.
Definition: *a link dependence holds between two pages if the first requests the second through an event (most commonly through pressing a button or a hyperlink).*

The home page of "To Do List" is "all open items" (represented by a square in Figure 3) from where we can go to another page by simply clicking the textual hyperlink "add a new item" (represented by the parallelogram in Figure 3). The "positive interaction" of the user (that of clicking the hyperlink) is indicated by the solid dot. Once the user has clicked, the event fires and the "new to do item" page is fetched.

(b) Visible Effect Dependence: sometimes when the user clicks a link or button (ex: add new item), this event directly takes him to a page where the effect of the event triggered is visible. This dependence is represented by a square dashed arrow pointing to the affected page by the event. The part of Figure 3 that includes the elements 114, 79 and 139 illustrate this dependence.
Definition: *a visible effect dependence holds between two pages if the first modifies the second through an event that will (1) implement the modification and (2) show the effect on the desired page by taking the user directly to it.*

In our application, once the user adds a new item on his agenda and presses "save changes" button, he is taken to the "all open items" page where the effect of his action is visible on the page: the added item is now on his tasks' list.

(c) Invisible Effect Dependence: sometimes when the user clicks a link or button (e.g.: delete an item), this event implements the requested action (removing the indicated record) without taking the user to the page where the effect takes place. The user has to go "manually" to the other page to see the effect taking place as supposed to. This dependence is represented by a square dotted arrow pointing to the affected page by the event. The part of Figure 3 that includes the elements 79, 82 and 63 illustrate this dependence.
Definition: *an invisible effect dependence holds between two pages if the first modifies the second through an event that will (1) implement the modification and (2) will not show the effect on the desired page by taking the user directly to it. The*

user has to go "manually" to the desired page to see the effect.

When the user of the application is done with one of the tasks of his agenda, he can close it by clicking on "done". When "done" is fired, the "open items" page is automatically updated, and the user can directly visualize the elimination of this task (this is the visible effect dependence). At the same time, the closed item rejoins the "closed items" page. The user isn't taken directly to this page upon pressing "done". This effect of the event is invisible to the user until he visits the "closed items" page intentionally (this is the invisible effect dependence).

The EDG for "To Do List" is shown in Figure 3, where irrelevant elements were removed for clarity. We note the presence of the graphical element: "row n" and the processing statement: "record p". "row n" refers to a typical row in the datagrid. Typically, it consists of a colored priority arrow, a colored background, some text, along with hyperlinks like "done", "edit" and "delete". As for "record p", it implies a typical record on the datagrid that can be either "reopened", or "deleted". We also note the presence of processing statements (ellipses) on the visible and invisible effect dependence lines in Figure 3. These ellipses inform about the action performed through each of these dependences. Example: "add record n" means a new record p is created on the datagrid of "all open items" page.

3. DATA FLOW TESTING AND EVENT FLOW TESTING FOR WEB APPLICATIONS

3.1 Data Flow Testing of .NET Applications

We use the previous experience of [11] in data flow testing of web applications and adapt it to suit .NET web applications based on the dependence graphs described in Section 2. For .NET web applications, three data flow testing levels from the previous work can be applied: function level, function cluster level and object level. In function level data flow testing, we use data dependence graphs. However, in function cluster and object level testing, we use call dependence graphs. For brevity, we include only one example of object level data flow testing of "To Do List" code.

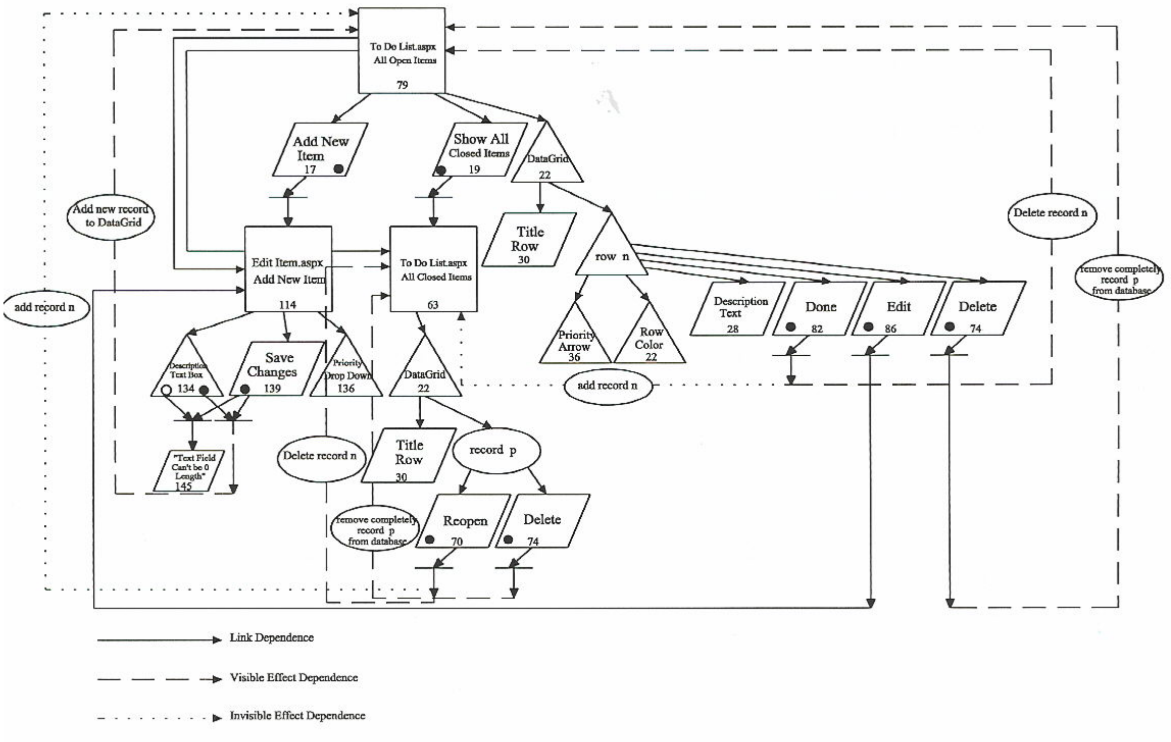


Figure 3: Event-Dependence Graph for "To Do List"

The “object” in .NET web applications corresponds to both elements of every page: the code-behind class and the presentation front. Therefore, to perform object level testing, we need to consult data the cascading call dependence graph. The code-behind class “edititem.aspx.cs” and its presentation front “edititem.aspx” serve as example for object level testing. Figure 4 illustrates this dependence between statement 118 and 152, and Table 1 identifies the variable of this object and its associated def-use chains. In this example, we have only one variable to be tested on this level. “_title” is defined in the presentation front and used in the code-behind class. Therefore, there exists a chain from the definition statement to the use statement of this variable on the object level.

3.2 Event Flow Testing of .NET Web Applications

We propose in this section event flow testing that focuses on events and their ripple effect. We identify two types of testing “levels” pertaining to the events: fetching and updating. With these two testing levels, a new type of chains is also introduced: the trigger-effect chain.

3.2.1 Fetch Level Testing

In the event flow testing, we aim to test events in an .NET web application by implementing a similar pattern to def-use pairs. For event flow testing, we introduce the trigger-effect chain that is a triplet of the following form:

<Page m, Page p, (Event i, Event ii, …, Event n)>, where Page m is where triggering of event takes place, Page p is where effect of event takes place, and (Event i, Event ii, …, Event n) are all the n events that can be triggered at the same time on page m and cause an effect on page p.

In Section 2, we described three event-based dependences. The first dependence introduced was the link dependence. This dependence concerns only fetch type events, i.e., events causing the user to move from one page to another without updating any of the pages involved. Event flow testing on the fetching level takes care of representing the Link dependence. Figure 3 illustrates the EDG for the ASP.NET web application “To Do List. In Table 2, we show the trigger-effect chains at the fetch level for this example.

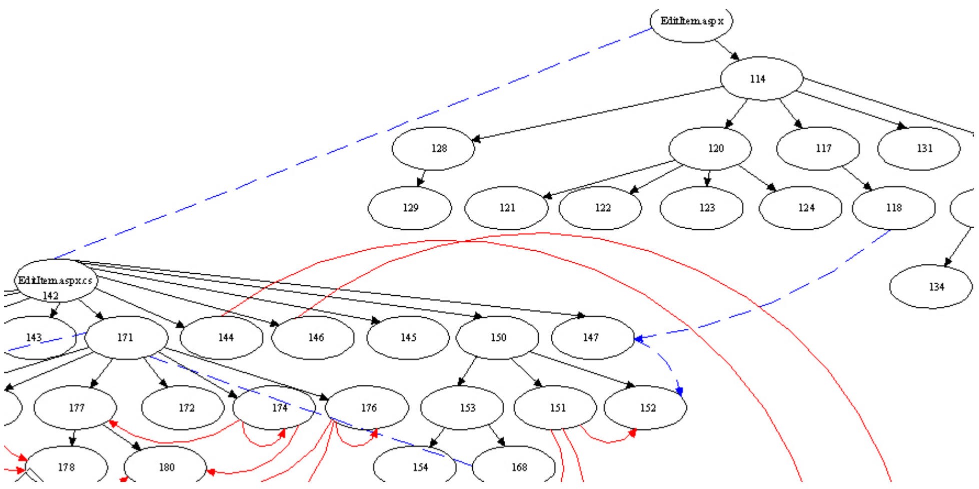


Figure 4: “Edititem.aspx.cs” and “edititem.aspx” code fragment with associated cascading call dependence graph.

Table 1: Variable of object “edititem” and its def-use chain on the object level.

Object	Variable	Test Level	Def-Use Chains
Edititem	_title	Object	<118, 152>

Table 2: Events of EDG “To Do List” and its corresponding trigger-effect chains at the fetch level.

EDG	Event(s)	Test Level	Trigger-Effect Chains
To Do List	Add New Item	Fetch	<79, 114, 17>
	Show all closed items	Fetch	<79, 63, 19>
	Edit	Fetch	<79, 114, 86>
	Not Filling Description text box, save changes	Fetch	<114, 145, (134, 139)>

Table 3: Events of EDG “To Do List” and its corresponding trigger-v-effect chains at the update level.

EDG	Event(s)	Test Level	Trigger-v-Effect Chains
To Do List	Add new record to datagrid	Update	<114, 79, (139, 134)>
	Delete record from closed items	Update	<63, 63, 70>
	Remove item permanently from database	Update	<63, 63, 74>
	Delete record from open items	Update	<79, 79, 82>
	Remove item permanently from database	Update	<79, 79, 84>

Table 4: Events of EDG “To Do List” and its corresponding trigger-i-effect chains on the update level.

EDG	Event(s)	Test Level	Trigger-i-Effect Chains
To Do List	Add record to open items	Update	<63, 79, 70>
	Add record to closed items	Update	<79, 63, 82>

On the EDG of “To Do List”, upon clicking “Add new item” (17) on the “all open items” page (79), an event is fired and the “add new item” page (114) is fetched. Also, upon clicking “save changes” without filling the “description text box” on the “edit item” page (114), two events (134 and 139) are fired where the text “Text Field can’t be 0 length” (145) is shown on the same page.

3.2.2 Update Level Testing

Data flow testing at the update level is based on the visible-effect and invisible-effect dependences. This is why it is necessary to differentiate two types of trigger-effect chains: trigger-v-effect (trigger-visible effect) and trigger-i-effect (trigger-invisible effect). Trigger-v-effect chain takes care of finding test cases for the visible effect dependence. Trigger-i-effect chain takes care of finding test cases for the invisible effect dependence. In Tables 3 and 4, we show the update level trigger-effect chains for “To Do List”. We observe from Tables 3 and 4 a complementary effect of events. For example, the event “delete record from closed items” (visible effect dependence) is the complement of the event “add record to open items”.

4. COVERAGE-BASED TESTING

Coverage testing techniques concern the process of finding areas of a program not exercised by a set of test cases, creating additional test cases to increase coverage, and determining a quantitative measure of code coverage, which is an indirect measure of quality. However, constructing a thorough set of tests that yield high coverage is often a very tedious, time-consuming task. The classical approach for code coverage includes coverage of statements, branches, paths, etc…. In this section, we present a new approach to coverage-based testing that targets web applications. In addition to the previous coverage testing types, we propose to add hyperlink coverage, input-GUI coverage and event coverage in order to test for web applications' additional features. In practice, a test criterion sets a collection of requirements to be fulfilled. These requirements are mapped to a set of entities of the web application's event dependence graph (EDG) that must be covered when tests are executed.

4.1 All-Hyperlinks Testing

To achieve hyperlinks coverage, test cases should exercise all hyperlinks, i.e. all elements pertaining to the solid square arrows on the EDG of the application. For example, in the EDG of "To Do List" shown in Figure 3, we can spot three solid square arrows. Providing test cases that exercise all these arrows will assure all-hyperlinks coverage. One all-hyperlinks coverage test case for "To Do List" is the sequence that travels from the "all Open Items" page through "Add New Item" button, to the "Add New Item" page.

4.2 All-Input-GUI Testing

To achieve input-GUI coverage, test cases should exercise all input-graphical elements, whether click buttons, or input text boxes, or drop down menus just to name a few, i.e. all elements having tokens on the EDG of the application. When consulting Figure 3, we can construct many test cases that assure all-Input-GUI coverage.

4.3 All-events Testing

To achieve events coverage, test cases should exercise all elements pertaining to the trigger and effect of every event of the application under test, i.e. all elements pertaining to the dashed and dotted square arrows on the EDG of the application. For example, with the assistance of the EDG of "To Do List", we can spot seven events.

An example of event coverage in "To Do List" could be the following: a test case must be constructed that covers the event "Add a new record to the datagrid" from the triggering point to the effect point. A sequence that covers this event can be derived from the dashed square arrow on the EDG on Figure 3. One all-events coverage test case for "To Do List" is the sequence that travels from the "all Open Items" page through "Add New Item" button, fill in the "description text box", click "save changes back to the "All Open Items" page. The other six test cases follow the same pattern as depicted in Figure 3.

5. CONCLUSION

We have presented data flow, event flow, and coverage-based testing techniques that address the features of .NET web applications and which are based on the construction of dependence graphs. These proposed techniques are useful to provide confidence about the quality of the rapidly proliferating web applications.

REFERENCES

[1] Microsoft. (2002) *Introduction to Microsoft ASP.NET*. Microsoft Official Curriculum, 2002.

[2] B. Beizer, *Software Testing Techniques*, New York, Van Nostrand Reinhold, 1990.

[3] N. Mansour and M. Salame, Data generation for path testing, *Software Quality Journal*, 12, 2004, pp. 121-136.

[4] S. Rapps and E. Weyuker, Selecting software test data using data flow information, *IEEE Trans. on Software Eng.*, April, 1985, pp. 367-375.

[5] F. Ricca and P. Tonnella, Analysis and testing of web applications, *In the Proceedings of the International Conference on Software Engineering*, Toronto, 1994, pp. 25-34.

[6] F. Ricca and P. Tonnella, Web application slicing. *In the Proceedings of the International Conference on Software Maintenance*, Italy, 2001, pp. 148-157.

[7] F. Ricca and P. Tonnella, Construction of the system dependence graph for web application slicing, *In the Proceedings of SCAM'2002 Workshop on Source Code Analysis and Manipulation*, Montreal, 2002, pp. 123-132.

[8] G. Di Lucca, et al., Testing web applications, *International Conference on Software Maintenance*, Italy, 2002, pp. 310-319.

[9] Y. Wu and J. Offutt, Modeling and testing web-based applications, GMU ISE, ISE-TR-02-08, 2002.

[10] S. Elbaum, S. Karre and G. Rothermel, Improving web application testing with user session data, *In the 25th International Conference on Software Engineering*, Portland, 2003, pp 49-53.

[11] C.H. Liu, et al., Object-based data flow testing of web applications, *International Journal of Software Engineering and Knowledge Engineering*, 11(2), 2001, pp. 157-179.

[12] J. Lyon-Smith, ASP.NET to Do List Application. 2002, www.codeproject.com.

Appendix. “To Do List” web application

Todolist .aspx:

```
<%@ Page language="c#" Codebehind="ToDoList.aspx.cs" AutoEventWireup="false"
Inherits="ToDo.ToDoListForm" %>
<HTML>
 <HEAD>
  <title>
   <%=_title%>
  </title>
  <style type="text/css">
   H1 {FONT-SIZE: 12pt; LINE-HEIGHT: 2pt; FONT-FAMILY: Verdana}
   BODY {FONT-SIZE: 8pt; FONT-FAMILY: Verdana}
   A {COLOR: navy}
   A:visited {COLOR: navy}
  </style>
</HEAD>
<body>
  <h1><%=_title%></h1>
  <form id="ToDoListForm" method="post" runat="server">
   <A href="EditItem.aspx">Add New Item</A>&nbsp;&nbsp;
   <a href="ToDoList.aspx?query=0">Show All Items</a>&nbsp;&nbsp;
   <a href="ToDoList.aspx?query=1">Show All Closed Items</a>&nbsp;&nbsp;
   <a href="ToDoList.aspx?query=2">Show All Open Items</a>&nbsp;&nbsp;
   <P>
    <asp:datagrid id="ToDoDataGrid" runat="server"
    OnItemCommand="ToDoDataGrid_Command" Width="100%"
    GridLines="Vertical" Font-Size="8pt" CellSpacing="0" CellPadding="2"
    BorderColor="lightgray" BorderWidth="1" AutoGenerateColumns="false">
     <Columns>
      <asp:BoundColumn Visible="false" DataField="ID"/>
      <asp:TemplateColumn ItemStyle-Width="12">
       <ItemTemplate><img src='<%# _priorityUrls[(int)DataBinder.Eval(Container.DataItem, "Priority") -
                                          1] %>'/></ItemTemplate>
      </asp:TemplateColumn>
      <asp:BoundColumn HeaderText="Description" DataField="Description" />
     </Columns>
     <HeaderStyle BackColor="teal" ForeColor="white" Font-Bold="true" />
     <ItemStyle BackColor="white" ForeColor="darkblue" />
     <AlternatingItemStyle BackColor="beige" ForeColor="darkblue" />
   </asp:datagrid>
```

Todolist.aspx.cs:

```
public class ToDoListForm : System.Web.UI.Page
    {
  protected System.Web.UI.WebControls.DataGrid ToDoDataGrid;
  protected string _title;
  protected string[] _priorityUrls = { "down.png", "nothing.png", "up.png" };
  private void Page_Load(object sender, System.EventArgs e)
     {
      int query = 2;
      if (IsPostBack)
         {
          query = (int)ViewState["query"];          }
     else
        {
          string queryStr = Request.Params["query"];
          if (queryStr != null)
          query = Int32.Parse(queryStr);
          ViewState["query"] = query;             }
      string connStr = ConfigurationSettings.AppSettings["ConnectionString"];
      string sql;
      string qryTitle;
      ButtonColumn bcDone;
      ButtonColumn bcEdit;
      ButtonColumn bcDelete;
      ButtonColumn bcReopen;
      BoundColumn bcOpened;
      BoundColumn bcClosed;
      switch (query)
      {
        case 0:
                 qryTitle = "All Items";
                 sql = "select * from items order by priority desc";
                   bcOpened = new BoundColumn();
                   bcOpened.HeaderText = "Opened";
                   bcOpened.DataField = "Opened";
                   ToDoDataGrid.Columns.Add(bcOpened);
                   break;
        case 1:
                   qryTitle = "All Closed Items";
                   sql = "SELECT * FROM Items WHERE Closed Is Not Null order by priority desc";
                   bcClosed = new BoundColumn();
                   bcClosed.HeaderText = "Closed";
                   bcClosed.DataField = "Closed";
                   ToDoDataGrid.Columns.Add(bcClosed);
                   bcReopen = new ButtonColumn();
                   bcReopen.Text = "Reopen";
                   bcReopen.CommandName = "ReopenToDo";
                   ToDoDataGrid.Columns.Add(bcReopen);
                   bcDelete = new ButtonColumn();
                   bcDelete.Text = "Delete";
                   bcDelete.CommandName = "DeleteToDo";
                   ToDoDataGrid.Columns.Add(bcDelete);
                   break;
        default:
        case 2:
                   qryTitle = "All Open Items";
                   sql = "SELECT * FROM Items WHERE Closed Is Null order by priority desc";
                   bcDone = new ButtonColumn();
                   bcDone.Text = "Done";
                   bcDone.CommandName = "DoneToDo";
                   ToDoDataGrid.Columns.Add(bcDone);
                   bcEdit = new ButtonColumn();
                   bcEdit.Text = "Edit";
                   bcEdit.CommandName = "EditToDo";
                   ToDoDataGrid.Columns.Add(bcEdit);
                   bcDelete = new ButtonColumn();
                   bcDelete.Text = "Delete";
                   bcDelete.CommandName = "DeleteToDo";
                   ToDoDataGrid.Columns.Add(bcDelete);
                   break;                      }
     _title = "To Do List - " + qryTitle;
     OleDbDataAdapter adapter = new OleDbDataAdapter(sql, connStr);
     DataSet ds = new DataSet();
     adapter.Fill(ds);
     ToDoDataGrid.DataSource = ds;
     ToDoDataGrid.DataBind();                   }
public void ToDoDataGrid_Command(Object sender, DataGridCommandEventArgs e)
   {
  TableCell idCell = e.Item.Cells[0];
  string idStr = idCell.Text;
  string cmdStr = ((LinkButton)e.CommandSource).CommandName;
  if (cmdStr == "EditToDo")   {
    Response.Redirect("EditItem.aspx?id=" + idStr);   }
  string connStr = ConfigurationSettings.AppSettings["ConnectionString"];
  string sql;
  switch (cmdStr)
  {
    case "DeleteToDo":
                       sql = "DELETE FROM Items WHERE ID=" + idStr;
                          break;

    case "ReopenToDo":
                          sql = "UPDATE Items SET Closed = Null WHERE ID=" + idStr;
                          break;
    default:
                          sql = "UPDATE Items SET Closed = NOW() WHERE ID=" + idStr;
                          break;
```

Edititem.aspx:

```
<%@ Page language="c#" Codebehind="EditItem.aspx.cs" AutoEventWireup="false"
Inherits="ToDo.EditItemForm" %>
<HTML>
 <HEAD>
  <title>
   <%=_title%>
  </title>
  <style type="text/css">
   H1 { FONT-SIZE: 12pt; LINE-HEIGHT: 2pt; FONT-FAMILY: Verdana }
   BODY { FONT-SIZE: 8pt; FONT-FAMILY: Verdana }
   A { COLOR: blue }
   A:visited { COLOR: blue }
  </style>
 </HEAD>
 <body>
  <script for="window" event="onload">
   window.document.forms["EditItemForm"].children["DescriptionTextBox"].focus();
  </script>
  <h1><%=_title%></h1>
  <form id="EditItemForm" method="post" runat="server">
    Description:<br>
    <asp:textbox id="DescriptionTextBox" runat="server" Font-Name="Verdana" Font-Size="8pt"
                                 Width="100%"></asp:textbox>
    <br>
    Priority:<br>
    <asp:dropdownlist id="PriorityList" Font-Name="Verdana" Font-Size="8pt"
                                 Runat="server"></asp:dropdownlist><br>
    <asp:label id="ErrorLabel" runat="server" Text="" Visible="False"
                                 ForeColor="Red"></asp:label><br>
    <asp:linkbutton id="SaveButton" onclick="SaveButton_Click" Text="Save Changes"
              Runat="server"></asp:linkbutton></form>
 </body>
</HTML>
```

Edititem.aspx.cs:

```
public class EditItemForm : System.Web.UI.Page
 {
  protected System.Web.UI.WebControls.LinkButton SaveButton;
  protected System.Web.UI.WebControls.TextBox DescriptionTextBox;
  protected System.Web.UI.WebControls.Label ErrorLabel;
  protected System.Web.UI.WebControls.DropDownList PriorityList;
  protected string _title;
  protected System.Web.UI.WebControls.LinkButton Save;
  protected static string[] _priorities = {"Low", "Medium", "High"};
  private void Page_Load(object sender, System.EventArgs e)
  {
   string idStr = Request.Params["id"];
   _title = (idStr == null ? "New" : "Edit") + " To Do List Item";
   if (!IsPostBack)
   {
        foreach (string s in _priorities)
        PriorityList.Items.Add(s);
        if (idStr != null)
        {
          string connStr = ConfigurationSettings.AppSettings["ConnectionString"];
          string queryStr = "select * from Items where id=" + idStr;
          OleDbDataAdapter adapter = new OleDbDataAdapter(queryStr, connStr);
          DataSet ds = new DataSet();
          adapter.Fill(ds);
          DataTable tbl = ds.Tables[0];
          if (tbl.Rows.Count > 0)
          {
            DataRow row = tbl.Rows[0];
            DescriptionTextBox.Text = row["Description"].ToString();
             PriorityList.SelectedIndex = (int)row["Priority"] - 1;
          }  }  }
   else  {
     OnSubmit();     }         }
  public void SaveButton_Click(object sender, System.EventArgs e)  {
    OnSubmit();      }
  protected void OnSubmit()   {
    string connStr = ConfigurationSettings.AppSettings["ConnectionString"];
    string sql;
    string idStr = Request.Params["id"];
    string desc = DescriptionTextBox.Text.Replace("'", "''");
    int priority = PriorityList.SelectedIndex + 1;
    if (idStr == null)
      sql = "INSERT INTO Items (Description, Priority) VALUES ('" + desc + "', " + priority + ")";
    else
      sql = "UPDATE Items SET Description = '" + desc + "', Priority=" + priority + " WHERE ID=" +
              idStr
```