Rt
507
c.1

Testing Web Applications

by

Manal Azzam Houri

G/T - 6992

Submitted in partial fulfillment of the requirements
for the Degree of Master of Science

Thesis Advisor: Dr. Nashat Mansour

Division of Computer Science & Mathematics
LEBANESE AMERICAN UNIVERSITY
July 2004

LEBANESE AMERICAN UNIVERSITY

GRADUATE STUDIES

We hereby approve the thesis of

Manal Azzam Houri

Candidate for the *Master of Science* degree*.

Dr. Nashat Mansour
Associate Professor
Division of Computer Science & Mathematics

Dr. Ramzi A. Haraty
Associate Professor
Division of Computer Science & Mathematics

Dr. May Abboud
Associate Professor
Division of Computer Science & Mathematics

*We also certify that written approval has been obtained for any proprietary material contained therein.

I grant to the LEBANESE AMERICAN UNIVERSITY the right to use this work, irrespective of any copyright, for the University's own purpose without cost to the University or to its students, agents and employees. I further agree that the University may reproduce and provide single copies of the work, in any format other than in or from microforms, to the public for the cost of reproduction.

Testing Web Applications

Abstract

by

Manal Azzam Houri

Classical testing techniques are not adequate for web applications, since they miss their additional features such as their multi-tier nature, hyperlink-based structure, event-driven feature, etc. Limited work has been done on testing web applications. In this thesis, we extend recent work and propose new techniques for white box testing of .NET web applications. We extend the modeling of web applications by enhancing previous dependence graphs and proposing an event-based dependence graph model. We also apply known data flow testing methods to the dependence graphs and propose an event-flow testing method. Then, we present a few coverage testing methods for web applications. Finally, we propose mutation testing for evaluating the adequacy of web application tests.

To Mom & Dad

ACKNOWLEDGMENTS

I would like to thank my advisor Dr. Nashat Mansour for his guidance throughout my M.S. studies. Thanks is also due to Dr. Ramzi Haraty and Dr. May Abboud for being on my Thesis committee.

I would like to express my sincere gratitude to the Lebanese American University whose financial support during my graduate studies made it all possible.

Finally, I would like to thank my family and my best friend for their long support.

Contents

1. Introduction **1**

2. Background **5**

2.1 ASP.NET 5

2.1.1 .NET Framework 6

2.1.2 ASP.NET Main Features 7

2.1.3 Web Forms 8

2.1.4 Server Controls 9

2.1.4.1 HTML Controls 10

2.1.4.2 Web Controls 10

2.1.4.2.1 Intrinsic Controls 11

2.1.4.2.2 Input Validation Controls 13

2.1.5 Data Binding 14

2.1.6 Code Behind in ASP.NET 17

2.2 Data Flow Testing 19

2.3 Mutation Testing 19

2.3.1 Conventional Mutation Testing 19

2.3.2 OO Mutation Testing 22

3. Dependence Graph Modeling for Web Applications **23**

3.1 "To Do List" ASP.NET Web Application 23

3.2 Previous Work in Web Application Slicing 26

3.2.1 Control Dependences 27

3.2.2 Data Dependences 28

3.2.3 Call Dependences 28

3.2.4 Semantic Dependences 30

3.3 .NET Web Applications Slicing 30

3.3.1 Control Dependences 30

3.3.2 Data Dependences 32

3.3.3 Call Dependences 33
3.3.4 Semantic Dependences 39
3.3.5 Event-based Dependences 42
3.3.6 Construction of the System Dependence Graph (SDG) 48

4. Data Flow Testing & Event Flow Testing for Web Applications 51
4.1 Previous Work in Data Flow Testing of Web Applications 51
4.1.1 Structure Model 51
4.1.1.1 Control Flow Graph (CFG) 51
4.1.1.2 Interprocedural Control Flow Graph (ICFG) 52
4.1.1.3 Object Control Flow Graph (OCFG) 52
4.1.1.4 Composite Control Flow Graph (CCFG) 53
4.1.2 Five-Level Data Flow Testing Approach 54
4.1.2.1 Function Level 54
4.1.2.2 Function Cluster Level 54
4.1.2.3 Object Level 54
4.1.2.4 Object Cluster Level 55
4.1.2.5 Application Level 55
4.2 Data Flow Testing of ASP.NET Web Applications 55
4.2.1 Function Level 55
4.2.2 Function Cluster Level 58
4.2.3 Object Level 60
4.3 Event Flow Testing of ASP.NET Web Applications 61
4.3.1 Fetch Level Testing 61
4.3.2 Update Level Testing 62
4.4 Summary Def-Use and Trigger-Effect Chains Table for "To Do List" 64
4.5 Coverage-Based Testing 66

5. Mutation Testing of ASP.NET Web Applications 69
5.1 Mutation Operators for .NET 69
5.1.1 Method Level Mutation Operators 70
5.1.2 Class Level Mutation Operators 71

5.1.3 Presentation Level Mutation Operators 73
5.1.4 Event Level Mutation Operators 75

6. Conclusions and Future Work **78**

References **80**

Appendix **83**

List of Figures

Figure 2.1: Program P before mutating statement 6 **20**

Figure 3.1: "To Do List": A Simple Personal Agenda **23**

Figure 3.2(a): The task "eat" was removed from the open items **24**

Figure 3.2(b): The task "eat" was added to the closed items **24**

Figure 3.3: The task "find a good reference" is under editing **24**

Figure 3.4: "Water the lawn" task was deleted permanently from agenda **25**

Figure 3.5: The task "Write chapter 5" with "high" priority is being added to the agenda items **25**

Figure 3.6: The task "write chapter 5" with the highest priority on top of tasks on agenda **25**

Figure 3.7: All Items page, a display for both opened (pending) and closed (already done) tasks **26**

Figure 3.8: Fragment of code and associated control dependences **27**

Figure 3.9: Fragment of PHP/HTML code and associated control dependences (straight lines) and data dependences (curve lines) **28**

Figure 3.10: Fragment of PseudoVBScript code and associated control/data dependences. Call/parameter-in dependences are depicted with dashed lines **29**

Figure 3.11: Fragment of ToDoList.aspx code (presentation end) and associated control dependences (straight arrows) **31**

Figure 3.12: Fragment of ToDoList.aspx.cs code (code behind class) and associatedcontrol dependences (straight arrows) and data dependences (curved arrows) **32**

Figure 3.13: Fragment of edititem.aspx.cs code (code behind class) where inheritance from edititem.aspx (presentation file) is explicit along with the graph of associated control dependences (straight arrows), data dependences (curved arrows) and inheritance call dependence (dashed straight line) **34**

Figure 3.14: Fragment of edititem.aspx.cs code (code behind class) with associated control dependences (straight arrows), data dependences (curved arrows) and call dependences: both inheritance and internal (dashed straight lines) **35**

Figure 3.15 (a): Fragment of Edititem.aspx.cs and edititem.aspx along with the associated inheritance call dependence, internal Call Dependence (straight dashed line) and cascading call dependence (straight dashed line from one subgraph node to another) **37**

Figure 3.15 (b): fragment todolist.aspx.cs and todolist.aspx along with the associated inheritance call dependence (straight dashed line) and cascading call dependence (two curved dashed arrows) **38**

Figure 3.16: Notational Element to represent a web page in the SDG **40**

Figure 3.17: Notational Element to represent a textual element in the SDG 40

Figure 3.18: Notational Element to represent a graphical element in the SDG 40

Figure 3.19: A Simple Petri Net 41

Figure 3.20: Representation of an Event in EDG 41

Figure 3.21: Solid dot represents a "positive interaction". White dot represents a "negative interaction" 42

Figure 3.22: Fragment of Event-Based Dependence Graph showing the link dependence between the "open items" page and the "new to do list item" page 43

Figure 3.23: Fragment of the Event-Based Dependence Graph for "To Do List" showing the visible effect dependence between the "new to do list item" page and the "all open items" page 44

Figure 3.24: Fragment of the Event-Based Dependence Graph for "To Do List" showing both the visible and the invisible effect dependences between the "all open items" page and the "all open items"(visible effect) and the "all closed items" page (invisible effect) 45

Figure 3.25: Event-Based Dependence Graph for "To Do List" 47

Figure 3.26: System Dependence Graph for "To Do List" 50

Figure 4.1: Examples of CFG and ICFG 52

Figure 4.2: Example of the OCFG of a Web page object 53

Figure 4.3: Example of the CCFG for an HTTP request between the client and server pages 53

Figure 4.4: "OnSubmit" method fragment of "Edititem.aspx.cs" with the associated data dependence graph 56

Figure 4.5: "Page_Load" method fragment of "Edititem.aspx.cs" with the associated data dependence graph 58

Figure 4.6: "Edititem.aspx.cs" code fragment with associated data dependence graph 59

Figure 4.7: "Edititem.aspx.cs" and "edititem.aspx" code fragment with associated cascading call dependence graph 60

Figure 4.8: All-Hyperlinks coverage 67

Figure 4.9: All-Events Coverage 68

List of Tables

Table 2.1: Web Controls & HTML Controls 12

Table 2.2: Validation Controls 14

Table 2.3: .NET data provider objects 16

Table 4.1: Variables of method OnSubmit and their def-use chains on the Function level 57

Table 4.2: Variables of method Page_Load and their def-use chains on the function level 58

Table 4.3: Variables of code-behind class "edititem.aspx.cs" and their def-use chains on the function cluster level 60

Table 4.4: Variable of object "edititem" and its def-use chain on the object level 61

Table 4.5: Events of EDG "To Do List" and its corresponding trigger-effect chains on the fetch level 62

Table 4.6: Events of EDG "To Do List" and its corresponding trigger-v-effect chains on the update level 63

Table 4.7: Events of EDG "To Do List" and its corresponding trigger-i-effect chains on the update level 63

Table 4.8: Summary of Data Flow and Event Flow def-use and trigger-effect chains for "To Do List" 64

Chapter 1

Introduction

The internet is becoming a heterogeneous, distributed, multi-platform, multilingual, multimedia, autonomous, cooperative wide area network computing environment (Powell, 1998). With its growth, the internet involved the creation of new problems and complications in the area of computing. New technologies are involved, more incompatibilities are questioned, and more sophisticated solutions are demanded which result obviously in more complex applications.

The web application started primarily as a typical client-server configuration that supported limited functionality and simple solutions with little flexibility or scalability. In the last few years, this situation has dramatically changed. An "N-tier" model has been widely adopted. In its simplest forms, the model consists of three tiers: the client, the server and the data store. Web applications are characterized by the use of internet-based protocols. They follow the event-driven programming model. Also, rich Graphical User Interfaces (GUI) are very common in web applications. Web applications started simple and static and consisted in their majority of HTML pages. With time, the integration of both HTML and other scripting languages yielded not only sophisticated web applications but also new complications that needed to be addressed. As for ASP for example, one had to write large chunks of code and interlace them in HTML which resulted in unreadable code in the majority of web pages. Obviously, debugging such pages is a headache for the developer as for the tester. The new .NET platform significantly lowered the barriers to web development (Introduction to Microsoft ASP.NET, 2002). In this thesis, we are concerned with this recent technology. ASP.NET applications are less complicated than applications with ASP content, as .NET offers some new features that enhanced the process of web applications development. These features will be reviewed in this thesis as well.

Testing is the process of finding programmers' errors or program faults in the behavior or in the code of a piece of software. It is used to give confidence that the implementation of a

program meets its specifications, although it cannot ensure the correctness of a program (Barbery and Strohmeier, 1994). In general, there are five levels of software testing: unit, integration, product, system, and acceptance testing. In this thesis, we are concerned with unit testing, where different testing criteria lead to a variety of testing methods. These methods are usually classified as black-box and white-box.

Black-box methods are based on the functionality of the software, where test cases are derived from the program specifications. Examples of such techniques are equivalence partitioning, boundary value analysis, random testing and functional analysis-based testing (Beizer, 1990; Duran and Ntafos, 1984; Hamlet and Taylor, 1990; Howden, 1986; Stocks and Carrington, 1993).

White-box testing methods use the control structure of design to derive test cases guaranteeing one or more of the following criteria: the exercise of all independent paths within a module at least once, the exercise of all logical decisions on both their true and false sides, the execution of all loops at their boundaries and within their operational bounds, and the exercise of internal data structures to ensure their validity. Examples of such techniques are statement testing, branch testing, path testing, predicate testing, dataflow testing, structured testing, mutation testing and domain testing (Jeng, 1984; Korel, 1992; Mansour and Salame, 2004; Marx and Frankl, 1999; McCabe, 1982; Offutt et al., 1999; Ramamoorthly et al., 1976; Rapps and Weyuker, 1985; Tai, 1993).

Testing and maintaining web-based applications is both challenging and critical. It is challenging because traditional quality models, testing methods and tools are not sufficient for web-based applications because they do not address problems the new features of web-based applications. Examples of the new features of web applications are: the exhaustive use of events, the rich Graphical User Interface (GUI), and the presence of many server side scripting.

Testing web-based applications is critical because failure may be very costly. A glitch during an unscheduled maintenance at Amazon.com in 1998 put the site offline for several hours, with an estimated cost as high as $400,000 (Wu and Offutt, 2002).

Research on web-based applications testing is still limited. Some work has been recently proposed. Ricca and Tonnella (2001) suggest a UML model of web applications. In this model, nodes represent web objects (web pages, forms, frames) and edges represent relationships and interactions among the objects (include, submit, and split). The proposed technique guarantees that all paths in the site which satisfy a selected criterion are properly exercised before delivery. Ricca and Tonnella (2002) and Ricca and Tonnella, (2001) also investigate web application slicing and data flow testing of web applications. They argue that slicing a web application is helpful in disclosing relevant information and understanding the internal system structure. To apply slicing to web applications, these authors had to define web application's specific dependences and build upon these dependences the corresponding system dependence graph (SDG) from which valuable slices can be extracted.

While Di Lucca (2002) exploits an Object-Oriented model of a web application as a test model. Based on this model, a method is proposed to test single units of a web application as well as for integration testing. In order to experiment the proposed technique, an integrated platform has been developed along with a test case producing encouraging results.

Gabrys and Dick, (2001) take an objectives-driven approach to testing web applications. The authors suggested that testing web applications should be driven by factors that are most associated with achieving business objectives (visibility as having a high hit-rate in search engines, taking care of the initial presentation or the first impression of the web site therefore focusing on those aspects that perhaps give rise to elements of test strategy which are different from traditional testing activities.

Wu and Offutt, (2002) also try to find a method for testing web applications by defining a generic analysis model that characterizes the typical behaviors of web-based applications independently of different technologies.

Elbaum, Karre, and Rothermel, (2003) explore the notion that user session data gathered as users operate web applications can be successfully employed in the testing of those applications. Results given by the authors show that user session data can produce test suites as effective overall as those produced by existing white-box techniques, but at less expense.

This paper finally proposes that the suggested technique be complemented to more formal white box testing approaches to yield satisfactory results.
Jia and Liu, (2002) propose an approach for rigorous and automatic testing of web applications using formal specifications. They have also developed a prototype tool based on the proposed approach which accepts formal specifications in XML and automatically generates test cases, execute them and validate the test results.

In (Liu, 2001), 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 thesis, we present new techniques for testing web applications 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 known data flow testing methods to the dependence graphs and propose an event-flow testing method. Also, we present a few coverage testing methods for web applications. Third, we propose mutation testing for evaluating the adequacy of web application tests. These contributions are illustrated with examples throughout the thesis, extracted mainly from a .NET web application "To Do List".

This thesis is organized as follows. Chapter 2 reviews the literature for .NET technology, data flow testing and mutation testing. In chapter 3, we model dependence graphs for .NET web applications. Chapter 4 presents data flow testing, event flow testing, and coverage-based testing for .NET web applications. Chapter 5 covers mutation testing on .NET web applications. Chapter 6 concludes the thesis.

Chapter 2
Background

We present the ASP.NET technology: its main features, its major capabilities, intricacies and differences from ASP. Then, we cover the purposes and concepts of two white-box testing techniques: data flow testing and mutation testing. Being a fault-based testing technique, mutation helps in validating test adequacy. We also include the history of mutation, its main types and operators.

2.1 ASP.NET

The Microsoft .NET platform provides all the tools and technologies that are needed to build distributed web applications. It exposes a consistent, language-independent programming model across all tiers of an application, while providing seamless interoperability with, and easy migration from, existing technologies (Introduction to Microsoft ASP.NET, 2002).

The .NET platform is composed of several core technologies:

- The Microsoft .NET Framework
- The Microsoft .NET Building Block Services
- The Microsoft .NET Enterprise Servers
- The Microsoft Visual Studio .NET
- The Microsoft Windows .NET

In the subsections of 2.1, we investigate .NET framework, list the main features of ASP.NET, explain web forms, differentiate the various server controls, explain data binding in .NET and consider code behind in ASP.NET applications.

2.1.1 .NET Framework

The .NET Framework is a set of technologies that is integral to the .NET platform. It provides the basic building blocks for developing web applications and services. The .NET framework includes the following components:

- **Common Language Runtime (CLR):**
  - CLR provides the programming interface between the .NET framework and the programming languages available for the .NET platform.
  - CLR simplifies application development.
  - CLR provides a robust and secure execution environment.
  - CLR supports multiple languages.
  - CLR simplifies application deployment and management.

- **Base Class Library**

.NET framework includes classes that encapsulate data structures, perform input/output (I/O), provide access to information about a loaded class, and provide a way to invoke security checks. It also includes classes that encapsulate exceptions and other helpful functionality such as data access, server-side user interface (UI) projections, and rich Graphical User Interface (GUI) generation. .NET framework provides both abstract base classes and class implementations derived from those base classes. Derived classes can be used as is or new classes can be derived from them.

- **Data**

ADO.NET provides improved support for the disconnected programming model and provides an XML support.

- **Web Forms and Web Services**

ASP.NET web forms provide an easy and powerful way to build dynamic user interfaces (Web forms will be tackled in detail in subsection 2.1.3). ASP.NET web services provide the building blocks for constructing distributed web-applications.

- **Windows Forms**

For applications based on windows, .NET framework provides the System.Windows.Forms namespace to create the user interface. It provides inheritance in the same client user interface library. Components can be built by using inheritance and then aggregate them by using a form designer.

2.1.2 ASP.NET Main Features

In this section we identify the main features of ASP.NET:

- ASP.NET is not just the next version of ASP. It is a totally re-architected technology for creating dynamic, web-based applications.

- ASP.NET supports event-driven programming. That is, objects on a web page can expose events that can be processed by ASP.NET code. Load, Click and Change events handled by code makes coding much simpler and much better organized.

- ASP.NET, allows developers to write cleaner code that is easier to reuse and share.

- ASP.NET boosts performance and scalability by offering access to compiled languages.

- ASP.NET provides a true language-neutral execution framework for web applications. Over 20 languages can be used to build .NET applications.

- In ASP.NET, code is compiled. When you request a page for the first time, the runtime compiles the code and the page itself, and keeps a cached copy of the compiled result. When the page is requested the second time, the cached copy is used. This results in greatly increased performance, because after this first request, the code can run from the much faster compiled version and the content on the page doesn't need to be parsed again.

- ASP.NET includes a range of useful classes and namespaces. Namespaces are used as an organizational system-a way to present program components that are exposed to other programs and applications. Namespaces contain classes. Namespaces are like class libraries and can make writing web applications easier.

- ASP.NET contains a large set of HTML controls. Almost all HTML elements on a page can be defined as ASP.NET control objects that can be controlled by scripts. ASP.NET also contains a new set of Object Oriented input controls, like programmable list boxes and validation controls. A new data grid control supports sorting, data paging, and everything expected from a dataset control.

- ASP.NET provides several server controls that simplify the task of creating pages. Server controls are tags that are understood by the server. These server controls encapsulate common tasks that range from displaying calendars and tables to validating user input.

2.1.3 Web Forms

In this part, we describe web forms, a main element of .NET technology:

- Web forms divide web applications into two pieces: the visual component and the user interface logic.

- Web forms have .aspx extension, commonly referred to as ASP.NET pages and work as containers for the text and controls to be displayed.
- The web forms framework is an object model:
    - Although web forms are created from separate components, they form a unit.
    - When the web form is compiled, ASP.NET parses the page and its code, generates a new class dynamically, and then compiles the new class.
    - The dynamically generated class is derived from the ASP.NET Page class, but it is extended with controls, code, and static HTML text in the .aspx file.
    - All intrinsic controls in an ASP.NET form are objects. Therefore, all the controls on a form have properties, methods and events.
- A web form is denoted by the runat="server" attribute. This ensures that the form is executed at the server.
- A web form contains client-side and server-side code.
- A web form can contain server controls and HTML.

2.1.4 Server Controls

By using server controls, ASP.NET has solved the "spaghetti-code" problem with classic ASP. In fact, server controls:

- Are controls that have built-in behavior. They have properties, methods, and events that can be accessed at run time from code running on the server.
- Provide client-specific HTML that is displayed on the client: this means that we don't need to create separate pages for each browser type, nor query what type of browser is being used, because the control does that for us.
- Server controls also provide a consistent object model for the controls, providing standard properties, methods and events.

There are two sets of server controls in the ASP.NET framework: HTML controls and Web controls.

2.1.4.1 HTML Controls

HTML controls offer web developers the power of the Web Forms page framework while retaining the familiarity and ease of use of HTML elements. These controls look exactly like HTML, except that they have a runat="server" attribute. Adding this attribute guarantees that the element will be treated as a server control. The HTML controls exist in the System.Web.UI.HtmlControls namespace.

2.1.4.2 Web Controls

ASP.NET uses server controls extensively to simplify the task of programming web pages. ASP.NET includes new controls, called Web controls, which have built-in functionality and are the preferred controls to use in an ASP.NET page. Web Controls:

- Include traditional form controls such as the TextBox and Button controls besides other higher-level abstractions such as the Calendar and DataGrid controls.
- They appear in the HTML markup as namespaced tags: tags with a prefix. This prefix is used to map the tag to the namespace of the run-time component.
- Like HTML controls, these tags must also contain a runat="server" attribute.
- In web controls, the namespace tag prefix maps to the System.Web.UI.WebControls namespace.

There are two sets of Web Controls in ASP.NET: intrinsic controls and input validation controls.

2.1.4.2.1 Intrinsic Controls

Intrinsic controls map to simple HTML elements which are designed to replace standard set of HTML controls.

a. Benefits of Intrinsic controls:

- **Provide a standard naming convention for similar controls.**
  One of HTML problems was the lack of a consistent naming convention for similar controls. Intrinsic controls eliminate this problem. Example:

  <input type = "radio">

  <input type = "checkbox">

  <input type = "submit">

  All of these controls are input controls; however they behave differently.

- **Provide common properties for all controls.**
  Common properties for intrinsic controls have the same name. For example, when setting the background for a control, the same attribute is always used irrespective of the control, e.g. all controls have a **Text** property that sets the text corresponding to the control. Controls also have **BackColor** property. Setting the **BackColor ="red"** property sets the background color to red for any control.

- **Create browser-specific HTML.**
  When a page is rendered for a browser, the web controls determine which browser is requesting the page and then deliver the appropriate HTML. e.g. if the requesting browser doesn't support client-side script, the controls create server-side code and require more round trips to the server to obtain the same functionality.

b. List of intrinsic controls:

Table 2.1 lists some of the intrinsic web controls and their corresponding HTML controls:

Table 2.1: Web Controls & HTML Controls.

Web Control	HTML Control
<asp: textbox>	<input type=text>
<asp: button>	<input type=submit>
<asp: imagebutton>	<input type=image>
<asp:checkbox>	<input type=checkbox>
<asp: radiobutton>	<input type=radiobutton>
<asp:listbox>	<select size= ”5”> </select>
<asp: dropdownlist>	<select> </select>
<asp: hyperlink>	
<asp: image>	<img src=”…”>
<asp: label>	
<asp: panel>	
<asp: table>	<table> </table>

c. Handling Intrinsic Control Events

Web controls support server-side events only. The number of events is very limited because all events are server-side events and cause a round trip to the server. They support click-type events.

d. Using Event Procedures

An event handler is a subroutine that executes code for a given event. Writing event handlers for Web controls is a two-step procedure. Step 1 is to set up the event handler in the control's tag. Step 2 is to write the event procedure itself.

e. Using AutoPostBack

By default, in web controls, only click events cause the form to be posted to the server. Change events are captured, but do not immediately cause a postback. Instead, they are cached by the control until the next time a post occurs. After the form is posted to the server, the associated event procedures will run. We can specify a change event to cause a form postback, and thus run the event procedure immediately, by setting the AutoPostBack property to true.

2.1.4.2.2 Input Validation Controls

Validation controls provide a variety of data validation:

- In HTML 3.2, validating data is a difficult process. You can run validation code on the client, on the server, or both. Each time a request is received, it is needed to write code that checks the input, error messages to help the user to correctly fill in the form, which was a very expensive process for users, developers and servers.

- The introduction of input validation controls in ASP.NET makes the task of validating input easier than it has been in the past.

- Validation controls are for specific types of validation, such as range checking or pattern checking, plus a **RequiredFieldValidator** control to ensure that a user doesn't skip any entry field.

- Validation controls have both uplevel and downlevel client support:

  Uplevel browsers perform validation on the client (using JavaScript and DHTML). Client-side validation enhances the validation scheme by checking user input as the user enters data. This allows errors to be detected on the client before the form is submitted, preventing unnecessary round trips to the server for validation. When no errors

are detected, a trip is made to the server, where fields are validated again.

In downlevel checking, browsers perform validation on the server.

The following table 2.2 lists the validation controls included in ASP.NET framework:

Table 2.2: Validation Controls.

Validation Control	Function
RequiredFieldValidator	Checks whether value has been entered into a control
CompareValidator	Compares an input control to a fixed value or other input control
RangeValidator	Like **CompareValidator** but can also check that the input is between 2 values
RegularExpressionValidator	Checks that the entry matches a pattern defined by a regular expression (e.g. phone numbers, postal code, etc.)
CustomValidator	Allows you to write your own code to take part in the validation framework
ValidationSummary	Displays a summary of all of the validation errors for all of the validation controls on the page

2.1.5 Data Binding

We may use data binding to fill lists with selectable items from an imported data source, like a database, an XML file, or a script. ADO.NET is a set of classes that we use to connect and manipulate data sources. ADO.NET is specifically designed for data-related connections in a disconnected environment, thereby making it the perfect choice for Internet-based web applications. ADO.NET uses XML as the format for transmitting data

to and from the database and the web application, thereby ensuring greater compatibility and flexibility. There are three namespaces that we need to import into an ASP.NET web form when using ADO.NET:

System.data
System.Data.SqlClient
System.Data.OleDb

The ADO.NET object model provides the structure for accessing data from different data sources. There are two main components of the ADO.NET object model: the DataSet and the .NET data provider:

A **DataSet** is made up of one or more DataTables, and it is designed for data access, regardless of the data source. A **DataSet** may contain data from a SQL Server 2000 database, an OLE DB source, and an XML file. A DataSet object allows storage of data, which has been collected from a data source, in the web application. The data stored in a DataSet can be manipulated without the ASP.NET web form maintaining a connection to the data source. A connection is re-established only when the data source is updated with changes.

The .NET **data provider** provides the link between the data source and the **DataSet.** Examples of objects that are provided by the .NET data providers are listed in the following table 2.3:

Table 2.3: .NET data provider objects.

.NET data provider objects	Purpose
Connection	Provides connectivity to the data source
Command	Provides access to database commands
DataReader	Provides data streaming from the source
DataAdapter	Uses the **Connection** object to provide a link between the **DataSet** and the Data provider. The **DataAdapter** object also reconciles changes that are made to the data in the **DataSet**

There are typically three stages in data access: first, the accessing of the data from a data source and displaying it on an ASP.NET web form; second, manipulating the data; and third, sending the data updates back to the database:

- **Accessing data**: in a typical scenario, a client makes a request to an ASP.NET page. The page creates the SqlConnection and SqlDataAdapter objects, populates a DataSet from the database by using the SqlDataAdapter object, and then returns the DataSet to the client by means of a list-bound control.

- **Manipulating data**: after the DataSet is populated, the client can view and manipulate the data. While the data is being viewed and manipulated there is no connection between the client and the web server, nor is there a connection between the web server and the database server. The design of the DataSet makes this disconnected environment easy to implement. Because the DataSet is stateless, it can be safely passed between the web server and the client without tying up server resources, such as database connections.

- **Updating the database**: when the user is finished viewing and modifying the data, the client passes the modified DataSet back to the ASP.NET page, which uses a DataAdapter to reconcile the changes in the returned DataSet with the original data that is in the database.

2.1.5 Code Behind in ASP.NET

ASP 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. The user interface and the user interface programming logic need not necessarily be written in a single page.

We can separate code and contents in three ways in ASP.NET:

- By using code-behind files that are pre-compiled modules written in any .NET run-time-compliant languages.
- By creating user controls-frequently used control sets and their logic-and using them as you would use controls in ASP.NET pages.
- By moving business logic into components that can run on the server and calling those components from server-side code.

Advantages of partitioning an ASP.NET page:

- Eliminates confusing pages where code and HTML are intermingled, thereby producing pages that are easier to maintain and understand.
- Members of the development team can work on separate, individually-owned parts.
- Developers can work within environments that are familiar.
- Web authors can use HTML development tools to build the visible interface part of an application.

A code-behind class file is identical to any other class file that we might create in a chosen programming language. The class declaration, the functions, sub-procedures and the class are all public. Public procedures are used to invoke code-behind pages from the user interface page. The basic structure of a Visual Basic class file looks like the following:

```
Public Class class_name
        Public variable_name As variable_type
        Public Function function_name (parameters) As return_type
        ...
        End Function
        Public Sub sub_name (parameters)
        ...
        End Sub
End Class
```

To convert the preceding class file structure into a code-behind class file, we need to perform two steps:

1. **Use the objects from the ASP.NET environment**

   Objects that are needed to use in a class file are imported from the ASP.NET environment. At the minimum, a class file needs to import the **System** and **Web libraries**:

   ```
   Imports System
   Imports System.Web
   ```

2. **Inherit from the ASP.NET Page class**

   The class must inherit from the ASP.NET Page class so that it can be integrated into the ASP.NET page in which it is used. This is done with the **Inherits** statement.

   ```
   Public Class class_name
           Inherits System.Web.UI.Page
           ...
   End Class
   ```

2.2 Data Flow Testing

Data flow testing technique is used to test individual procedures based on the flow graphs of the procedures. Tests for a variable are conducted by executing subpaths from the definition of the variable to the points where the variable is used. Data flow testing is also used in interacting procedures. Interprocedural data flow analysis techniques are used to compute def-use chains (definition-use chains) for a variable that has a definition in one procedure and use in another.

Data flow analysis mainly focuses on the value assignment of a variable (i.e., definition) and its potential value fetch (i.e., use). The use of a variable can be either a c-use (computation use) or a p-use (predicate use). A c-use occurs whenever a variable is used in a computation or output statement while a p-use occurs whenever a variable is used in a predicate statement. Test paths of a program are then selected based on the definition-use chains of variables. A def-use chain of a variable is a path from the definition to the use of the variable without any intervening redefinitions. For c-uses, the def-use chains are paths from the statement containing the definition to the statement containing the computation use. For p-uses, the def-use chains are paths from the statement containing the definition to two executional successors of the statement containing the predicate use.

2.3 Mutation Testing

Mutation is a fault based white box testing technique, it will be tackled in this section from two perspectives: conventional and object-oriented.

2.3.3 Conventional Mutation testing

Let's first look at mutation from a historical view. Actually, mutation was proposed in 1971 by Lipton in a paper as a graduate student at Carnegie Mellon University. The paper's title was Fault Diagnosis of Computer Programs. The idea was not under the spot until 1976 when Budd, DeMillo, Lipton and few others gathered at Yale University and started investigations. At one of the presentations held by Lipton, the name "mutation

testing" was first suggested. Further research efforts were exploited in this area through master theses and doctoral dissertations in the 80s (Mathur, 1994).

Consider a program P that computes the value of z. Suppose that P is altered by replacing the statement on line 6 of Figure 2.1 by a mutated statement z: =x*(y+1). The program thus obtained is known as a mutant of P, which is referred to as M.

```
0.   begin
1.   integer x,y,z
2.   read (x, y);
3.   if (x<y) then
4.       z:=x*(y+x);
5.   else
6.       z:=x*(y-1);
7.   endif;
8.   write (z);
9.   end
```

Figure 2.1: Program P before mutating statement 6.

Mutation requires the tester to find a test case that will cause P to generate an output that is different from that of the mutant. This test case is said to distinguish the mutant from P. For our example, a test case consisting of x=2, and y=1 causes P to output z=0 but the mutant to output z=4. Clearly, this test case distinguishes M from P.

A mutant is considered live if it has not been distinguished from P. There may or may not exist a test case that can distinguish a mutant. A mutant for which there exists no test case that can distinguish it from P is considered equivalent to P.

In mutation testing, a test set T consists of zero or more test cases for a program P. If for every element of T, P behaves as expected, we evaluate T with respect to the mutation adequacy criterion according to the following steps in order:

1. Generate N mutants of P.

2. Find how many of the mutants can be distinguished by T. This is done by executing each mutant against elements of T and comparing its output with the output generated by P. D will denote the number of mutants distinguished by T.
3. For each mutant that could not be distinguished by T, determine if this mutant is equivalent to P. Let E be the number of mutants equivalent to P.
4. Compute the mutation score of test set T for program P as the ratio: D/ (N-E).

A mutation score of 1 implies that T is adequate with respect to the mutation criterion. A score of less than 1 implies that T can be improved by addition of new test cases.

Next, we need to define mutation operators. A mutation operator mutates a program in one way. Mutant operators are designed to model commonly occurring simple errors in programs. These operators are considered to be of first order, because they induce exactly one syntactic change in the program being mutated. It is evident that even though a mutant operator introduces a simple change, it might result in a significant semantic difference between that of the program and its mutant.

Two main assumptions are to be present in mind when exercising mutation: the competent programmer hypothesis and the coupling effect. As for the first, it is a name given to the assumption that given a specification, a programmer writes a program that is either almost correct or contains few simple errors. This assumption forms the basis of mutation testing. The process of program development can be considered an iterative process whereby an initial version of a program is refined by making simple changes toward a correct program.
Now the coupling effect is a hypothesis that test data that reveals simple faults is sensitive enough to reveal complex faults. It is the coupling effect that explains the ability of mutation to reveal a variety or program errors by simply mutating the program in simple manner.

A question occurs to the tester is how to select test data. In fact, when attempting to distinguish a mutant, a tester must examine the mutant and reason about which test case will distinguish it. There are three conditions that must be satisfied to distinguish M from P: reachability, necessity and sufficiency.
The first condition is satisfied if there exists a test case that forces program execution to reach line l in P. Necessity is satisfied if after reaching line l, the execution of statement S

moves P to a different state from that achieved after executing the same statement mutated S_M. sufficiency is satisfied if upon termination the state of P is different from that of M.

2.3.4 OO mutation testing

Object-Oriented programming languages contain new language features which allow new kinds of faults, some of which are not modeled by traditional mutation operators.
According to previous work in mutation testing on OO programs, faults modeled by OO-specific operators are not general; therefore, to execute mutation testing with operators created in (Ma, Kwon, & Offutt, 2002), (Kim, Calarck, & McDermid, 2000), they should be selected based on the characteristics of the program to be tested.

To be able to model OO mutation operators, one has to look at the faults on all these levels: intra-method, inter-method, intra-class and inter-class.
As for intra-method level faults, they usually occur when the functionality of a method is implemented incorrectly. In this case traditional operators for procedural programs will be enough.

The inter-method level and intra-class level faults are due to interactions between pairs of methods of a single class. These types of faults are usually addressed in integration testing.
As for the last level: inter-class, it includes faults that occur due to the OO specific features like encapsulation, inheritance, polymorphism and dynamic binding. Many efforts have been invested in this direction specifically for Java.
The earlier efforts investigated in applying mutation to OO software all tend to fall under these classes of operators:

1. Information hiding
2. Inheritance
3. Polymorphism and dynamic binding
4. Overloading
5. OO language specific features
6. Common programming mistakes

Chapter 3

Dependence Graph Modeling for Web Applications

In this chapter, we present dependence graph models of .NET web applications. We employ previously proposed control, data and call dependences' graphs (Ricca and Tonnella, 2002). However, we elaborate the semantics dependences, detail the call dependence graph, and add the event dependence graph. At the end of the chapter, all these dependences are gathered in the System Dependence Graph. To illustrate this graph modeling, we use an ASP.NET application called "To Do List".

3.1 "To Do List" .NET Web Application

To illustrate our dependence graph modeling approach, we will consider the web application for a simple personal agenda as shown in Figure 3.1. This application contains only the essential elements of an agenda. This application was obtained from (Lyon-Smith, 2002).

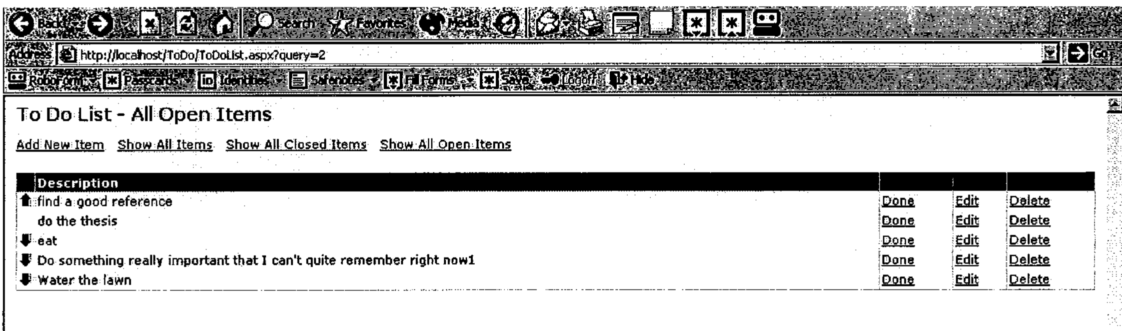


Figure 3.1: "To Do List": A Simple Personal Agenda.

"To Do List" is an ASP.NET web application. 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).

As depicted in Figure 3.1, the user views his unfinished tasks by priority order (blue arrow down for low priority, red arrow up for high priority). Once finished with a task, he can close it by clicking "Done". This removes the task from the open items and places it in the closed items as shown in Figure 3.2 (a) and 3.2 (b) respectively:

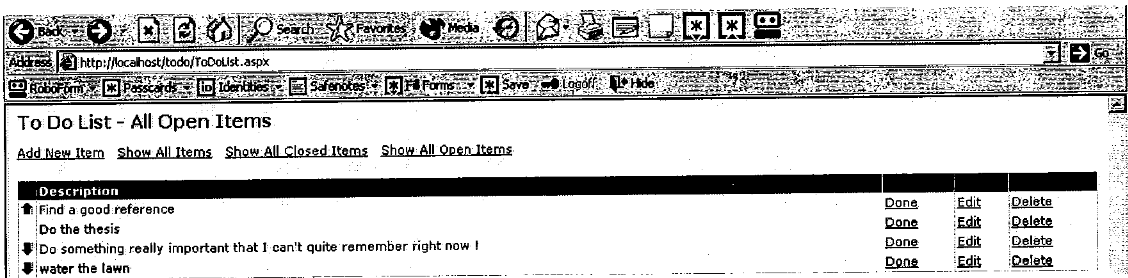


Figure 3.2(a): The task "eat" was removed from the open items.

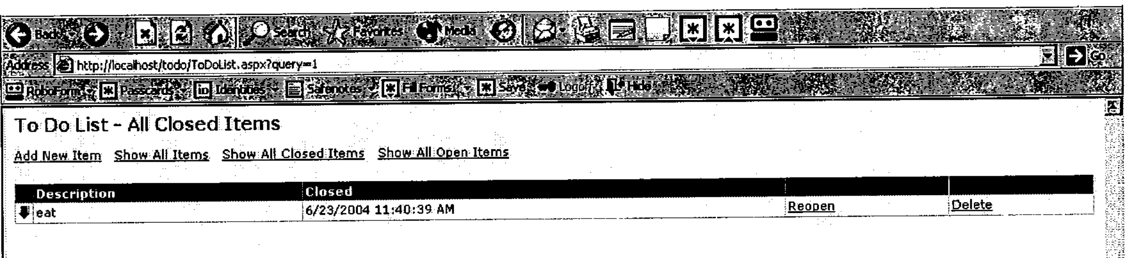


Figure 3.2(b): The task "eat" was added to the closed items.

The user can also "Edit" a task, i.e. he can modify its priority or description as depicted in Figure 3.3:

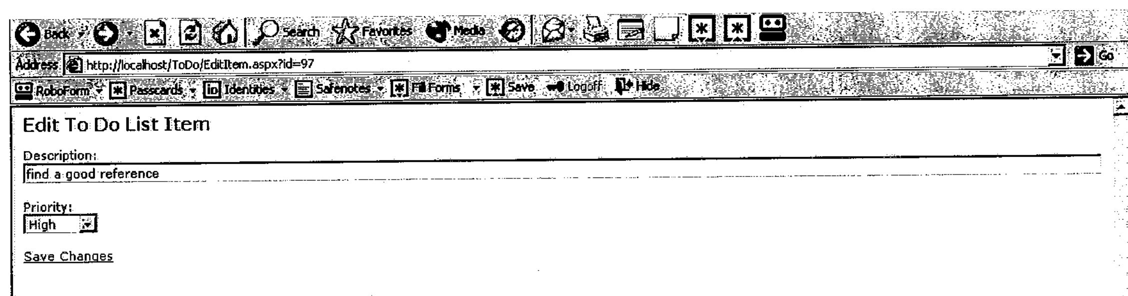


Figure 3.3: The task "find a good reference" is under editing.

The user may also delete permanently a task. Figure 3.4 shows all open items where "water the lawn" task was removed from the agenda.

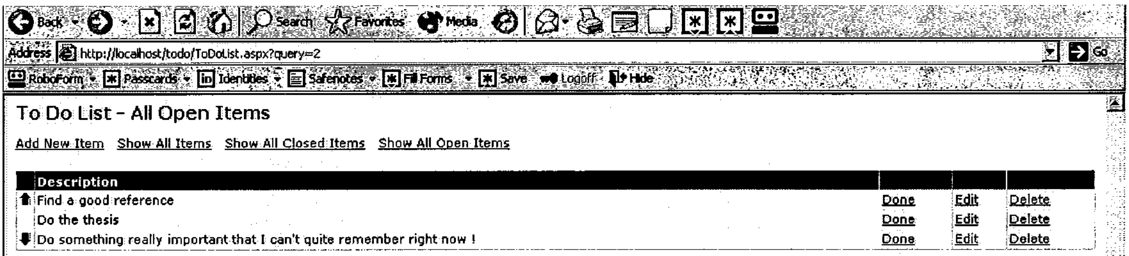


Figure 3.4: "Water the lawn" task was deleted permanently from agenda.

Besides the previously described actions, the user can add a new item to his agenda as shown in Figure 3.5:

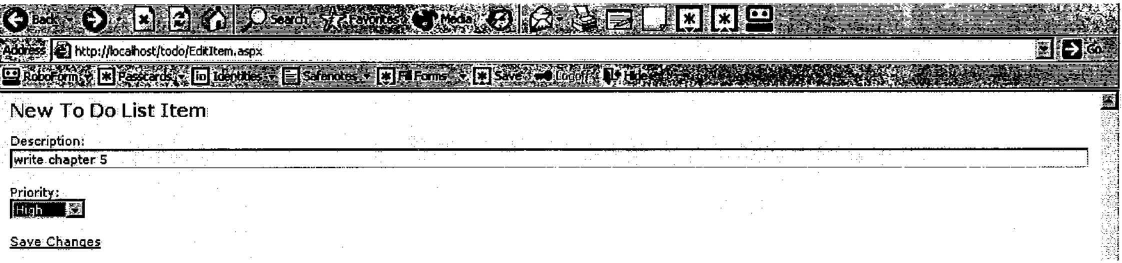


Figure 3.5: The task "Write chapter 5" with "high" priority is being added to the agenda items.

Upon saving the changes, the newly created item is added to the open items page as the highest priority task on agenda as exposed in Figure 3.6 below:

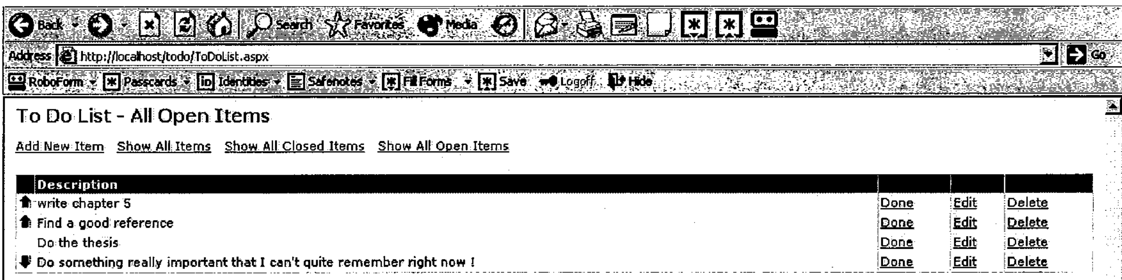


Figure 3.6: The task "write chapter 5" with the highest priority on top of tasks on agenda.

And finally, the user is able to review all his opened and closed items on one screen as shown in Figure 3.7:

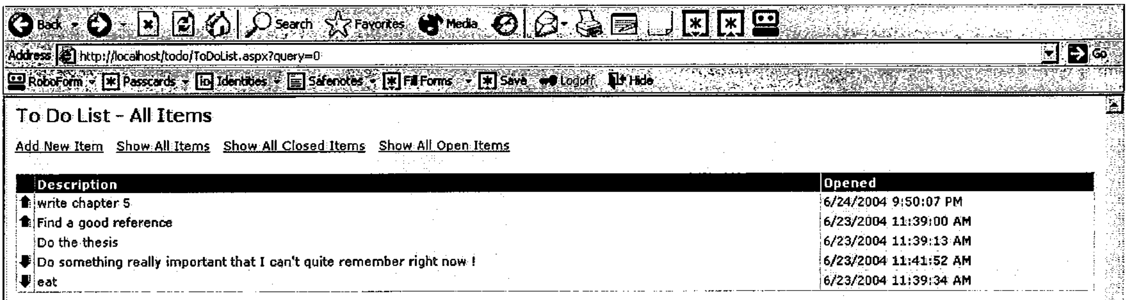


Figure 3.7: All Items page, a display for both opened (pending) and closed (already done) tasks.

Having presented thoroughly "To Do List", we will refer to it in all our subsequent work as our illustrating example.

3.2 Previous Work in Web Application Slicing

Some work has been reported on slicing web applications in (Ricca and Tonnella, 2001) and (Ricca and Tonnella, 2002). Their main contribution was the establishment of the concept of web slicing through the construction of a System Dependence Graph (SDG) for the web application under test based on a number of dependences. These dependences were extensively explained and will be reviewed in this chapter as they represent the basis for our work.

Let's review first the definition of web slicing as stated in (Ricca and Tonnella, 2001):

Definition: *a web application slice is obtained from a given set of web pages and server programs by removing HTML and server/client code statements, so that part of the behavior of the initial web application is replicated.*

Program slices are usually computed by incrementally adding consecutive sets of transitively relevant statements. Statements that directly affect the computation at another statement are

connected to the latter by a dependence relation, which can be explicitly represented in the SDG (Ricca, and Tonnella, 2002).

Having defined a web slice and how to compute it, we are ready now to review the dependences that make the base for the SDG.

3.2.1 Control Dependences

Control dependences for traditional programs connect the predicate tested at a conditional or loop statement to the instructions the execution of which directly depends on the true value of the predicate. The same holds for web applications. Besides, web applications are characterized by an additional kind of control dependence found in the HTML code. In fact, HTML statements can't be interpreted correctly unless all enclosed tags are available. Therefore, a control dependence holds between a tag and all directly enclosed statements.

Definition: *a control dependence holds between two statements if the former defines a scope which directly includes the latter.*

Example:

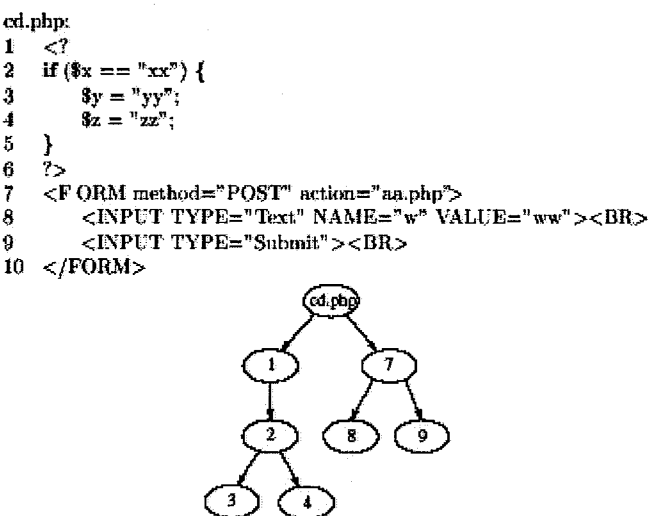


Figure 3.8: Fragment of code and associated control dependences.

3.2.2 Data Dependences

Data dependences for traditional programs hold when a statement defines the value of a variable, which is used at statement after being propagated to it along a definition clear path (i.e., a path containing no redefinition of the given variable). In addition to such a situation, which still can be found in server/client code, data flows may occur in a Web application from server/client side statements to the HTML code.

Definition: *a* data dependence *holds between two statements if the former defines the value of a variable which is used by the latter, and a definition clear path exists between the two.*

Example:

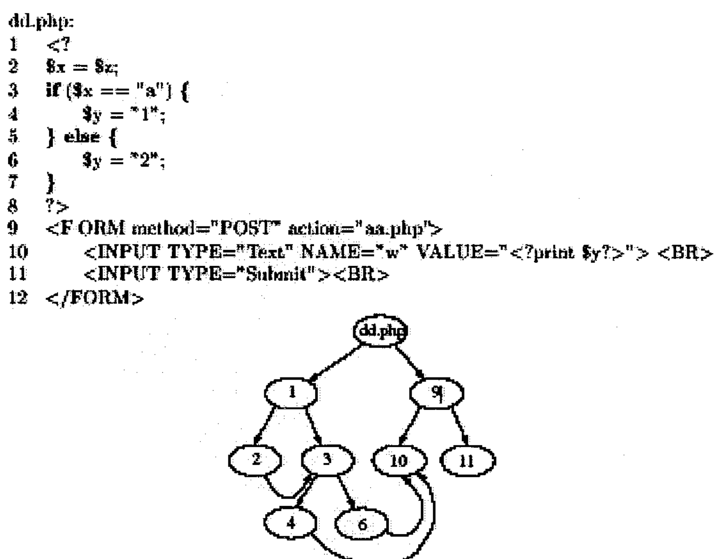


Figure 3.9: Fragment of PHP/HTML code and associated control dependences (straight lines) and data dependences (curve lines).

3.2.3 Call Dependences

Interprocedural data dependences and slices for traditional software are made quite difficult to compute due to the calling context problem. This is associated to the additional constraints that must be applied to the interprocedural data flows, when a procedure is

invoked. Since a procedure may be invoked at more than one statement, it will receive several data flows from all calling instructions. Such data flows are transformed inside the called procedure and then must be returned back to the respective call sites. The problem of keeping the data flow associated to the different call sites separated is the calling context problem.

In web applications, the calling context problem is completely absent. In fact, by no means an invoked procedure can return some value to the calling page. As a consequence, data flows can only be propagated inside an invoked server side script and never back from it. This property of web applications highly simplifies interprocedural dependences computation and slicing.

Definitions: *(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.*

Example:

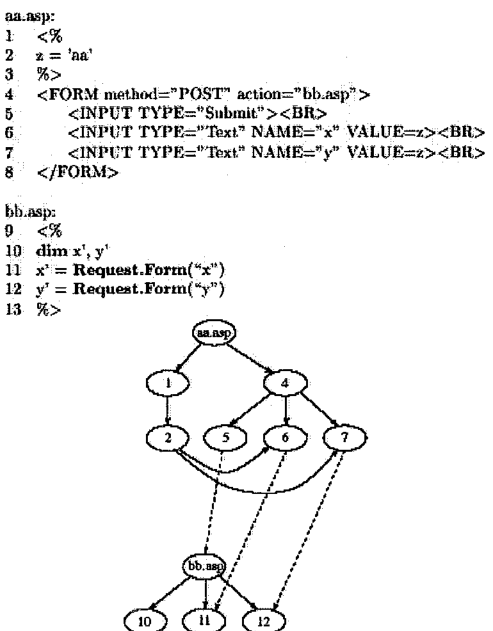


Figure 3.10: Fragment of PseudoVBScript code and associated control/data dependences. Call/parameter-in dependences are depicted with dashed lines.

3.2.4 Semantic Dependences

If the information displayed in a web page is considered part of the behavior of a web application, then a slice should also include all text, images and other objects that provide relevant information on the ongoing computation.

Definition: *a semantic dependence holds between an informative object and a statement if the former provides information for the latter.*

3.3 .NET Web Applications Slicing

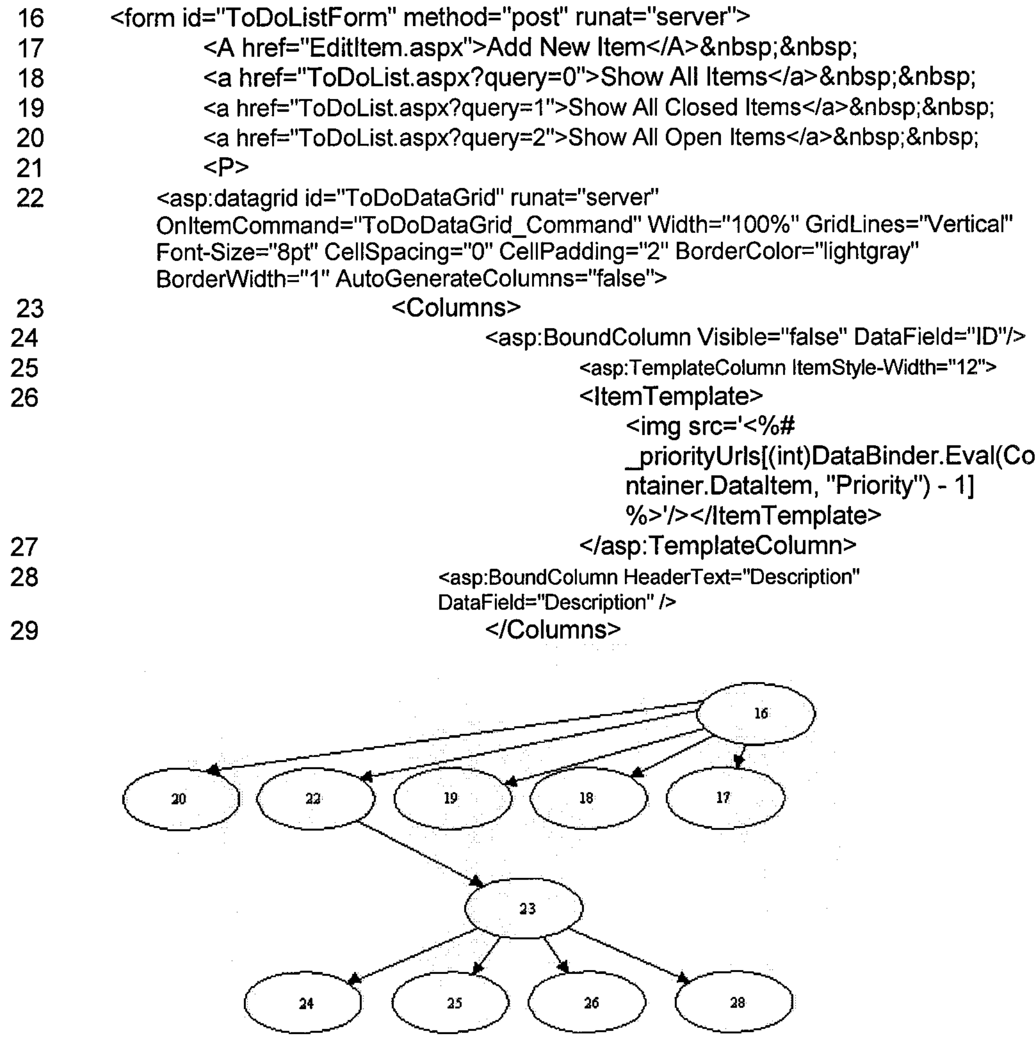
In this section, we present dependences of .NET web applications. Some of these dependences are directly applied from previous work. Others were customized to cover the unique features of ASP.NET. We start by presenting a detailed study of all the dependences that are to be taken into consideration when 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). And finally, we combine all these graphs to form the SDG for "To Do List".

As stated earlier in chapter 2, .NET can be regarded as an event-driven environment. Having this major property, events occupy a large chunk of the life of ASP.NET web applications and this chunk can't be disregarded at all. In fact, events in this case may be the source for additional errors in the application.
Consequently, the dependences provided in 3.2 are to be customized to suit .NET web applications better. Therefore, we adapt control and data dependences of (Ricca and Tonnella, 2001), enhance the call dependences they proposed to fit .NET applications, and finally propose event dependences, in order to cover the original aspects of .NET.

3.3.1 Control Dependences

Obviously, .NET web applications like any other web application contain HTML tags (that should be enclosed properly). Therefore, as far as control dependences are concerned,

we will adapt the same definitions provided by (Ricca and Tonnella, 2001) and thoroughly presented in the previous section. In Figure 3.11, we provide a portion of the presentation file of "To Do List" (todolist.aspx) along with the corresponding control dependences.



```
16   <form id="ToDoListForm" method="post" runat="server">
17        <A href="EditItem.aspx">Add New Item</A>  
18        <a href="ToDoList.aspx?query=0">Show All Items</a>  
19        <a href="ToDoList.aspx?query=1">Show All Closed Items</a>  
20        <a href="ToDoList.aspx?query=2">Show All Open Items</a>  
21        <P>
22     <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">
23                 <Columns>
24                        <asp:BoundColumn Visible="false" DataField="ID"/>
25                               <asp:TemplateColumn ItemStyle-Width="12">
26                               <ItemTemplate>
                                     <img src='<%#
                                     _priorityUrls[(int)DataBinder.Eval(Co
                                     ntainer.DataItem, "Priority") - 1]
                                     %>'/></ItemTemplate>
27                               </asp:TemplateColumn>
28                  <asp:BoundColumn HeaderText="Description"
                    DataField="Description" />
29                        </Columns>
```


Figure 3.11: Fragment of ToDoList.aspx code (presentation end) and associated control dependences (straight arrows).

Figure 3.11 contains a fragment of aspx code where the form todolistform is initialized through a series of parameters. Lines 16 through 29 are run at server side, since statement 16 encloses all the other statements. The server execution of this fragment consists of creating a

datagrid and hyperlinks to other pages. Properties of the datagrid are enclosed within the tag of datagrid at line 22 and are all executed on the server side.

Control dependences for this fragment are shown at the bottom of Figure 3.11. When a statement is inside a scope (e.g., assignment of a **hyperlink** at line 17), the associated node is connected as a child to the current scope node (e.g., the **Form** at line 16).

3.3.2 Data Dependences

As stated by (Ricca & Tonnella, 2001), data dependences for web applications have the same purpose as in traditional programs. From this perspective, .NET web applications do not differ from other applications concerning data dependences and therefore, we will adapt the same definition given previously for web applications in general.

Figure 3.12 contains an example of data dependences for a string and an integer. An integer variable of value 2 is defined at line 38 and subsequently used at lines 40 and 44. The same thing applied for the string variable defined at line 47 and used later at line 58.

```
38        int query = 2;
39        if (IsPostBack)
          {
40            query = (int)ViewState["query"];
          }
41        else
          {
42            string queryStr = Request.Params["query"];
43            if (queryStr != null)
44                query = Int32.Parse(queryStr);
```

```
47        string sql;
55        switch (query)
56            case 0:
58                sql = "select * from items order by priority desc";
```

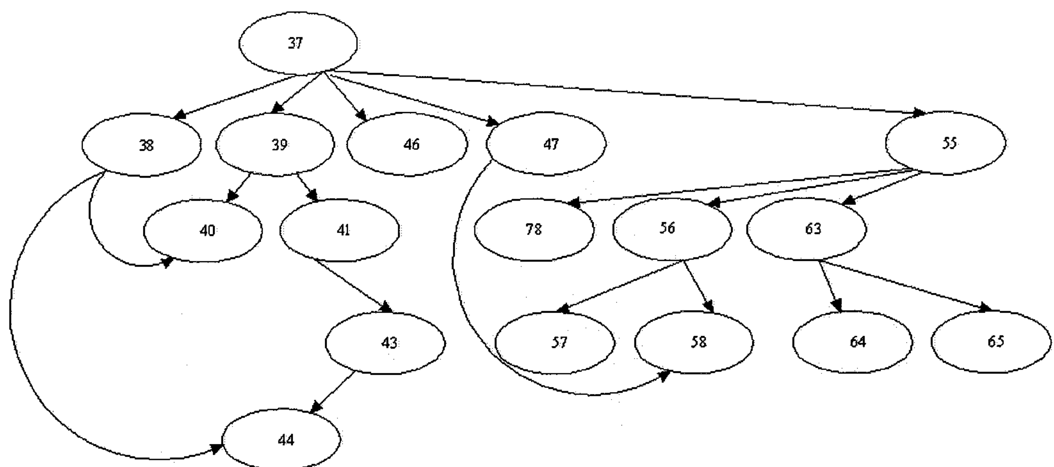


Figure 3.12: Fragment of ToDoList.aspx.cs code (code behind class) and associated control dependences (straight arrows) and data dependences (curved arrows).

3.3.3 Call Dependences

The definition provided for call dependences by (Ricca & Tonnella, 2001) is rather ambiguous. In fact, the example provided with the definition shows some confusing representations that we see worth addressing and enhancing.

In referral to the example provided in Figure 3.10, we can retrace three dashed lines, i.e. three call dependences emanating from the HTML segment to a script "bb.asp". What attracted our attention was the fact that the authors used the same representation (the straight dashed line) for –what we think are- three different dependences worth to be differentiated clearly.

In fact, pressing a button (**submit** at statement 5) and passing a parameter to the script (statements 6 and 7), both have the same representation, where in fact, they don't carry the same implication. In the following text, we will differentiate these subtypes' dependences and accommodate the previous definition of call dependence to fit .NET specific features.

Call dependences for .NET web applications are varied. We can have inheritance call dependence as well as internal call dependence. In the next paragraph, we will differentiate

these dependences through examples. We will also present the cascading call dependence which combines both call and data dependences.

a. 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.*

An example of inheritance call dependence is depicted in Figure 3.13. "To Do List" consists of two presentation files. Therefore, this application has two inheritance call dependences to two code behind classes.

<%@ Page language="c#" Codebehind="EditItem.aspx.cs"
AutoEventWireup="false" **Inherits="ToDo.EditItemForm"** %>

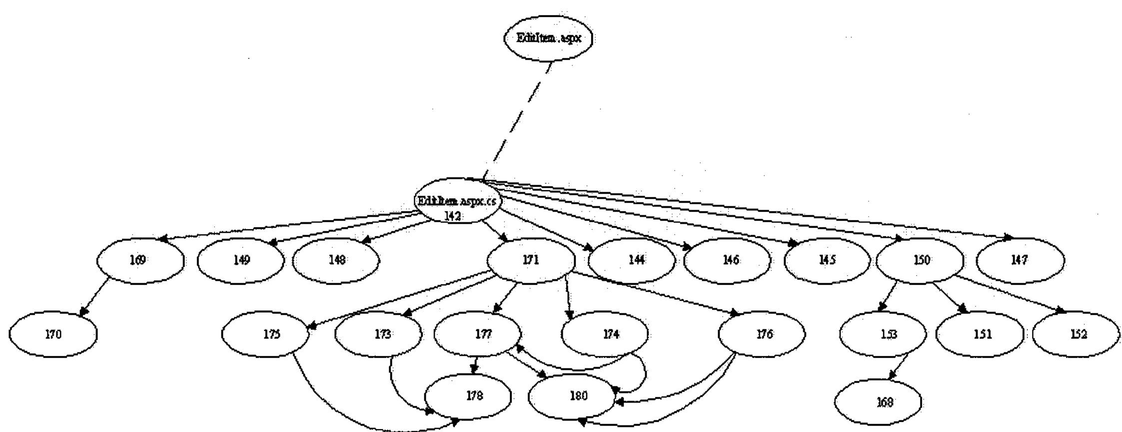


Figure 3.13: Fragment of edititem.aspx.cs code (code behind class) where inheritance from edititem.aspx (presentation file) is explicit along with the graph of associated control dependences (straight arrows), data dependences (curved arrows) and inheritance call dependence (dashed straight line).

b. Internal Call Dependence

Besides the existing call dependence due to inheritance, **internal call dependence** may be present in ASP.NET code behind classes as depicted in Figure 3.14.

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.*

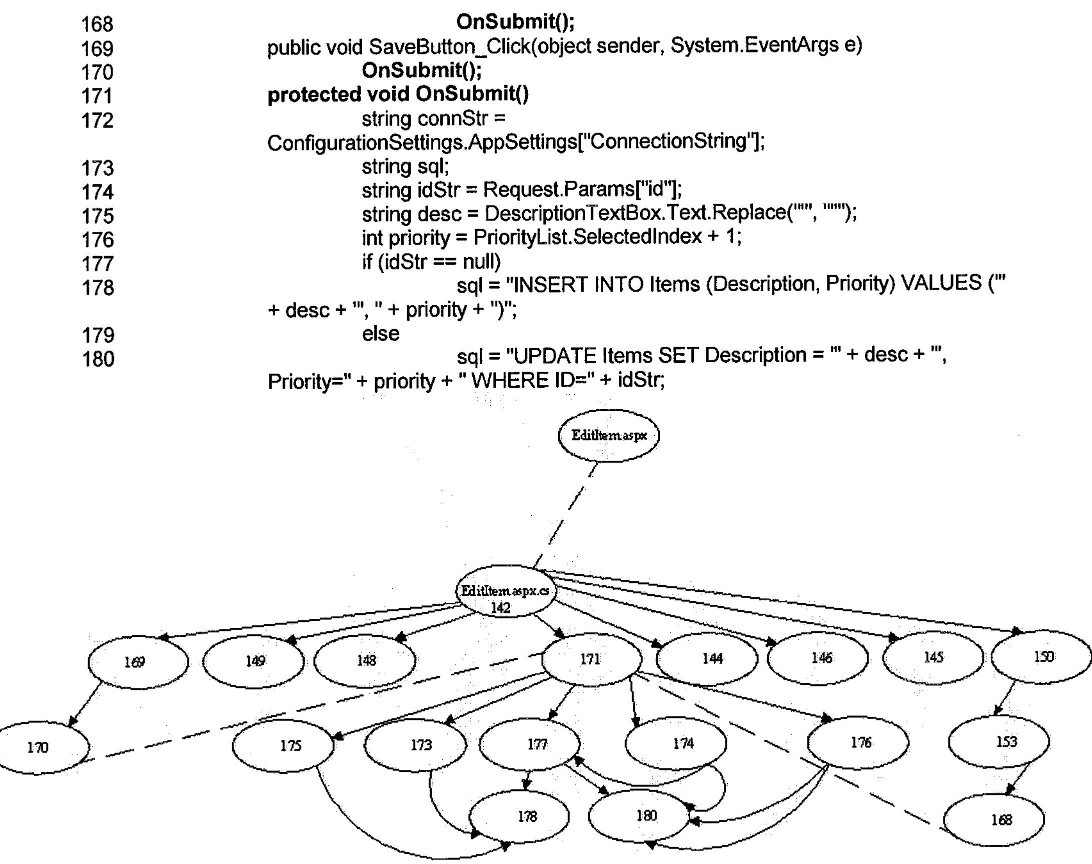


Figure 3.14: Fragment of edititem.aspx.cs code (code behind class) with associated control dependences (straight arrows), data dependences (curved arrows) and call dependences: both inheritance and internal (dashed straight lines).

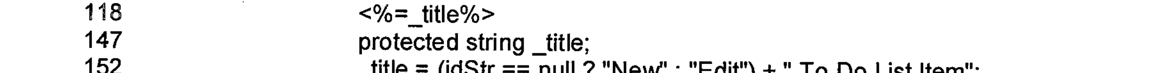
Line 168 is a call to the internal method OnSubmit() which is included in this class from lines 171 to 180. Another internal call to the same method occurs at line 170.

c. 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. Figure 3.15 will clarify the two subtypes of this dependence further.

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.*

Figures 3.15 (a) and (b) cover only control, inheritance call dependence, internal call dependence and cascading call dependences. Data dependences were omitted for clarity purposes.



```
118        <%=_title%>
147        protected string _title;
152        _title = (idStr == null ? "New" : "Edit") + " To Do List Item";
```

Figure 3.15 (a): Fragment of Edititem.aspx.cs and Edititem.aspx along with the associated inheritance call dependence, internal call dependences (straight dashed line) and cascading call dependence (two curved dashed arrow from one subgraph node to another).

As in Figure 3.15 (a), in 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.

```
22    <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">

34    protected System.Web.UI.WebControls.DataGrid ToDoDataGrid;
62        ToDoDataGrid.Columns.Add(bcOpened);
```

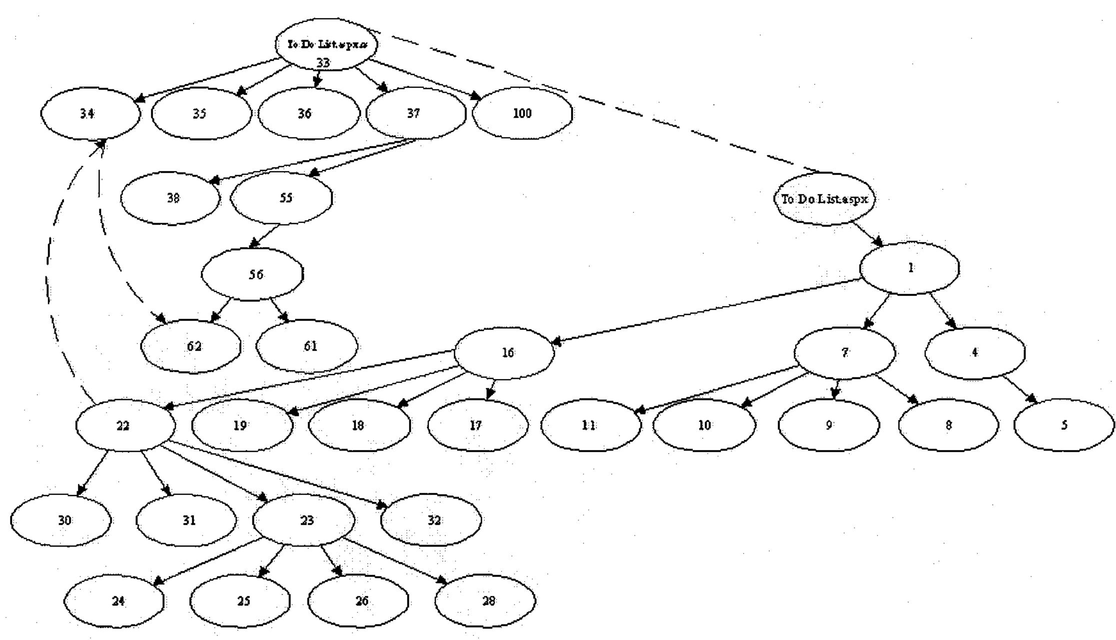


Figure 3.15 (b): fragment todolist.aspx.cs and todolist.aspx along with the associated inheritance call dependence (straight dashed line) and cascading call dependence (two curved dashed arrows).

In todolist.aspx, we encounter the first definition for the datagrid as shown on line 22 of Figure 3.15.(b) then, datagrid plays the role of the type of the ToDoDataGrid variable defined in todolist.aspx.cs (code behind class) on line 34. ToDoDataGrid is subsequently used on line 62. This cascading call dependence has a call dependence side and a data dependence side. Cascading call dependence is represented by two curved dashed arrows.

3.3.4 Semantic Dependences

(Ricca & Tonnella, 2001) addressed semantic dependences concisely, whereas this dependence is of critical importance to .NET web applications. We have praised repeatedly the value of graphics, events and informative text in .NET. Semantic dependences are thought to be the answer to the problem of representing all these elements in the SDG. In fact, we pay much importance to their representation because serious errors may also be sourced in graphical elements.

To satisfy this type of dependences, we propose the addition of new notational elements to the SDG. As we mentioned earlier, the call dependences suggested in (Ricca & Tonnella, 2001) lack the differentiation between different types of calls (whether submitting a button or passing a parameter). In our new notational representation, we will make the effort to reduce this ambiguity.

Another motivation behind these notational elements is that we realized that numbering the code statements and their inclusion in the form of ellipses in the SDG isn't always informative. When the number of statements grows, the SDG becomes lengthy and unreadable.

Besides, some graphical elements may carry information that must be included but the limitations of the present SDG doesn't support their representation.

Since ASP.NET may use both intermingled code and the code behind feature (see section 2.1 for the difference between the two), numbering the statements and using their numbers as done so far doesn't really make the representation very clear. Therefore, we introduce four new elements to the SDG: page, text, image, event and we will keep the ellipse for statements of processing nature as introduced by (Ricca & Tonnella, 2001). 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 valuable information to the interactions between the different elements of the web application.

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 will use the notation in Figure 3.16 to represent a web page (the building block of a web application). The notation shown in Figure 3.17 represents a textual element (text to be included on page). And the notation of Figure 3.18 represents a graphical element (button, text box, drop down, etc).

Figure 3.16: Notational Element to represent a web page in the SDG.

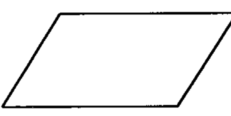


Figure 3.17: Notational Element to represent a textual element in the SDG.

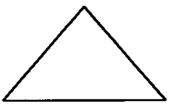


Figure 3.18: Notational Element to represent a graphical element in the SDG.

One basic specification of .NET web application is the overwhelming presence of events. In fact, it has been said enough through this text about the event-driven environment of .NET. Therefore, It is essential to introduce along the new notations added previously a notational 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 representational notation of events that we are presenting is borrowed from the concept of Petri nets.

We will introduce Petri nets first for a better understanding of the representation adapted.

Petri Nets: a Petri net consists of places (represented as circles), transitions (represented as bars), place input/output (represented as directed arcs) and tokens (represented as black dots) as in Figure 3.19 below:

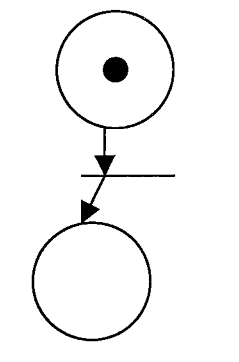


Figure 3.19: A Simple Petri Net.

The dynamics of a Petri net consists of a sequence of transition firing. Upon a transition firing, two things happen:

1. Tokens are taken away from positions (which have arrows going from these positions) to the transition considered.
2. New tokens are placed on positions indicated by arrows that originate from the transition.

In our case, 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 since we have defined our representation for pages as illustrated in Figure 3.16. Therefore the following representation in Figure 3.20 will be used to depict events:

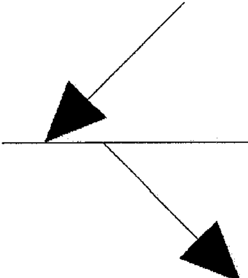


Figure 3.20: Representation of an Event in EDG.

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 3.21 shows the two kinds of tokens. An example will be provided in 3.3.5.

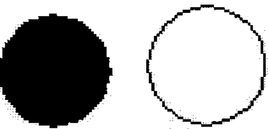


Figure 3.21: Solid dot represents a "positive interaction". White dot represents a "negative interaction".

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.*

The notations introduced so far will be illustrated with an example in section 4.3.1.5, after establishing the event-based dependences.

3.3.5 Event-Based Dependences

In the SDG constructed by (Ricca & Tonnella, 2001), events are implicitly addressed through call dependences. But, in the previous section, we have argued enough about the necessity of splitting the call dependences in order to conserve the objective of the SDG which is providing a better understanding of the web application in question. Therefore, we find it ultimately essential to add other types of dependences to satisfy the main feature of ASP.NET: events. These dependences are link, visible effect and invisible effect.

After the establishment of the event-based dependences, we move to the construction of the Event-Based Dependence Graph (EDG):

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. Figure 3.22 will 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).*

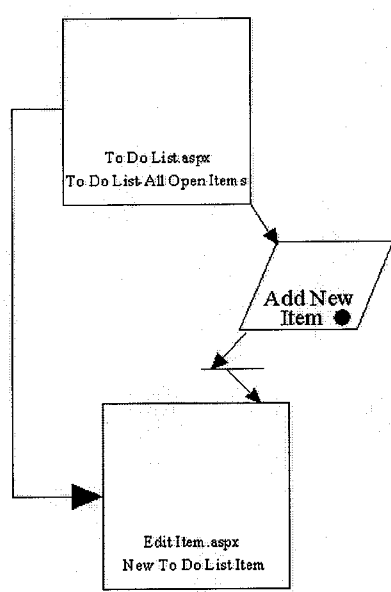


Figure 3.22: Fragment of Event-Based Dependence Graph showing the link dependence between the "open items" page and the "new to do list item" page.

The home page of "To Do List" is "all open items" (represented by a square in Figure 3.22) from where we can go to another page by simply clicking the textual hyperlink "add a new item" (represented by the parallelogram in Figure 3.22). 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.

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. Figure 3.23 illustrates 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.*

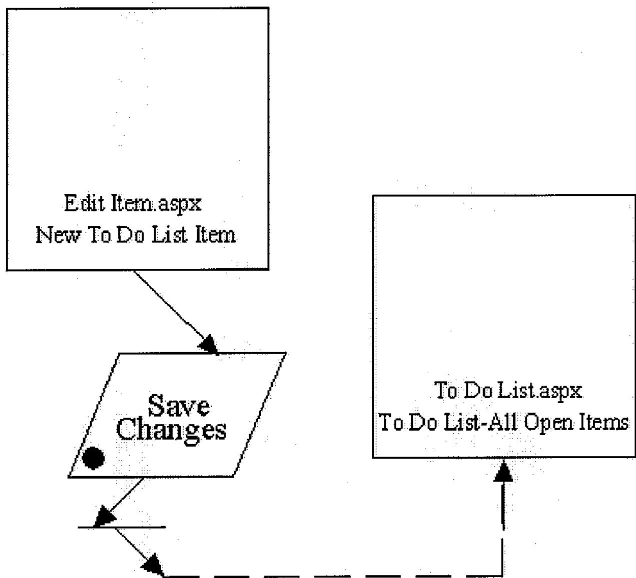


Figure 3.23: Fragment of the Event-Based Dependence Graph for "To Do List" showing the visible effect dependence between the "new to do list item" page and the "all open items" page.

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. Figures 3.5 and 3.6 show this scenario through an example.

<u>**Invisible Effect Dependence**</u>: sometimes when the user clicks a link or button (ex: 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. Figure 3.24 represents this dependency through our application "To Do List".

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.*

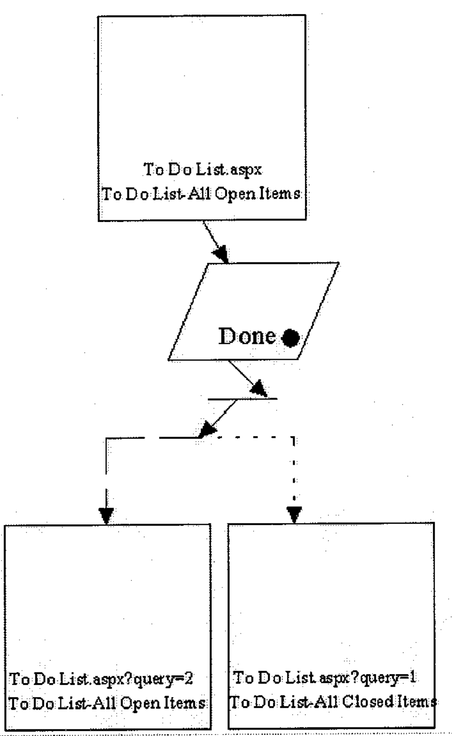


Figure 3.24: Fragment of the Event-Based Dependence Graph for "To Do List" showing both the visible and the invisible effect dependences between the "all open items" page and the "all open items"(visible effect) and the "all closed items" page (invisible effect).

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).

Having presented the event-based dependences, we present in Figure 3.25 the EDG for "To Do List". Uninformative elements were removed for clarity purposes.

All the elements of the EDG were gradually introduced in the previous subsections. Additionally, 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 3.25. 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.

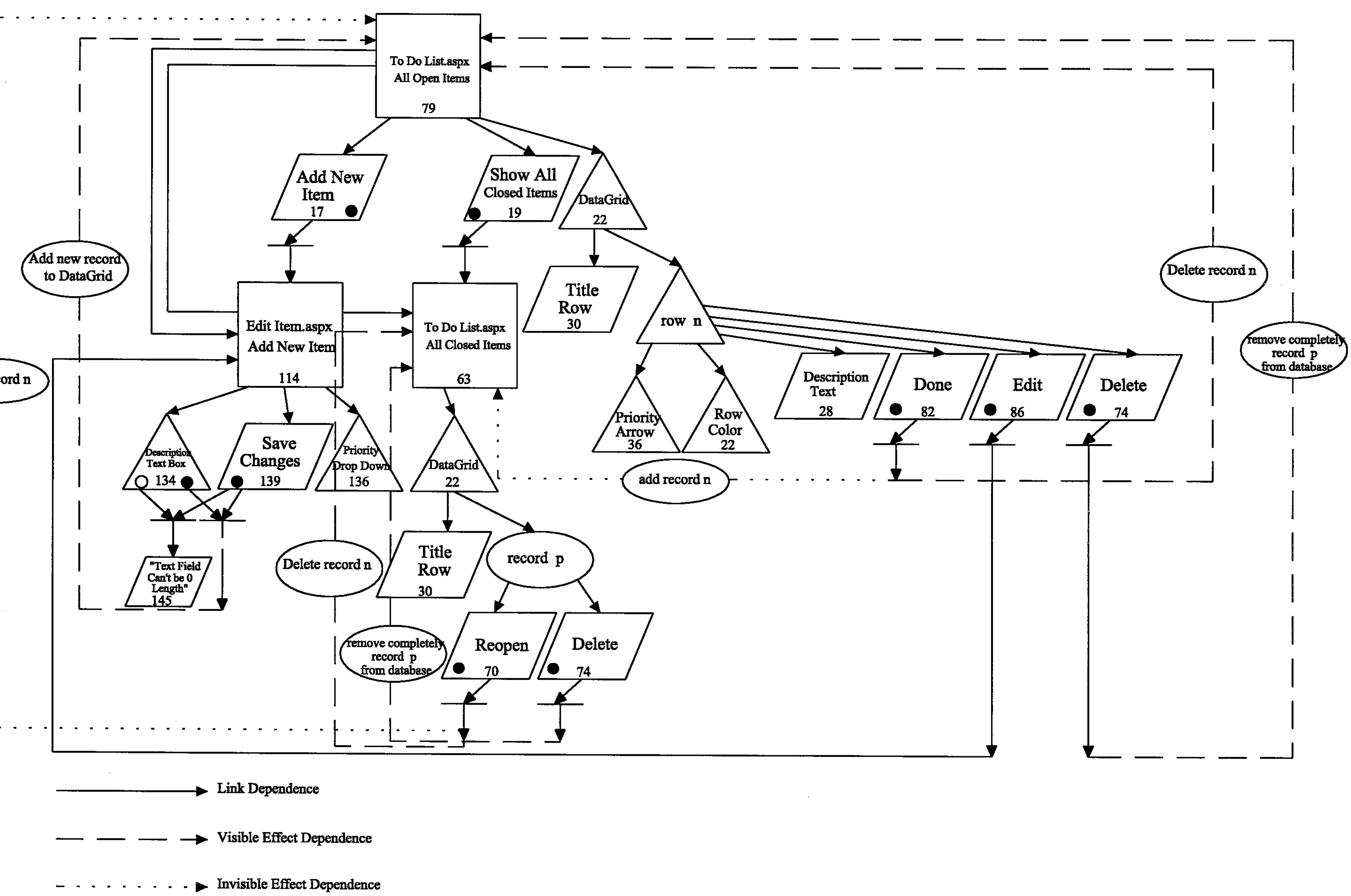


Figure 3.25: Event-Based Dependence Graph for "To Do List".

3.3. Construction of the System Dependence Graph (SDG)

Having presented all the dependences relative to .NET web applications, we are ready now to combine all the dependences in the SDG. The SDG will be the source for a better understanding of any .NET web application as it covers the main sources of ambiguity and eventually errors for ASP.NET web applications.

To construct the SDG, we perform the following steps:

1. Identify the pages of the application:
    a. For each page the application contains, we represent it with a square containing the name of the page. i.e., "To Do List.aspx" (All Open Items).
    b. Connect - through control dependence arrows - the graphical, textual, and computational elements pertaining to each of these pages.
    c. Place tokens on elements where the "user's interaction" (whether positive or negative) is permitted and attach to them events representations.

2. Connect pages having link dependence through square solid arrows.
3. Connect pages having visible effect dependence through square dashed arrows.
4. Connect pages having invisible effect dependence through dotted square arrows.
5. Identify code behind pages for each of the presentation pages (.aspx.cs):

    a. Connect control dependences belonging to each code behind class through straight arrows.
    b. Connect data dependences for each code behind class through curved arrows.
    c. Connect internal call dependences for each code behind class through dashed lines.
    d. For each presentation –code behind pair, connect them through inheritance call straight dashed line.
    e. Identify elements of cascading call dependence and connect them together through two curved dashed arrows.

The SDG of "To Do" is partially represented (for clarity purposes) in Figure 3.26.

Appendix A contains the full listing of all the application files along with all the dependences graphs.

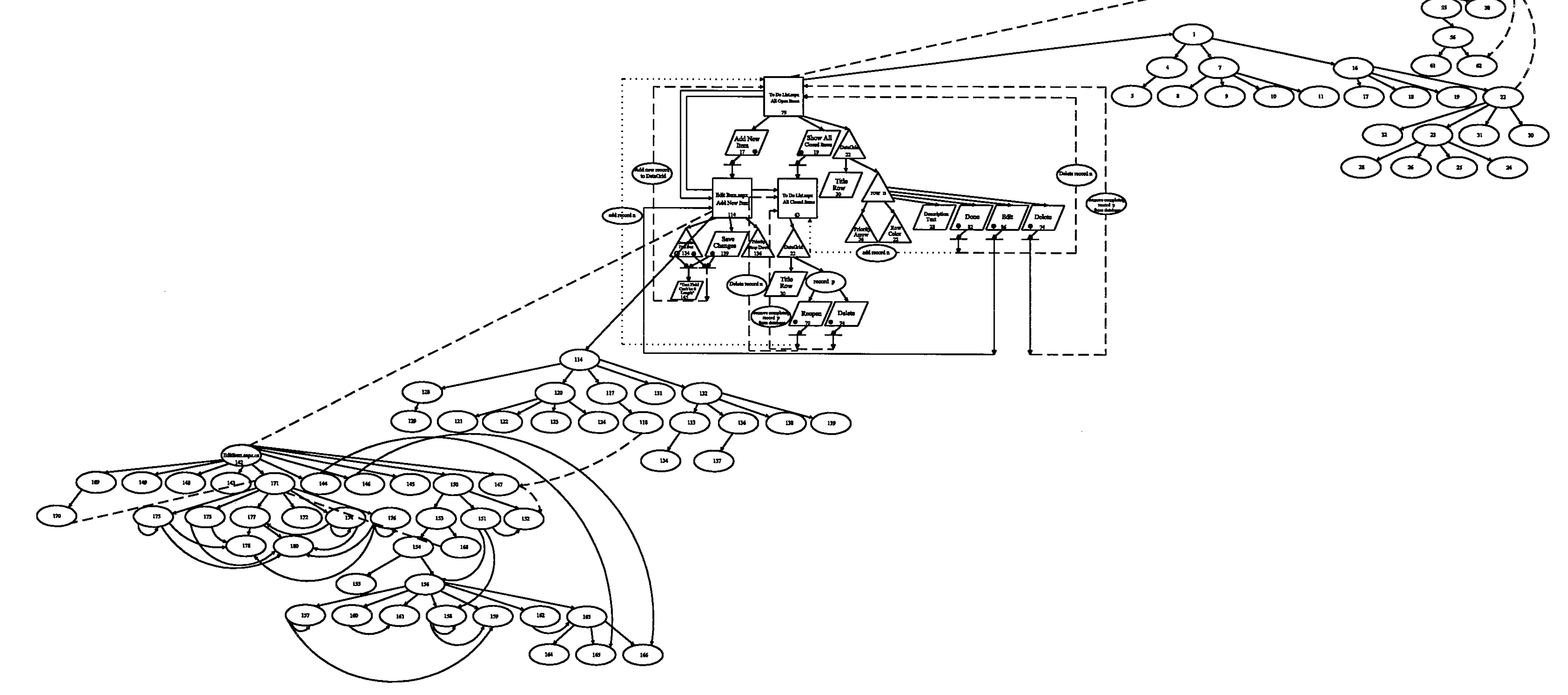


Figure 3.26: System Dependence Graph for "To Do List"

Chapter 4

Data Flow Testing & Event Flow Testing for Web Applications

Data flow testing, a white-box testing introduced in chapter 2, was employed for testing web applications. We review the previous work in testing web applications using data flow testing. Then we customize this experience and add to it event flow testing, in order to cover, when testing, all the main features of .NET web applications.

4.1 Previous Work in Data Flow Testing of Web Applications

In the work of (Liu et al., 2001), traditional data flow testing was extended to cover web applications. (Liu et al., 2001) provide a structure model where each component of the web application is modeled as an object. Then, the data flow information of the web application is captured using flow graphs. Eventually, they derived data flow test cases for the web application based on the intra-object, inter-object, and inter-client perspectives.
We review the structure model and the corresponding graphs of (Liu et al., 2001). Then, we examine the five level testing approach adopted.

4.1.1 Structure Model

The structure model consists of four types of flow graphs used to capture data flow information of a web application. These flow graphs are described below.

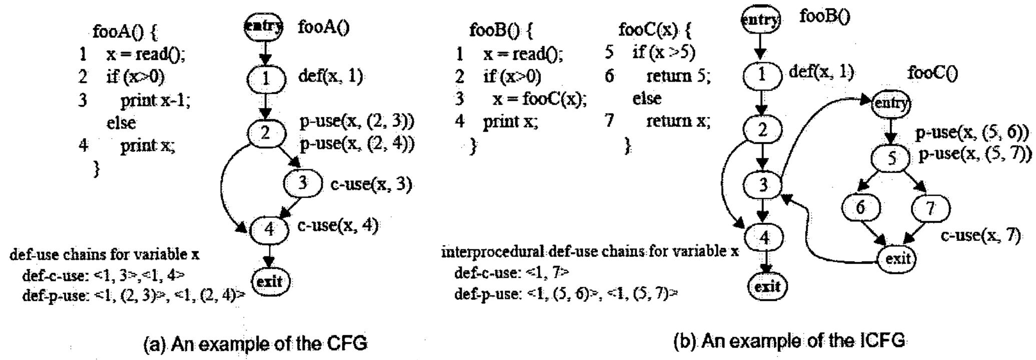
4.1.1.1 Control Flow Graph (CFG)

The CFG is used to depict the def-use chains of variables of an individual function.

4.1.1.2 Interprocedural Flow Graph (ICFG)

The ICFG is the hybrid of call graphs and CFG's def-use chains of variables defined in one function and used in other functions.

Figure 4.1 depicts an example for CFG and ICFG.



(a) An example of the CFG

(b) An example of the ICFG

Figure 4.1: Examples of CFG and ICFG.

4.1.1.3 Object Control Flow Graph (OCFG)

To obtain the def-use chains that result from different function invocation sequence within an object, an Object Control Flow Graph is employed. The OCFG is constructed by connecting together all the ICFGs (or CFGs) of the functions within an object. An example of an OCFG is depicted in Figure 4.2.

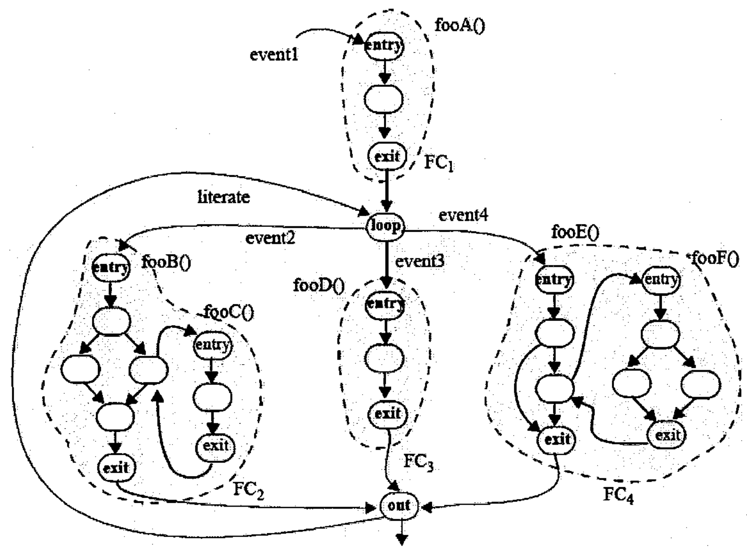


Figure 4.2: Example of the OCFG of a Web page object.

4.1.1.4 Composite Control Flow Graph (CCFG)

To describe the data flow information between interacting web pages, the Composite Control Flow Graph (CCFG) is introduced so that the def-use chains across web pages can be obtained. The CCFGs can be constructed by connecting the related CFGs (or ICFGs) of the interacting web pages together. Figure 4.3 shows an example of the CCFG for an HTTP request.

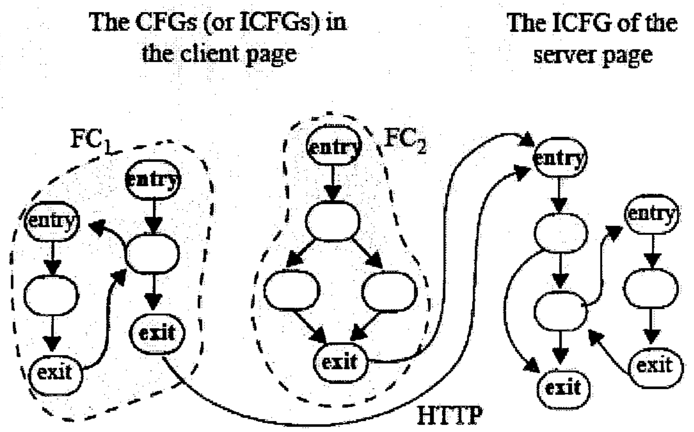


Figure 4.3: Example of the CCFG for an HTTP request between the client and server pages.

4.1.2 Five-Level Data Flow Testing Approach

Since a web application is represented using objects as basic units in the testing model, data flow test cases can be derived from three different perspectives: intra-object, inter-object, and inter-client. From the intra-object, inter-object, and inter-client perspectives, data flow testing for web applications can be accomplished via a five-level testing approach. For the intra-object perspective, def-use chains require to be computed at function, function cluster, and object levels. For the inter-object perspective, def-use chains need to be derived at object cluster level. For the inter-client perspective, def-use chains need to be considered at the application level. We describe each of the levels next.

4.1.2.1 Function Level

This level is used to test individual functions for the variables that have def-use chains limited to a single function. These def-use chains can be obtained from the CFG of the function.

4.1.2.2 Function Cluster Level

This level of testing is used to test a cluster of functions within an object for the variables whose def-use chains involve more than one function within the cluster. This level of testing is required if the def-use chain for a definition in one function consists of uses of the definition in other functions of the cluster. The def-use chains for a function cluster can be computed from an ICFG.

4.1.2.3 Object Level

Object level testing is used to test an object that has variables whose def-use chains can be changed by different function invocation sequences. The def-use chains of different function invocation sequences can be computed from the OCFG of an object.

4.1.2.4 Object Cluster Level

An object cluster is a set of objects that are associated via message passing. Object cluster level testing is used to test a cluster of objects for the variables whose def-use chains cross the objects in the cluster. Test paths for this level can be derived from the ICFG of involved functions in the object cluster.

4.1.2.5 Application Level

Application level testing is used to test application-scope variables whose def-use chains cross different clients that can access the application. Test paths for the application level can be derived from the reachability graph that is constructed using the CFGs, relating to the application-scope variables of the concurrent clients.

4.2 Data Flow Testing of .NET Web Application

We use the previous experience of (Liu et al., 2001) in data flow testing of web applications and adapt it to suit .NET web applications based on the dependence graphs we have provided earlier in chapter 3. Analogically, our dependence graphs resemble to the structure model of (Liu et al., 2001). For .NET web applications, three data flow testing levels from the previous work are applied in our work. Function level, function cluster level and object level. Each of these levels is detailed below.

4.2.1 Function Level

Even though we followed the same naming pattern of (Liu et al., 2001) for .NET web applications testing levels. Yet, we have to note that the word "function" in our work refers to a "method" of a code-behind class. To put things in context, we are testing the method level of a class, but we will keep the naming as introduced by (Liu et al., 2001).
This level is used to test individual methods for the variables that have def-use chains limited to a single method inside a code-behind class. The def-use chains for a single method can be obtained from the data dependence graph for each method.

Again, we will be illustrating every step of the work through "To Do List", the .NET web application.

We represent in the data dependence graph, (as thoroughly explained in chapter 3), a data dependence with a curved arrow emanating from the definition statement to the use statement. Two methods of the code-behind class "edititem.aspx.cs" have a data dependence graph: OnSubmit and Page_Load. Figures 4.4 and 4.5 show the data dependence graphs of both along with their associated code fragments. Variables in question are highlighted on the code excerpts. Tables 4.1 and 4.2 give the def-use chains for each of the variables in the two methods for the function level testing.

```
171     protected void OnSubmit()
        {
172             string connStr = ConfigurationSettings.AppSettings["ConnectionString"];
173             string sql;
174             string idStr = Request.Params["id"];
175             string desc = DescriptionTextBox.Text.Replace("'", "''");
176             int priority = PriorityList.SelectedIndex + 1;

177             if (idStr == null)
178                     sql = "INSERT INTO Items (Description, Priority) VALUES ('" + desc + "', " + priority + ")";
179             else
180                     sql = "UPDATE Items SET Description = '" + desc + "', Priority=" +
                                priority + " WHERE ID=" + idStr;
```

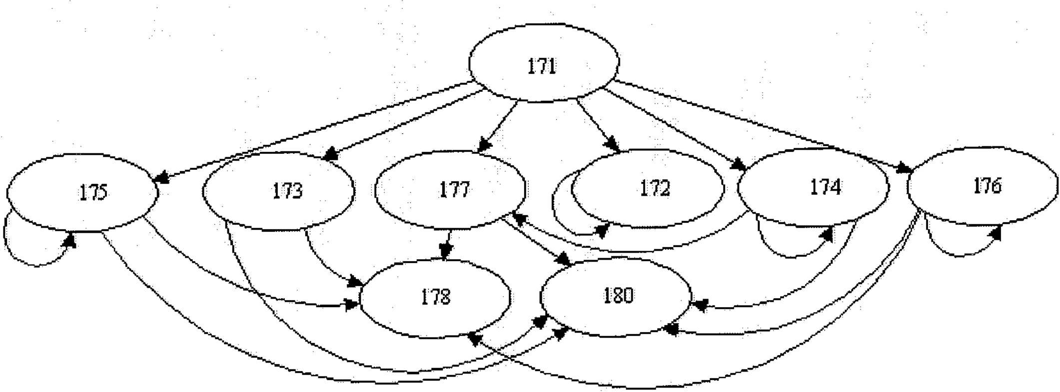


Figure 4.4: "OnSubmit" method fragment of "Edititem.aspx.cs" with the associated data dependence graph.

Table 4.1: Variables of method OnSubmit and their def-use chains on the function level.

Method	Variable	Test Level	Def-Use Chains
OnSubmit	**ConnStr**	Function	<172, 172>
	Sql	Function	<173, 178>, <173, 180>
	IdStr	Function	<174, 174>, <174, (177, 180)>
	Desc	Function	<175, 175>, <175, 178>, <175, 180>
	Priority	Function	<176, 176>, <176, 178>, <176, 180>

The variable **IdStr** belonging to OnSubmit has a p-use chain as shown on Table 4.1. We also note that some def-use chains are self-referring. Example, the variable **sql** is defined and c-used at the same statement. The variable **row** has a self referring def-c-use on line 164. The variable **tbl** has a def-p-use. **Tbl** is assigned on line 162 and used in the predicate (163, 164).

```
150     private void Page_Load(object sender, System.EventArgs e)
        {
151             string idStr = Request.Params["id"];
152             _title = (idStr == null ? "New" : "Edit") + " To Do List Item";
153             if (!IsPostBack)
                {
154                     foreach (string s in _priorities)
155                             PriorityList.Items.Add(s);

156                     if (idStr != null)
                        {
157                             string connStr = ConfigurationSettings.AppSettings["ConnectionString"];
158                             string queryStr = "select * from Items where id=" + idStr;
159                             OleDbDataAdapter adapter = new OleDbDataAdapter(queryStr, connStr);
160                             DataSet ds = new DataSet();
161                             adapter.Fill(ds);
162                             DataTable tbl = ds.Tables[0];
163                             if (tbl.Rows.Count > 0)
                                {
164                                     DataRow row = tbl.Rows[0];
165                                     DescriptionTextBox.Text =row["Description"].ToString();
166                                     PriorityList.SelectedIndex = (int)row["Priority"] - 1;
                                }
```

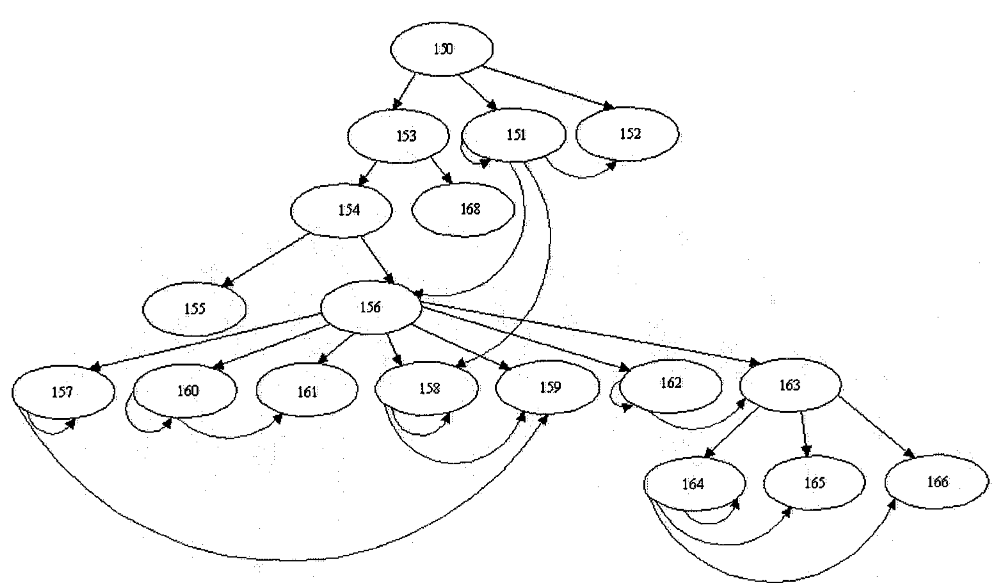


Figure 4.5: "Page_Load" method fragment of "Edititem.aspx.cs" with the associated data dependence graph.

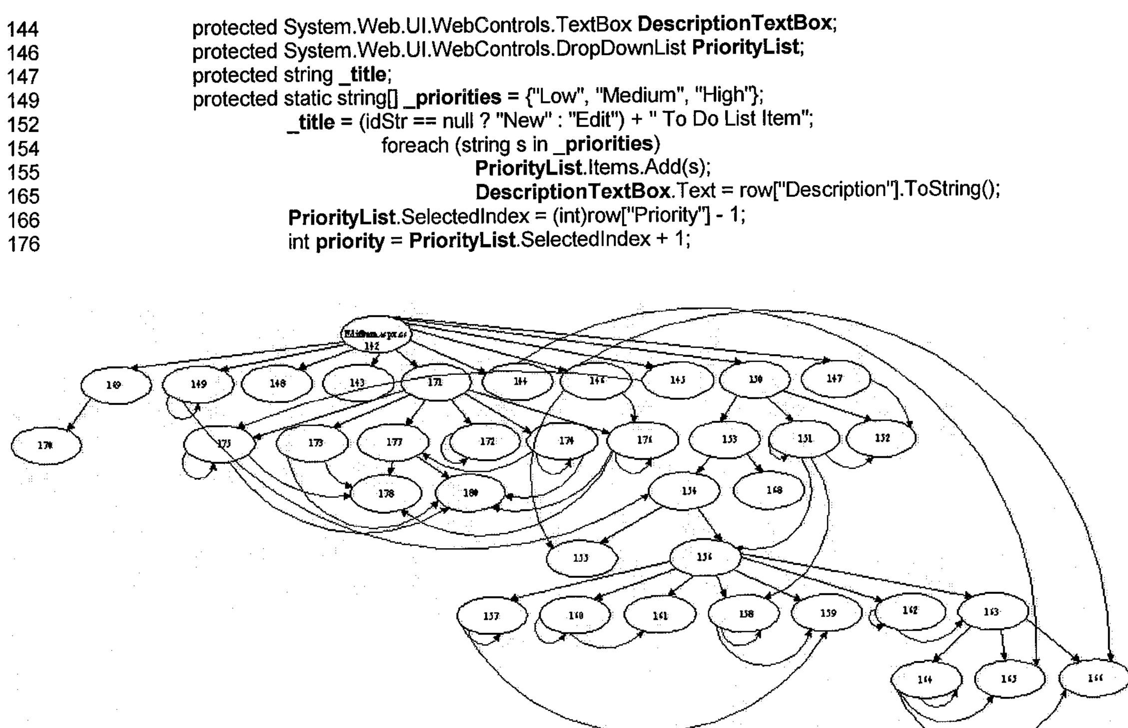
Table 4.2: Variables of method Page_Load and their def-use chains on the function level.

Method	Variable	Test Level	Def-Use Chains
Page_Load	**ConnStr**	Function	<157, 157>, <157, 179>
	IdStr	Function	<151, 151>, <151, 152>, <151, 158>
	QueryStr	Function	<158, 158>, <158, 159>
	Ds	Function	<160, 160>, <160, 161>, <160, 162>
	Tbl	Function	<162, 162>, <162, (163, 164)>
	Row	Function	<164, 164>, <164, 165>, <164, 166>

4.2.2 Function Cluster Level

The name "function cluster" actually refers in ASP.NET web application to "method cluster", but as mentioned earlier, we use the same naming pattern of (Liu et al., 2001).

This level is used to test methods for the variables that have def-use chains limited to a single code-behind class. The corresponding def-use chains can be obtained from the data dependence graph for "edititem.aspx.cs". In Figure 4.6, we present the data dependence graph for "edititem.aspx.cs" along with the associated code excerpt.



```
144    protected System.Web.UI.WebControls.TextBox DescriptionTextBox;
146    protected System.Web.UI.WebControls.DropDownList PriorityList;
147    protected string _title;
149    protected static string[] _priorities = {"Low", "Medium", "High"};
152            _title = (idStr == null ? "New" : "Edit") + " To Do List Item";
154                    foreach (string s in _priorities)
155                            PriorityList.Items.Add(s);
165                            DescriptionTextBox.Text = row["Description"].ToString();
166            PriorityList.SelectedIndex = (int)row["Priority"] - 1;
176            int priority = PriorityList.SelectedIndex + 1;
```

Figure 4.6: "Edititem.aspx.cs" code fragment with associated data dependence graph.

Now, we identify in Table 4.3, the variables and their associated def-use chains of "edititem.aspx.cs" on the function cluster level.

Table 4.3: Variables of code-behind class "edititem.aspx.cs" and their def-use chains on the function cluster level.

Class	Variable	Test Level	Def-Use Chains
Edititem.aspx.cs	**Description TextBox**	Function Cluster	<144, 165>, <144, 175>
	Priority List	Function Cluster	<146, 166>, <146, 176>, <146, 155>
	_title	Function Cluster	<147, 152>
	_priorities	Function Cluster	<149, 149>, <149, 154>

4.2.3 Object Level

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.

```
118     <%=_title%>
152     _title = (idStr == null ? "New" : "Edit") + " To Do List Item";
```

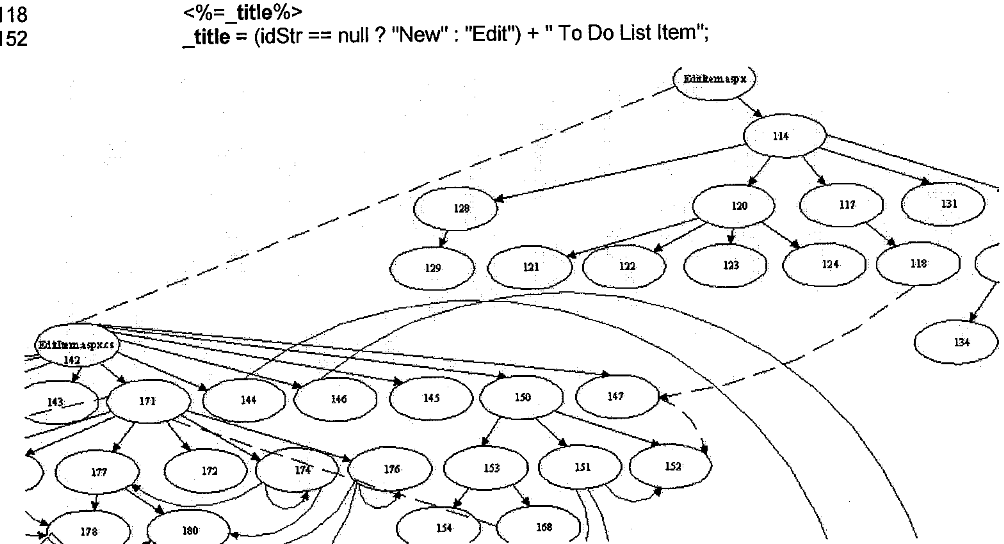


Figure 4.7: "Edititem.aspx.cs" and "edititem.aspx" code fragment with associated cascading call dependence graph.

Figure 4.7 illustrates this dependence, and Table 4.4 identifies the variable of this object and its associated def-use chains.

Table 4.4: 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>

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.

4.3 Event Flow Testing of .NET Web Applications

In the previous section, we applied partially the five-level testing approach of (Liu et al., 2001) to .NET web applications. However, we believe that events in web applications are important and need additional testing. For this purpose, we propose in this section event flow testing that focuses on events and their ripple effect. We identify two new types of testing "levels" pertaining to the events: fetching and updating. With these two new testing levels, a new type of chains is also introduced: the trigger-effect chain.

4.3.1 Fetch Level Testing

In the event flow testing, we attempt testing all events in an .NET web application by implementing a similar pattern to def-use pairs. For event flow testing, we talk about trigger-effect chain (in analogy to def-use chain).

Trigger-effect chain: is a triplet of the following form:

<Page m, Page p, (Event i, Event ii, …, Event n)>,

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 chapter 3, we detailed 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 3.25 illustrates the EDG for the ASP.NET web application "To Do List". In Table 4.5, we represent the trigger-effect chains on the fetch level for this example.

Table 4.5: Events of EDG "To Do List" and its corresponding trigger-effect chains on 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)>

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.

4.3.2 Update Level Testing

In chapter 3, two other dependences were presented: visible effect dependence and invisible-effect dependence. Data flow testing on the update level regards these two dependences. That's why it is necessary to differentiate two subtypes 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 4.6 and 4.7, we represent the trigger-effect chains on the fetch level for "To Do List".

Table 4.6: Events of EDG "To Do List" and its corresponding trigger-v-effect chains on 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 4.7: 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>

We observe from tables 4.6 and 4.7, 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".

Having presented event flow testing for .NET web applications, we summarize both data flow def-use chains and event flow trigger-effect chains in Table 4.8 of section 4.4.

4.4 Summary Def-Use and Trigger-Effect Chains Table for "To Do List"

Table 4.8 summarizes all the data flow and event flow testing chains.

Table 4.8: Summary of Data Flow and Event Flow def-use and trigger-effect chains for "To Do List".

Method	Variable	Test Level	Def-Use Chains
OnSubmit	ConnStr	Function	<172, 172>
	Sql	Function	<173, 178>, <173, 180>
	IdStr	Function	<174, 174>, <174, (177, 180)>
	Desc	Function	<175, 175>, <175, 178>, <175, 180>
	Priority	Function	<176, 176>, <176, 178>, <176, 180>
Page_Load	ConnStr	Function	<157, 157>, <157, 179>
	IdStr	Function	<151, 151>, <151, 152>, <151, 158>
	QueryStr	Function	<158, 158>, <158, 159>
	Ds	Function	<160, 160>, <160, 161>, <160, 162>
	Tbl	Function	<162, 162>, <162, (163, 164)>
	Row	Function	<164, 164>, <164, 165>, <164, 166>
Class	**Variable**	**Test Level**	**Def-Use Chains**
Edititem.aspx.cs	Description TextBox	Function Cluster	<144, 165>, <144, 175>
	Priority List	Function Cluster	<146, 166>, <146, 176>, <146, 155>

	_title	Function Cluster	<147, 152>
	_priorities	Function Cluster	<149, 149>, <149, 154>

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

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)>

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>

4.5 Coverage-Based Testing for .NET Web Applications

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 traditional approach for code coverage includes code statement coverage, branch coverage, path coverage and many more. In this section, we present a new approach to coverage-based testing that targets web applications. In addition to the previously mentioned coverage testing types, we see it necessary to add hyperlinks 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.

4.5.1 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, with the assistance of the EDG of "To Do List", we can spot three solid square arrows. Providing test cases that exercise all these arrows will assure all-hyperlinks coverage. Figure 4.8 contains only the portion of the EDG that takes care of hyperlinks.

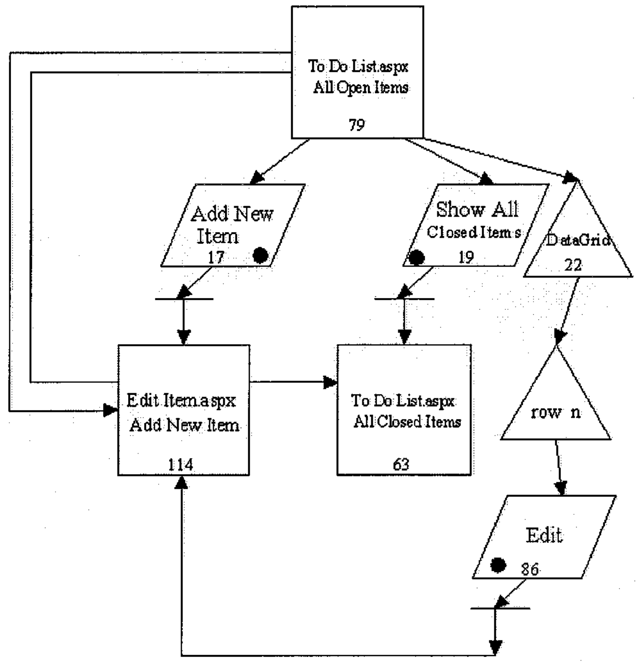


Figure 4.8: All-Hyperlinks Coverage.

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. The other two test cases follow the same pattern as depicted in Figure 4.8

4.5.2 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 3.25, we can construct many test cases that assure all-Input-GUI coverage.

4.5.3 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 3.25.

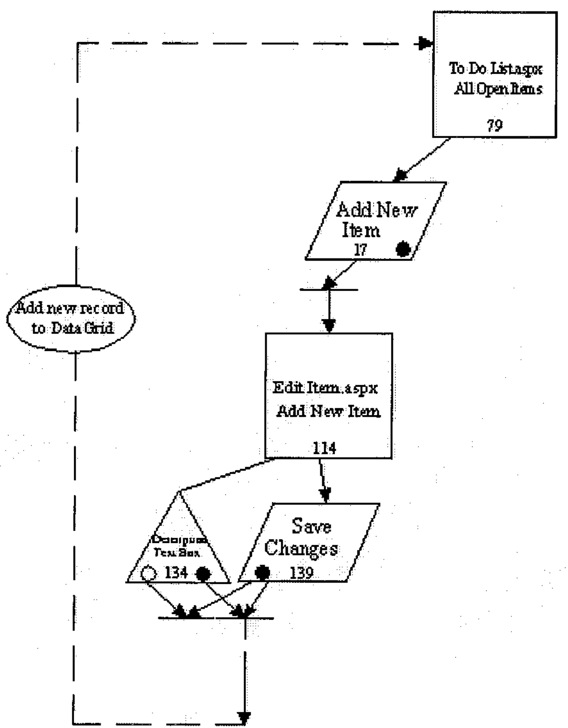


Figure 4.9: All-Events Coverage.

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 3.25.

Chapter 5
Mutation Testing of ASP.NET Web Applications

Two major schemes in fault based testing and evaluation are competing: data flow coverage testing and mutation testing. Some experiments have related the first scheme to high software reliability but no conclusive evidence about the effectiveness of coverage exists. As for the mutation testing, it was empirically performed on a wider scale where artificially generated mutants are based on hypothetical faults. The testing process produces an enormous number of mutants, and each mutant must be recompiled and tested. Some of the mutants produced are too trivial (very easily killed) or too unrealistic (may never be activated).

Nevertheless, experiments show that mutation coverage has no equivalent in the effectiveness of detecting software faults. It is believed that despite its relatively high cost (Ma, Kwon, & Offutt, 2002), mutation is considered a more truthful indicator of testing quality. All previously applied experiences in mutation testing to traditional coding and OO coding were for us a source of confidence to investigate mutation testing on .NET web applications.

After the generation of test cases for testing ASP.NET web applications, a question arises about the adequacy of these test cases: How well-revealing are the test cases? Mutation testing is a technique for evaluating the adequacy of test cases. we apply it on .NET web applications by proposing mutation operators that take advantage of the code-behind feature in .NET. One group of the operators presented is designed to take care of the event feature of .NET web applications.

5.1 Mutation Operators for .NET

The effectiveness of mutation testing depends heavily on the types of faults that the mutation operators are designed to represent. Therefore, the quality of the mutation operators is the key to mutation testing (Ma, Kwon, & Offutt, 2002).

In order to find mutation operators, the best resort was to transform any ASP.NET file into: a code-behind file and a presentation file. Actually, interlocking ASP and HTML makes it hard to locate the fault we are trying to test for. Given this advantage in .NET technology, we move all the coding in a behind file written in VB.NET or C#.NET and keep the presentation and layout tags of HTML in the aspx file.

The separation of code from the presentation leads us to two main areas to test: the VB.NET (or C#.NET eventually) file and the aspx file (the webform in general). Since code-behind files enclose classes to be inherited in the aspx file, we can approach these files with an OO mutation testing. Previous work exists for OO mutation testing and will be applied where appropriate. As for the presentation file, mainly ASP.NET server controls are to be tested using traditional mutation operators. Remains the group of operators targeted to test for the main element of web applications: event.

In the following subsections, we present four different groups of operators that apply to ASP.NET web applications after separating code from presentation. The first two groups are applied to the OO code-behind file, the third to the presentation file, and finally the fourth is applied on the event level of the web application.

5.1.1 Method Level Mutation Operators

Even though we are testing the code-behind class, testing the methods themselves can be satisfied by the application of the traditional mutation operators described below:

1. **ORO - Operand Replacement Operator**: it replaces a single operand with another operand or constant:

Original Code	**ORO Mutant**
X > Y	X < Y

2. **EMO - Expression Modification Operator**: it replaces or inserts new operators:

Original Code	**EMO Mutant**
If (X = = Y)	If (X >= Y)

3. **SMO - Statement Modification Operator:** removal of part or all of a statement:

Original Code	**SMO Mutant**
If (X = = Y) then Y = 4	If (X = = Y) then Y = 4
Else Y = 5	//Else Y = 5

5.1.2 Class Level Mutation Operators

After testing the methods of a class independently, other class mutation operators need to be applied on the class level to test for the faults that may be introduced due to the OO paradigm. The OO features that are mainly addressed are respectively polymorphism, overloading, information hiding. In this category, we also address exception handling though it is not unique to OO:

1. **ICE – class Instance Creation Expression changes**: this operator replaces the class name in 'instance creation expression' with compatible class names (Kim, Clark, & McDermid, 2000). This results in calling the constructors of compatible types, which will create the objects of the replaced types. This operator is also used to handle object initialization. Initialization faults are the most frequent errors irrespective of language paradigm (Kim, Clark, & McDermid, 2000):

Original Code	**ICE Mutant**
S s = new S(1, "Login");	S s = new U(1, "Login", "Msg");

2. **POC – method Parameter Order Change:** this operator changes the order of parameters in method declarations if the method has more than one parameter:

Original Code	POC Mutant
Public LogMsg (int level, String key, Object [] inserts)	Public LogMsg (String key, int level, Object [] inserts)

3. **VMR – oVerloading Method declaration Removal**: this operator removes a whole method declaration of overloading/overloaded methods. The intent of this operator is to check if all the overloading/overloaded methods are invoked at least once because test data must reference the method in order to notice that the method has been deleted.

4. **AOC – Argument Order Change:** this operator changes the order of arguments in method invocation expressions, if more than one argument exists:

Original Code	AOC Mutant
Trace.entry("Logger", "addLogCatalogue");	Trace.entry("addLogCatalogue", Logger");

The second, third and fourth operators are meant to check if one of the overloaded methods is mistakenly invoked instead of the intended ones, as it may happens to be better matched.

5. **AMC – Access Modifier Changes**: the job of this operator is to guide testers to generate enough test cases for testing accessibility/visibility:

Original Code

```
Public LogMsg (int level, String key, Object [] inserts)
Public LogMsg (int level, String key, Object inserts)
```

AMC Mutant

```
Private LogMsg (int level, String key, Object [] inserts)
Public LogMsg (int level, String key, Object inserts)
```

By replacing 'public' with 'private', the first constructor becomes unavailable and all instance creation expressions that used to call the first constructor, will now use the second constructor.

5.1.3 Presentation Level Mutation Operators

After testing the classes behind of the ASP.NET application, one should test the presentation file mainly constructed of server controls and simple HTML.
Since most work on the presentation file involve graphical components, no complicated faults will be exhibited. In fact, most of the presentation file can be done in design view. This leaves us with very simple categories of faults yet with expensive inconveniences if left untested.

1. **ICC - Inherited Class Change**: this operator takes care of testing whether or not we are inheriting in the presentation file the correct code-behind class. Sometimes, when numerous classes and presentation files exist, the programmer might be confused and may inherit for a specific presentation file the wrong class. Example:

   Original code:
   ```
   <%@ Page Inherits="CodeBehindDemo01"src="codebehinddemo01.vb"%>
   ```

   ICC Mutant:
   ```
   <%@ Page Inherits="CodeBehindDemo02"src="codebehinddemo02.vb"%>
   ```

2. **EIO - Element Interchange Operator:** this operator swaps or rotates elements of same type. This operator tests if the associated image file for example is the one that should be invoked. This operator makes sure we are not associating the wrong element to the wrong item. Example:

Original code:

```
<select id="select1" runat="server" NAME="select1">
    <option value="smiley.gif" selected>Smiley</option>
    <option value="angry.gif">Angry</option>
    <option value="stickman.gif">Stickman</option>
</select>
```

EIO Mutant:

```
<select id="select1" runat="server" NAME="select1">
    <option value="angry.gif" selected>Smiley</option>
    <option value="stickman.gif">Angry</option>
    <option value="smiley.gif">Stickman</option>
</select>
```

3. **OVO - Opposite Value Operator:** this operator makes sure that the displayed text corresponds to the resulting output, i.e. the message reads successful when the payment transaction is "actually" done (payment was approved). This operator makes sure that the programmers are issuing the right message for the right event. Example:

Original code:

```
<asp:CompareValidator id="compval" Display="dynamic" ControlToValidate="txt1"ControlToCompare="txt2" ForeColor="red" BackColor="yellow"Type="String" EnableClientScript="false" Text="not equal" runat="server" />
```

OVO Mutant:

```
<asp:CompareValidator id="compval" Display="dynamic" ControlToValidate="txt1" ControlToCompare="txt2" ForeColor="red" BackColor="yellow" Type="String" EnableClientScript="false" Text="equal" runat="server" />
```

4. **ERO - Element Removal Operator:** this operator tests whether all elements of a list are included especially in long lists where easily elements can be missed. Example:

Original code:

```
<asp:CheckBoxList id="check1" AutoPostBack="True" TextAlign="Right"
OnSelectedIndexChanged="Check" runat="server">
    <asp:ListItem>Item 1</asp:ListItem>
    <asp:ListItem>Item 2</asp:ListItem>
    <asp:ListItem>Item 3</asp:ListItem>
    <asp:ListItem>Item 4</asp:ListItem>
    <asp:ListItem>Item 5</asp:ListItem>
    <asp:ListItem>Item 6</asp:ListItem>
</asp:CheckBoxList>
```

ERO Mutant:

```
<asp:CheckBoxList id="check1" AutoPostBack="True" TextAlign="Right"
OnSelectedIndexChanged="Check" runat="server">
    <asp:ListItem>Item 1</asp:ListItem>
    <asp:ListItem>Item 2</asp:ListItem>
    <asp:ListItem> Item 3</asp:ListItem>
    <asp:ListItem> Item 4</asp:ListItem>
    <asp:ListItem>Item 6</asp:ListItem>
</asp:CheckBoxList>
```

5.1.4 Event Level Mutation Operators

After testing the class and the presentation file of the web application independently, we need to test for the interaction between the two components. This interaction is provided through events. we propose in this group three operators ready to test for events.

1. **VRO - eVent Removal Operator**: this operator takes care of testing whether hyperlinks are activated correctly, i.e., we test if the hyperlink is "clickable" and able to take us to another page. Example:

Original code:

```
.....
<A href="EditItem.aspx">Add New Item</A>
.....
```

VRO mutant:

```
.....
Add New Item
.....
```

2. **VIO – eVent Implementation Removal Operator:** this operator ensures that the effect of the fired event was implemented correctly. In "To Do List", when the user wishes to close a task, he clicks "done". This click must remove the record from "all open items" page where he is and must add the record to the "closed items" page. In this example, this operator makes sure that the deletion and addition are taking place correctly. Records are "actually" being removed or added to the database.

Original code:

```
bcDelete = new ButtonColumn();
            bcDelete.Text = "Delete";
            bcDelete.CommandName = "DeleteToDo";
    ToDoDataGrid.Columns.Add(bcDelete);
...
switch (cmdStr)
            {
                    case "DeleteToDo":

            sql = "DELETE FROM Items WHERE ID=" + idStr;
....
```

VIO mutant:

```
switch (cmdStr)
            {
                    case "DeleteToDo":

                    //sql = "DELETE FROM Items WHERE ID=" + idStr;
....
```

3. **VSO – eVent Swap Operator**: this operator ensures that the intended effect for the fired event was implemented, i.e, we don't delete a record when we press add for example. Through this operator, we ensure that correctly implemented effects aren't swapped.

Original code:

```
            bcDelete.CommandName = "DeleteToDo";
    bcReopen.CommandName = "ReopenToDo";
...
case "DeleteToDo":
            sql = "DELETE FROM Items WHERE ID=" + idStr;
                            break;

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

VSO mutant:

```
            bcDelete.CommandName = "ReopenToDo";
    bcReopen.CommandName = "DeleteToDo ";
...
case "DeleteToDo":
            sql = "DELETE FROM Items WHERE ID=" + idStr;
                            break;

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

Chapter 6

Conclusions & Future Work

In this thesis, we have extended some traditional white-box testing techniques - data flow testing, coverage-based testing and mutation testing – with new components in order to provide a better-suited testing process for web applications. The new components introduced address the special features of web applications which are not present in traditional applications.

1. We have identified the features of web applications that make the present traditional white-box testing insufficient. These features are mainly the event driven environment, the rich GUIs and the extensive server scripting.
2. We focused on the newest environment under which most recent web applications are being modeled: .NET.
3. We benefited from the main features of this environment and used them in favor of testing web applications. These main features are: being event-driven, compatibility with many programming languages, code separation from presentation front, organization of large amount of classes through namespaces, etc.
4. We have adjusted previously established dependence graphs to fit the characteristics of .NET web applications. We also established a new dependence graph based on the main characteristic of web applications: events. We called this graph the event-based dependence graph.
5. We used our event-based dependence graph to apply event flow testing. Besides def-use chains, we set up a new type of chains: event trigger-effect chains. Both types of chains serve in providing test cases.
6. We also applied coverage-based testing to web applications. We added coverage testing methods strictly related to .NET web applications: all-hyperlinks testing, all-input GUI testing, and all events testing.

7. Finally, in order to evaluate the adequacy of the test cases generated using the previous enhanced techniques, we resorted to mutation testing. The traditional operators don't cover the event feature of web applications. In this context, we established a new group of operators that takes care of the event feature.

Future work may involve empirical work to apply all above presented work on real-world .NET web applications. Also this work provides a basis for regression testing .NET web applications.

References

Barbery, S. and Strohmeier, A. (1994) The problematics of testing object-oriented software. *In Proceedings of the second Conference on Software Quality Management,* Edinburgh, Scotland, UK, pp. 411-426.

Beizer, B. (1990) *Software Testing Techniques*, New York, Van Nostrand Reinhold.

Chen, T.Y., Tse, T. H., and Zhou, Z. (2003) Fault-based testing without the need of oracles. *Information and Software Technology* 45 (1): 1-9.

Di Lucca, G., et al. (2002) Testing web applications. *International Conference on Software Maintenance. (ICSM'02),* Firenze, Italy, 310-319.

Duran, J. and Ntafos, S. (1984) An evaluation of random testing. *IEEE Trans. On Software Engineering* 10(4): 438-443.

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

Gabrys, B.J., and Dick, J. (2001) An objectives-driven approach to testing web applications. from http://mcs.open.ac.uk/bjg7/objectivesdriven_final.pdf.

Gadgil, H.S. (2002) A paper on regression testing techniques.
from www.cs.indiana.edu/~hgadgil/work/projects/sem/bibliography.pdf.

Hamlet, G. and Taylor, R. (1990) Partition testing does not inspire confidence. *IEEE Trans. on Software Engineering*, December: 1402-1411.

Howden, W. (1986) A functional approach to program testing and analysis. *IEEE Trans. on Software Engineering,* 12(10): 997-1005.

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

Jeng, B. (1994) Integrating data flow and domain testing. *Asia-Pacific Software Engineering Conference*, Tokyo, Japan, pp. 123-135.

Jia, X., and Liu, H. (2002) Rigorous and automatic testing of web applications.
from http://venus.cs.depaul.edu/xjia/sea2002.pdf.

Kim, S., Clark, J., and McDermid, J. (2000) Class mutation: mutation testing for object-oriented programs. *Object-Oriented Software Systems, Net.ObjectDays'2000*, Netobjectdays Forum, Erfurt, Germany.

Korel, B. (1992) Dynamic method for software test data and generation. *Journal of Software Testing, Verification, and Reliability,* 9: 203-213.

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

Lyon-Smith, J. (2002) ASP.NET to Do List Application.*www.codeproject.com*.

Ma, Y., Kwon, Y-R, and Offutt, J. (2002) Inter-class mutation operators for Java. *In the Thirteenth International Symposium on Software reliability Engineering,* Annapolis, MD, pp. 352-366.

Mansour, N., and Salame, M. (2004) Data generation for path testing. *Software Quality Journal,* 12: 121-136.

Marx, D. I. S. and Frankl, P. G. (1999) Path-sensitive alias analysis for data flow testing. *Journal of Software Testing, Verification, and Reliability,* 9: 51-73.

Mathur, A. (1994) Mutation testing, *In Encyclopedia of Software Engineering,* J. Marciniak Ed., Wiley Interscience, pp. 707-712.

McCabe, T. (1982) *Structured Testing,* Washington, DC, US Government Printing Office.

Offutt, A. J., Jin, Z., and Pan, J. (1999) The dynamic domain reduction procedure for test data generation, *Software Practice and Experience,* 29(2): 167-193.

Offutt, R., et al. (2001) A fault model for subtype inheritance and polymorphism. *In the twelfth IEEE International Symposium on Software Reliability Engineering (ISSRE'01),* Hong Kong, pp. 84-95.

Powell, T.A. (1998) Web testing. *Web Site Engineering: Beyond Web Page Design,* N.J.: Prentice Hall, pp. 251-267

Ramamoorthly, C., Ho, S. and Chen, W. (1976) On the automated generation of program test data. *IEEE trans. On software engineering,* 2(4): 293-300.

Rapps, S. and Weyuker, E. (1985) Selecting software test data using data flow information. *IEEE trans. On software engineering,* April: 367-375.

Ricca, F., and Tonnella, P. (2001) Analysis and testing of web applications. *In the Proceedings of the International Conference on Software Engineering, (ICSE'2001),* Toronto, Canada, pp. 25-34.

Ricca, F., and Tonnella, P. (2001) Web application slicing. *In the Proceedings of the International Conference on Software Maintenance, (ICSM'2001),* Firenze, Italy, pp. 148-157.

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

Stocks, P.A. and Carrington, D. A. (1993) Test template framework: A specification-based testing case study. *International Conference of Software Engineering,* May: pp. 405-414.

Tai, K.-C., (1993) Predicate-based test generation for computer programs. *International Conference of Software Engineering*, pp. 267-276.

Wu, Y., and Offutt, J. (2002) Modeling and testing web-based applications, *GMU ISE Technical ISE-TR-02-08.*

Appendix A

In this appendix we list the complete source code for "To Do List" ASP.NET web application. Next, the complete control, data, call, event and system dependence graphs are illustrated.

A.1 Source Code for "To Do List"

Todolist .aspx:

```
1 <%@ Page language="c#" Codebehind="ToDoList.aspx.cs" AutoEventWireup="false" Inherits="ToDo.ToDoListForm" %>
2 <HTML>
3 	<HEAD>
4 		<title>
5 			<%=_title%>
6 		</title>
7 		<style type="text/css">
8 			H1 {FONT-SIZE: 12pt; LINE-HEIGHT: 2pt; FONT-FAMILY: Verdana}
9 			BODY {FONT-SIZE: 8pt; FONT-FAMILY: Verdana}
10			A {COLOR: navy}
11			A:visited {COLOR: navy}
12		</style>
13	</HEAD>
14	<body>
15		<h1><%=_title%></h1>
16		<form id="ToDoListForm" method="post" runat="server">
17			<A href="EditItem.aspx">Add New Item</A>  
18			<a href="ToDoList.aspx?query=0">Show All Items</a>  
19			<a href="ToDoList.aspx?query=1">Show All Closed Items</a>  
20			<a href="ToDoList.aspx?query=2">Show All Open Items</a>  
21			<P>
22						<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">
23					<Columns>
24						<asp:BoundColumn Visible="false" DataField="ID"/>
25						<asp:TemplateColumn ItemStyle-Width="12">
26								<ItemTemplate><img src='<%#
		_priorityUrls[(int)DataBinder.Eval(Container.DataItem, "Priority") - 1] %>'/></ItemTemplate>
27						</asp:TemplateColumn>
28						<asp:BoundColumn HeaderText="Description" DataField="Description" />
29					</Columns>
30				<HeaderStyle BackColor="teal" ForeColor="white" Font-Bold="true" />
31				<ItemStyle BackColor="white" ForeColor="darkblue" />
32				<AlternatingItemStyle BackColor="beige" ForeColor="darkblue" />
33				</asp:datagrid>
```

Todolist.aspx.cs:

```
33      public class ToDoListForm : System.Web.UI.Page
        {
34              protected System.Web.UI.WebControls.DataGrid ToDoDataGrid;
35              protected string _title;
36              protected string[] _priorityUrls = { "down.png", "nothing.png", "up.png" };
37              private void Page_Load(object sender, System.EventArgs e)
                {
38                      int query = 2;
39                      if (IsPostBack)
                        {
40                              query = (int)ViewState["query"];
                        }
41                      else
                        {
42                              string queryStr = Request.Params["query"];
43                              if (queryStr != null)
44                                      query = Int32.Parse(queryStr);
45
                                ViewState["query"] = query;
                        }

46                      string connStr = ConfigurationSettings.AppSettings["ConnectionString"];
47                      string sql;
48                      string qryTitle;
49                      ButtonColumn bcDone;
50                      ButtonColumn bcEdit;
51                      ButtonColumn bcDelete;
52                      ButtonColumn bcReopen;
53                      BoundColumn bcOpened;
54                      BoundColumn bcClosed;

55                      switch (query)
                        {
56                              case 0:
57                                      qryTitle = "All Items";
58                                      sql = "select * from items order by priority desc";
59                                      bcOpened = new BoundColumn();
60                                      bcOpened.HeaderText = "Opened";
61                                      bcOpened.DataField = "Opened";
62                                      ToDoDataGrid.Columns.Add(bcOpened);
                                        break;

63                              case 1:
64                                      qryTitle = "All Closed Items";
65                                      sql = "SELECT * FROM Items WHERE Closed Is Not Null order by priority desc";
66                                      bcClosed = new BoundColumn();
67                                      bcClosed.HeaderText = "Closed";
68                                      bcClosed.DataField = "Closed";
69                                      ToDoDataGrid.Columns.Add(bcClosed);
70                                      bcReopen = new ButtonColumn();
71                                      bcReopen.Text = "Reopen";
72                                      bcReopen.CommandName = "ReopenToDo";
73                                      ToDoDataGrid.Columns.Add(bcReopen);

74                                      bcDelete = new ButtonColumn();
75                                      bcDelete.Text = "Delete";
76                                      bcDelete.CommandName = "DeleteToDo";
77                                      ToDoDataGrid.Columns.Add(bcDelete);
                                        break;

78                              default:
79                              case 2:
80                                      qryTitle = "All Open Items";
81                                      sql = "SELECT * FROM Items WHERE Closed Is Null order by priority desc";
82                                      bcDone = new ButtonColumn();
```

```
83                              bcDone.Text = "Done";
84                              bcDone.CommandName = "DoneToDo";
85                              ToDoDataGrid.Columns.Add(bcDone);

86                              bcEdit = new ButtonColumn();
87                              bcEdit.Text = "Edit";
88                              bcEdit.CommandName = "EditToDo";
89                              ToDoDataGrid.Columns.Add(bcEdit);

90                              bcDelete = new ButtonColumn();
91                              bcDelete.Text = "Delete";
92                              bcDelete.CommandName = "DeleteToDo";
93                              ToDoDataGrid.Columns.Add(bcDelete);
                                break;
                }

94              _title = "To Do List - " + qryTitle;
95              OleDbDataAdapter adapter = new OleDbDataAdapter(sql, connStr);
96              DataSet ds = new DataSet();
97              adapter.Fill(ds);
98              ToDoDataGrid.DataSource = ds;
99              ToDoDataGrid.DataBind();
        }

100     public void ToDoDataGrid_Command(Object sender, DataGridCommandEventArgs e)
        {
101             TableCell idCell = e.Item.Cells[0];
102             string idStr = idCell.Text;
103             string cmdStr = ((LinkButton)e.CommandSource).CommandName;

104             if (cmdStr == "EditToDo")
                {
105                     Response.Redirect("EditItem.aspx?id=" + idStr);
                }

106             string connStr = ConfigurationSettings.AppSettings["ConnectionString"];
107             string sql;

108             switch (cmdStr)
                {
                        case "DeleteToDo":
109                             sql = "DELETE FROM Items WHERE ID=" + idStr;
                                break;

                        case "ReopenToDo":
111                             sql = "UPDATE Items SET Closed = Null WHERE ID=" + idStr;
                                break;

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

Edititem.aspx:

```
114<%@ Page language="c#" Codebehind="EditItem.aspx.cs" AutoEventWireup="false" Inherits="ToDo.EditItemForm" %>
115<HTML>
116	<HEAD>
117		<title>
118			<%=_title%>
119		</title>
120		<style type="text/css">
121			H1 { FONT-SIZE: 12pt; LINE-HEIGHT: 2pt; FONT-FAMILY: Verdana }
122			BODY { FONT-SIZE: 8pt; FONT-FAMILY: Verdana }
123			A { COLOR: blue }
124			A:visited { COLOR: blue }
125		</style>
126	</HEAD>
127	<body>
128		<script for="window" event="onload">
129			window.document.forms["EditItemForm"].children["DescriptionTextBox"].focus();
130		</script>
131		<h1><%=_title%></h1>
132		<form id="EditItemForm" method="post" runat="server">
133			Description:<br>
134			<asp:textbox id="DescriptionTextBox" runat="server" Font-Name="Verdana" Font-Size="8pt"
Width="100%"></asp:textbox>
135			<br>
136			Priority:<br>
137			<asp:dropdownlist id="PriorityList" Font-Name="Verdana" Font-Size="8pt"
Runat="server"></asp:dropdownlist><br>
138			<asp:label id="ErrorLabel" runat="server" Text="" Visible="False"
ForeColor="Red"></asp:label><br>
139			<asp:linkbutton id="SaveButton" onclick="SaveButton_Click" Text="Save Changes"
Runat="server"></asp:linkbutton></form>
140	</body>
141</HTML>
```

Edititem.aspx.cs:

```
142    public class EditItemForm : System.Web.UI.Page
       {
143        protected System.Web.UI.WebControls.LinkButton SaveButton;
144        protected System.Web.UI.WebControls.TextBox DescriptionTextBox;
145        protected System.Web.UI.WebControls.Label ErrorLabel;
146        protected System.Web.UI.WebControls.DropDownList PriorityList;

147        protected string _title;
148        protected System.Web.UI.WebControls.LinkButton Save;
149        protected static string[] _priorities = {"Low", "Medium", "High"};

150        private void Page_Load(object sender, System.EventArgs e)
           {
151            string idStr = Request.Params["id"];
152            _title = (idStr == null ? "New" : "Edit") + " To Do List Item";
153            if (!IsPostBack)
               {
154                foreach (string s in _priorities)
155                    PriorityList.Items.Add(s);

156                if (idStr != null)
                   {
157                    string connStr = ConfigurationSettings.AppSettings["ConnectionString"];
158                    string queryStr = "select * from Items where id=" + idStr;
159                    OleDbDataAdapter adapter = new OleDbDataAdapter(queryStr, connStr);
160                    DataSet ds = new DataSet();
161                    adapter.Fill(ds);
162                    DataTable tbl = ds.Tables[0];
163                    if (tbl.Rows.Count > 0)
                       {
164                        DataRow row = tbl.Rows[0];
165                        DescriptionTextBox.Text = row["Description"].ToString();
166                        PriorityList.SelectedIndex = (int)row["Priority"] - 1;
                       }
                   }

               }
167            else
               {
168                OnSubmit();
               }
           }
169        public void SaveButton_Click(object sender, System.EventArgs e)
           {
170            OnSubmit();
           }
171        protected void OnSubmit()
           {
172            string connStr = ConfigurationSettings.AppSettings["ConnectionString"];
173            string sql;
174            string idStr = Request.Params["id"];
175            string desc = DescriptionTextBox.Text.Replace("'", "''");
176            int priority = PriorityList.SelectedIndex + 1;

177            if (idStr == null)
178                sql = "INSERT INTO Items (Description, Priority) VALUES ('" + desc + "', " + priority + ")";
179            else
180                sql = "UPDATE Items SET Description = '" + desc + "', Priority=" +
                       priority + " WHERE ID=" + idStr;
```

A.2 Dependence Graphs

We illustrate in this section control, data, call, event and system dependence graphs for "To Do List".

A.2.1 Control Dependence Graphs

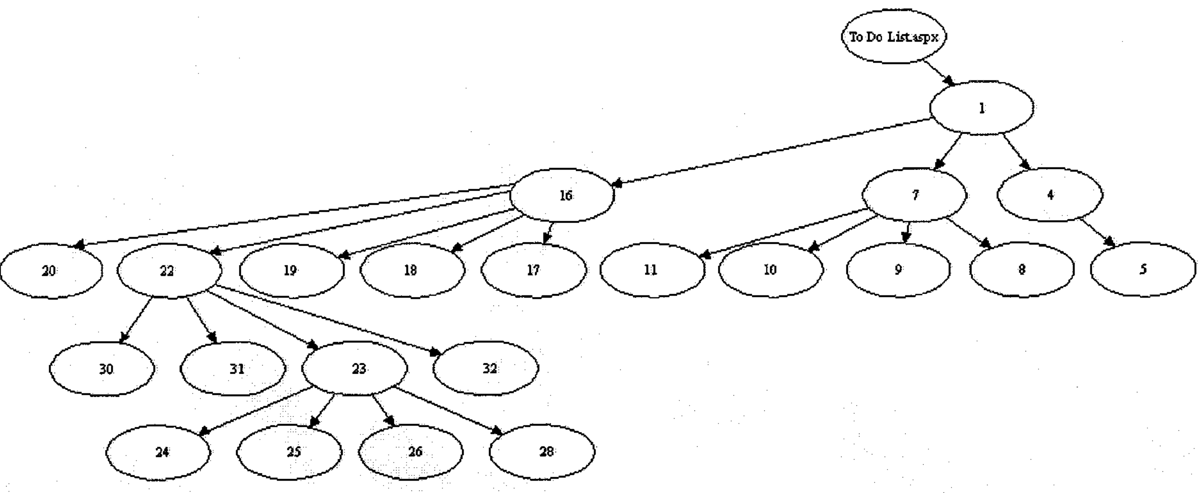


Figure A.1: Control Dependence Graph for Todolist.aspx

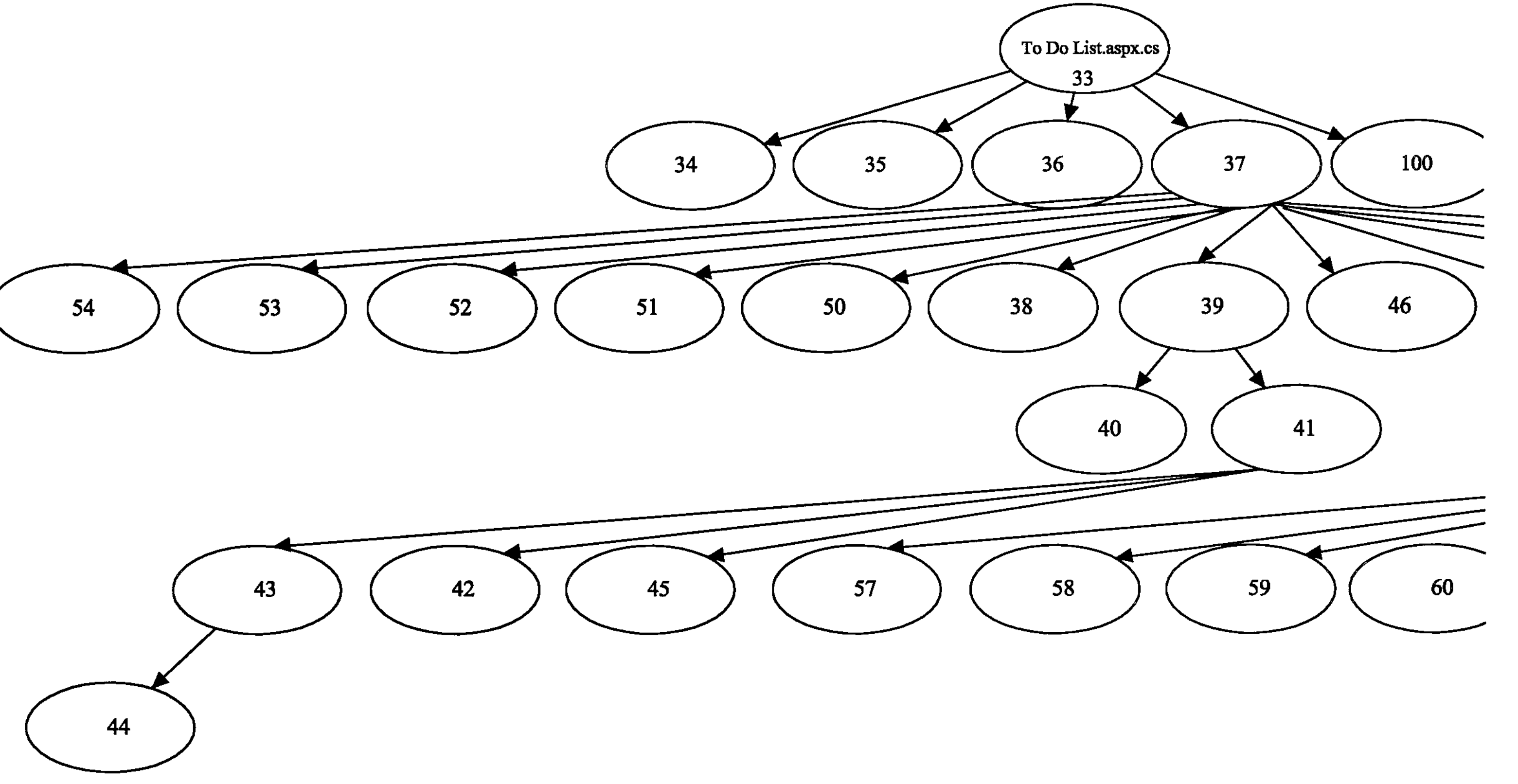
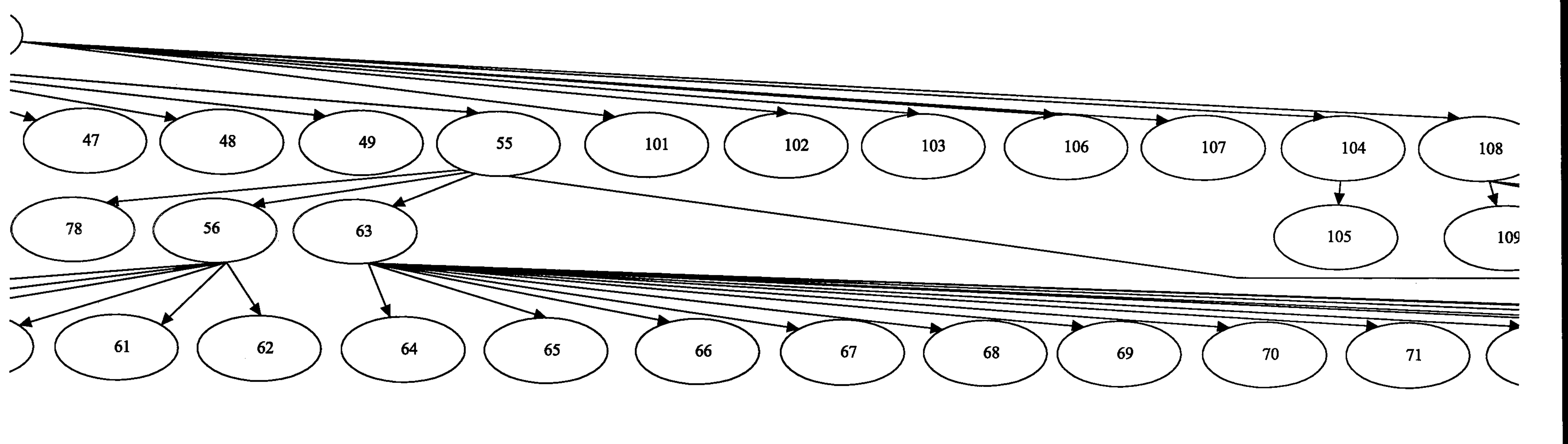
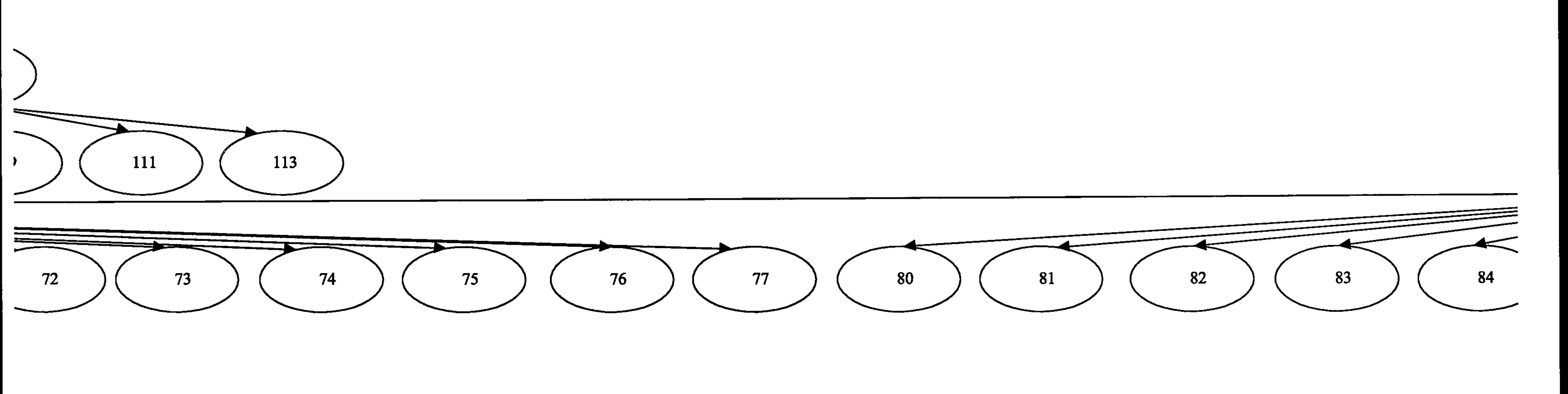
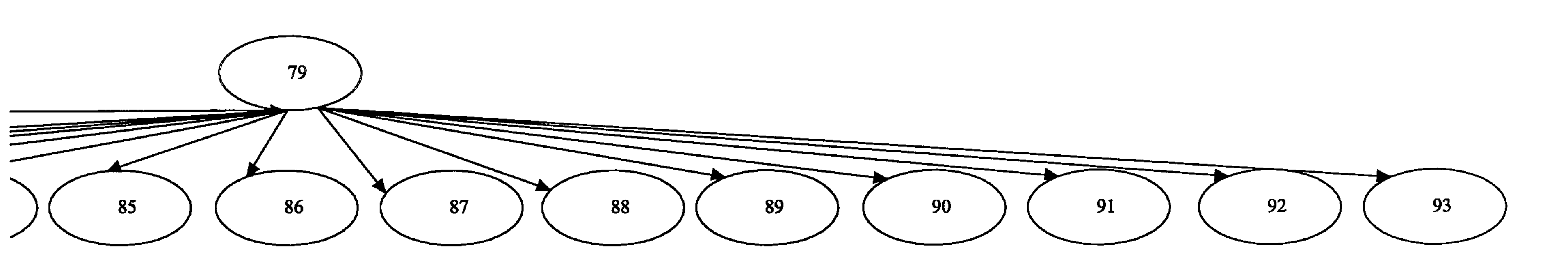


Figure A.2: Control Dependence Graph for To Do List.aspx.cs


47
48
49
55
101
102
103
106
107
104
108
78
56
63
105
109
61
62
64
65
66
67
68
69
70
71


111
113
72
73
74
75
76
77
80
81
82
83
84



79
85
86
87
88
89
90
91
92
93

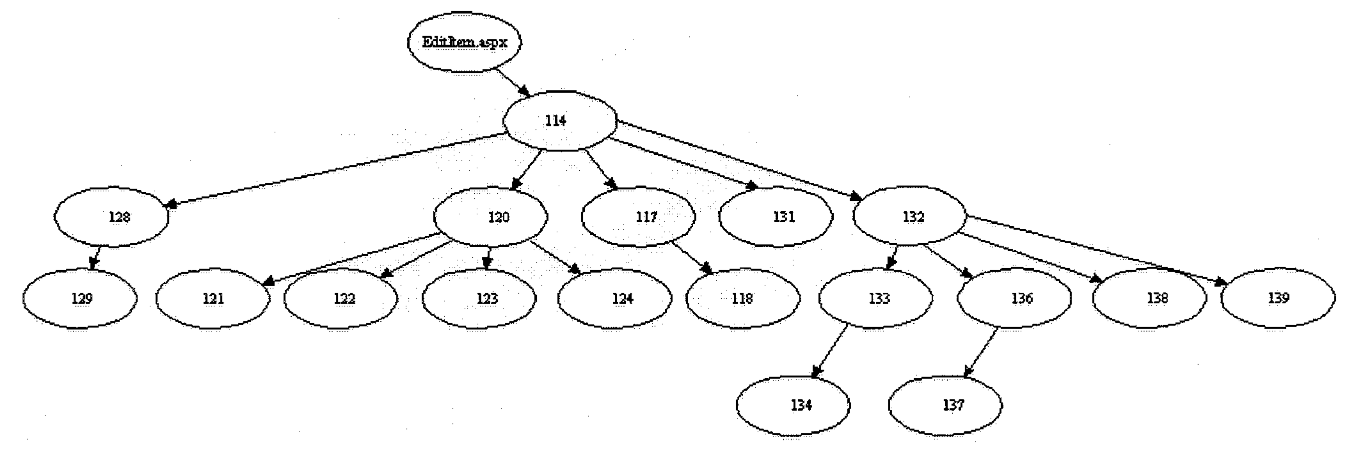


Figure A.3: Control Dependence Graph for Edititem.aspx.

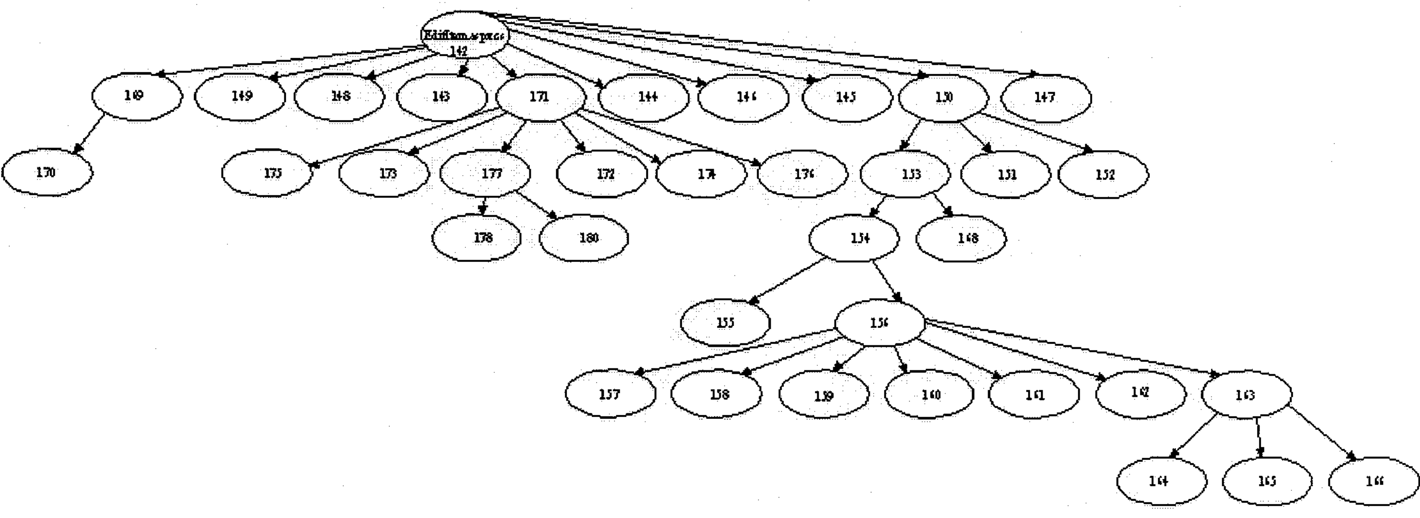


Figure A.4: Control Dependence Graph for Edititem.aspx.cs.

A.2.2 Data Dependence Graphs

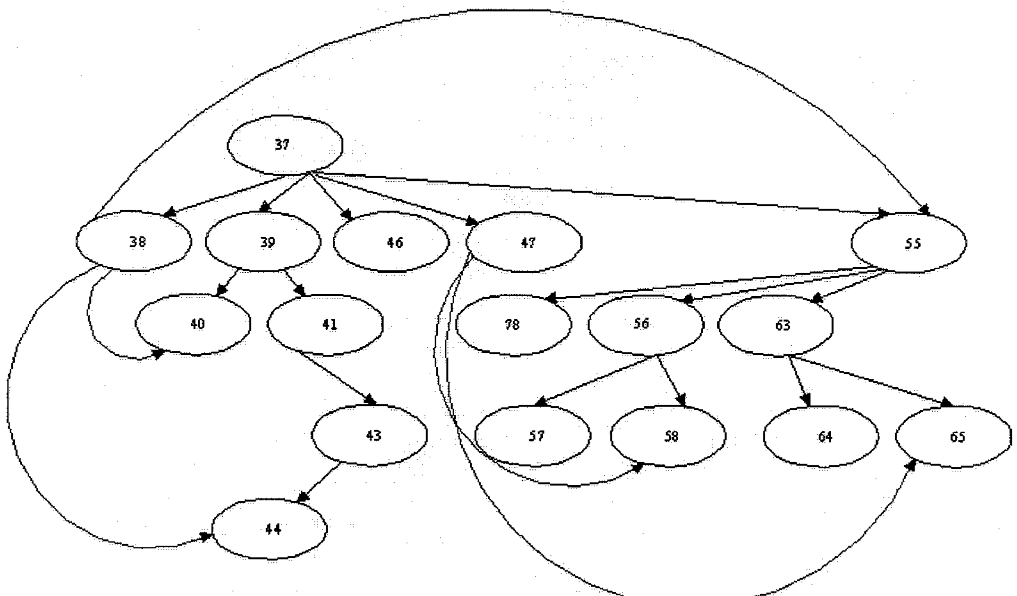


Figure A.5 (a): Data Dependence Graph for Todolist.aspx.cs.

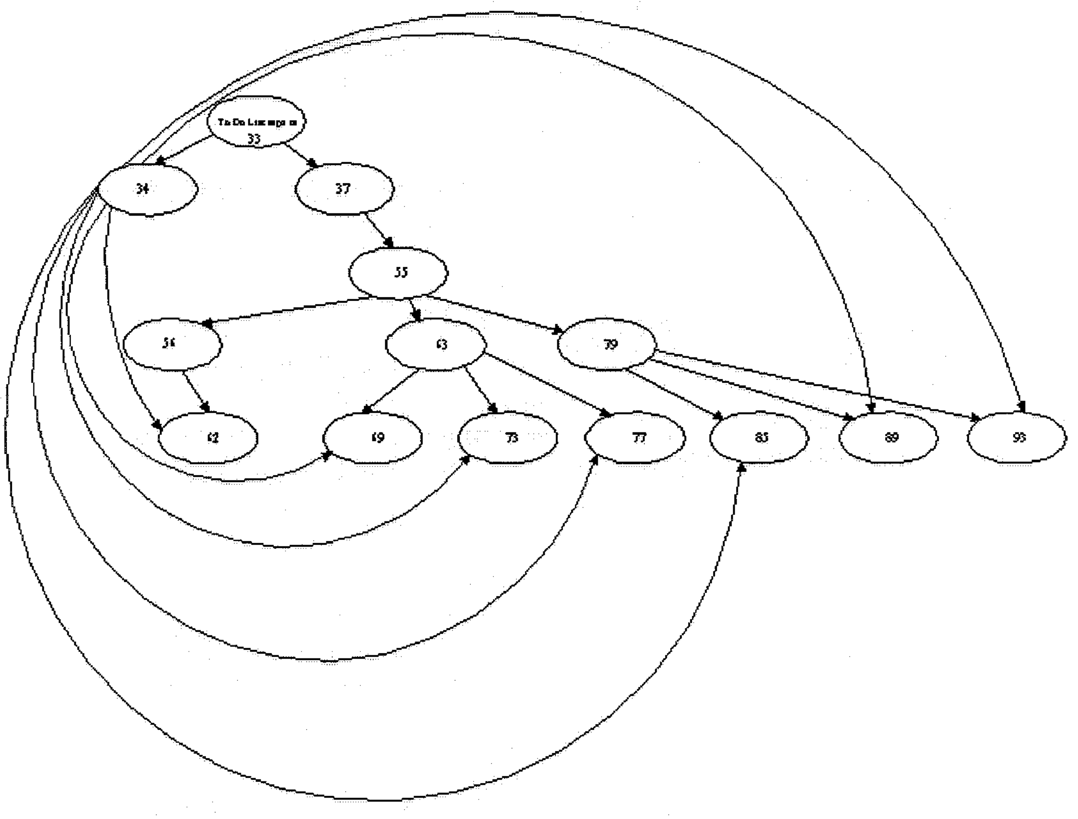


Figure A.5 (b): Data Dependence Graph for Todolist.aspx.cs.

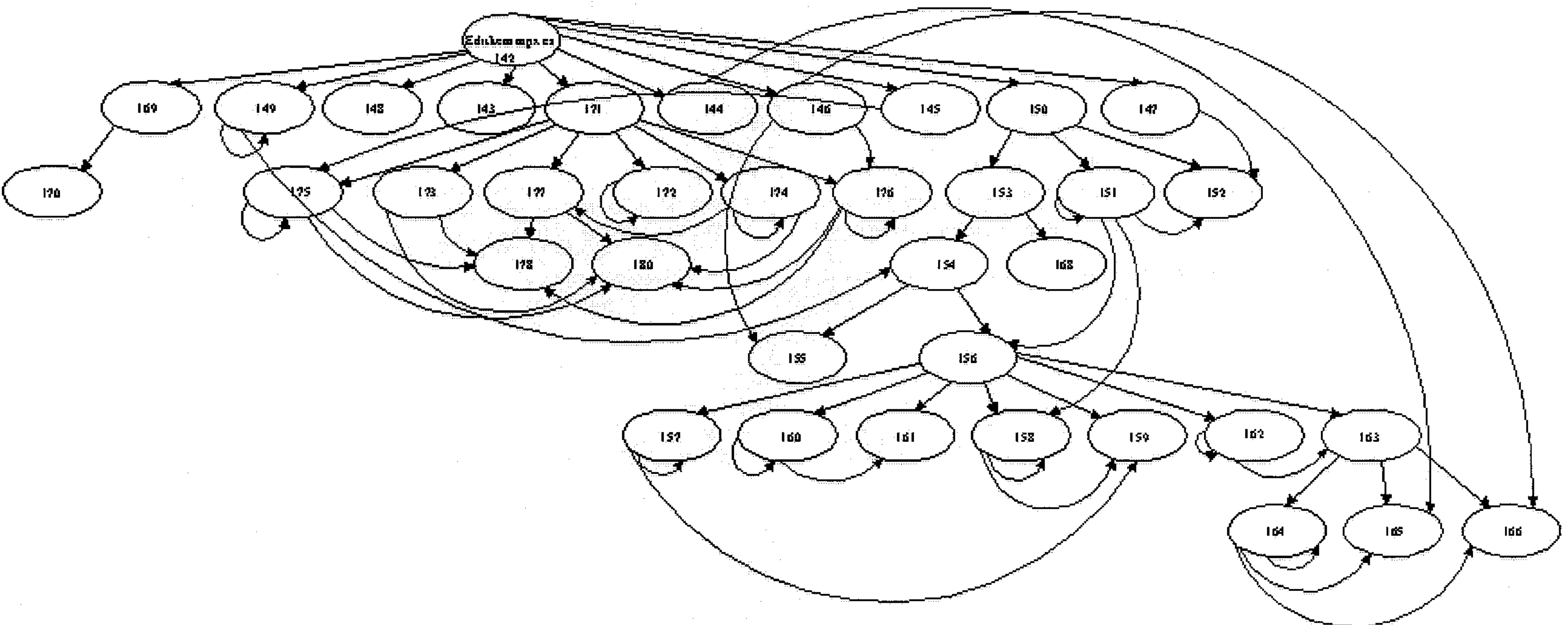


Figure A.6: Data Dependence Graph for Edititem.aspx.cs.

A.2.3 Call Dependence Graphs

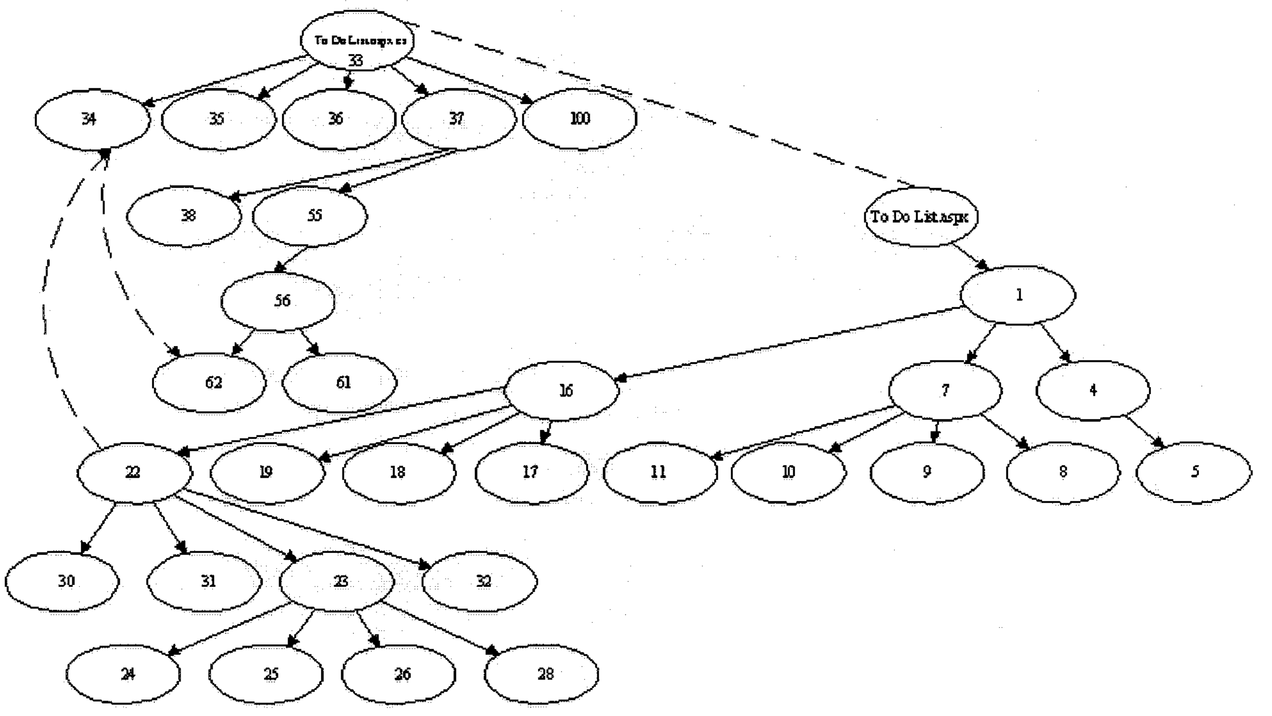


Figure A.7: Call Dependence Graph for Todolist.

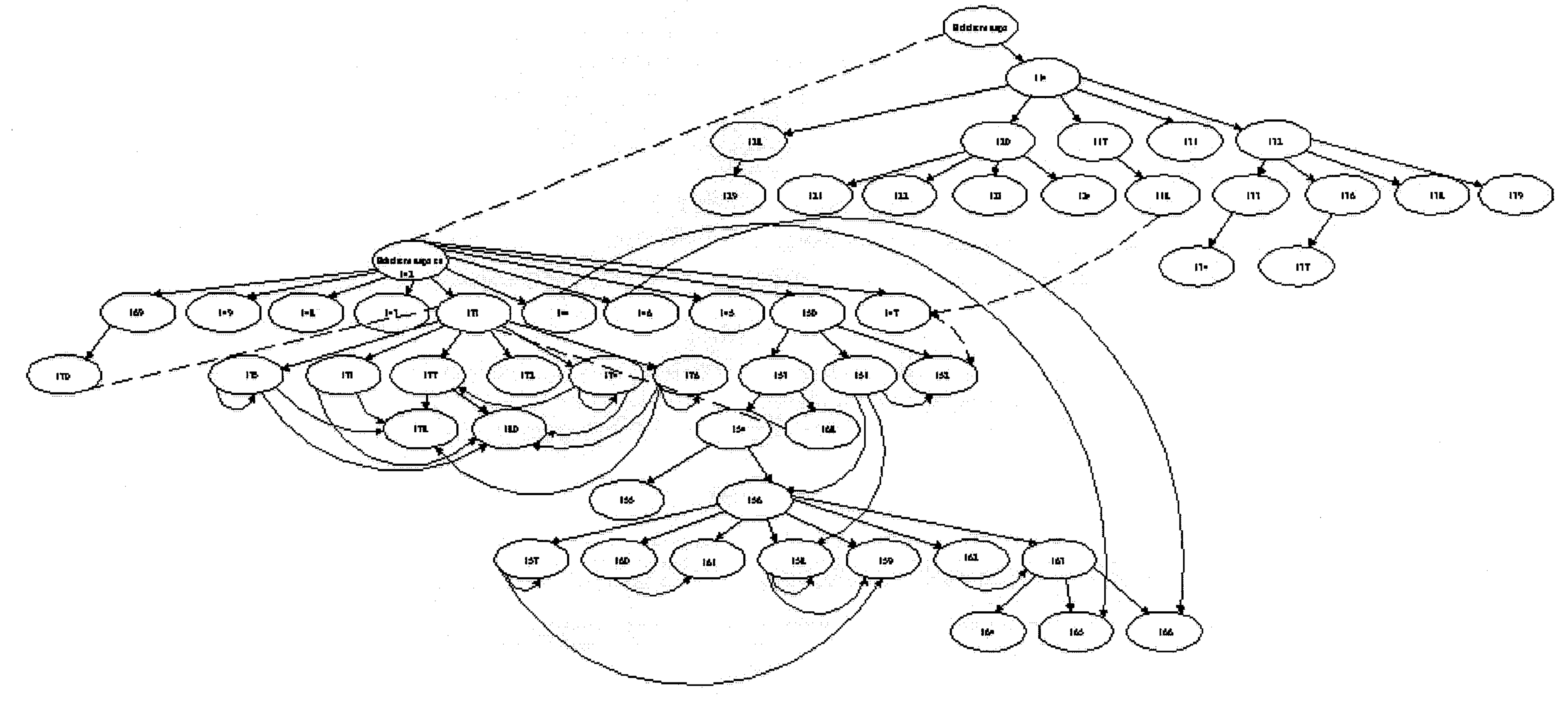


Figure A.8: Call Dependence Graph for Edititem.

A.2.4 Event Dependence Graph (EDG)

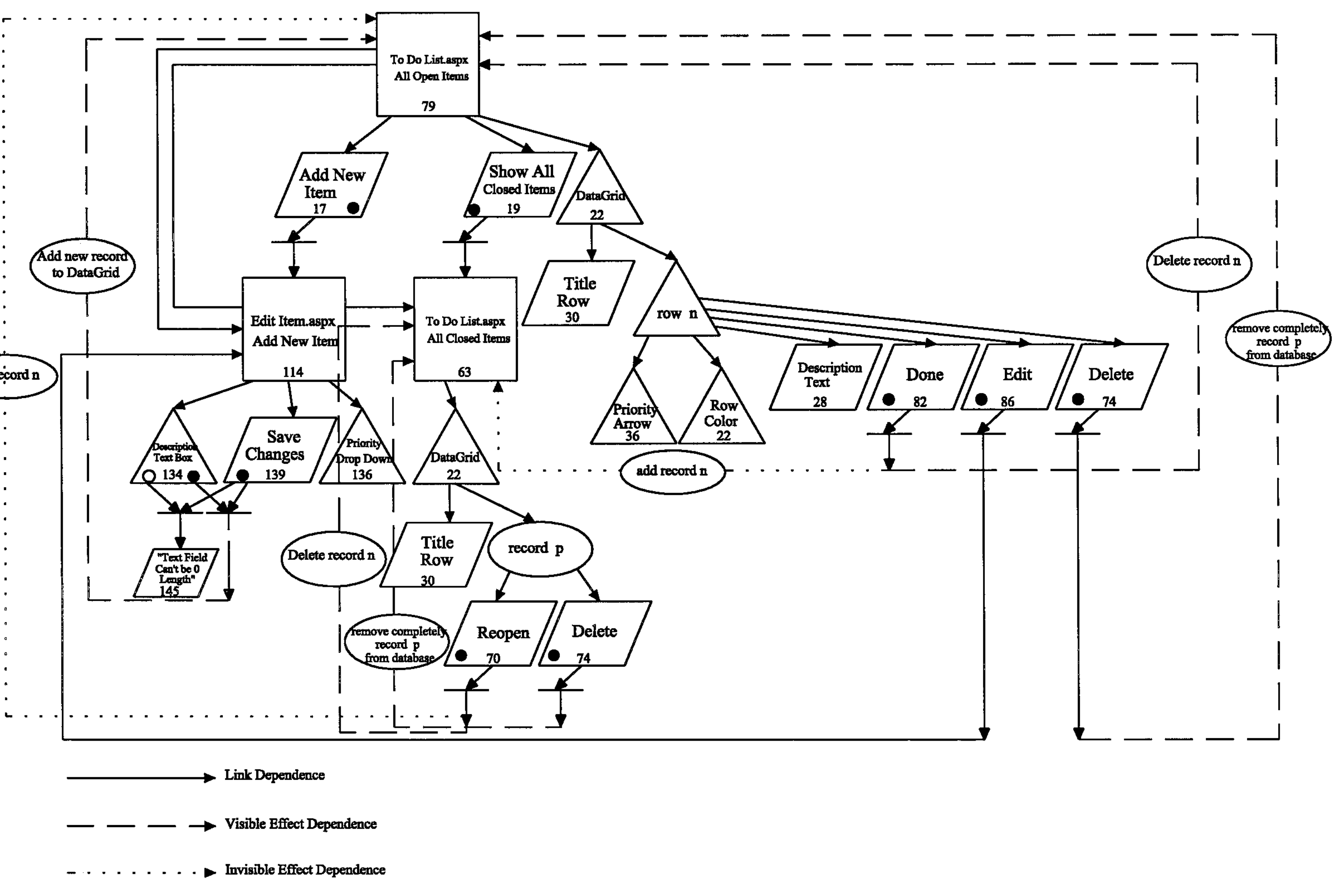


Figure A.9: Event Dependence Graph for To do list.

A.2.5 System Dependence Graph (SDG)

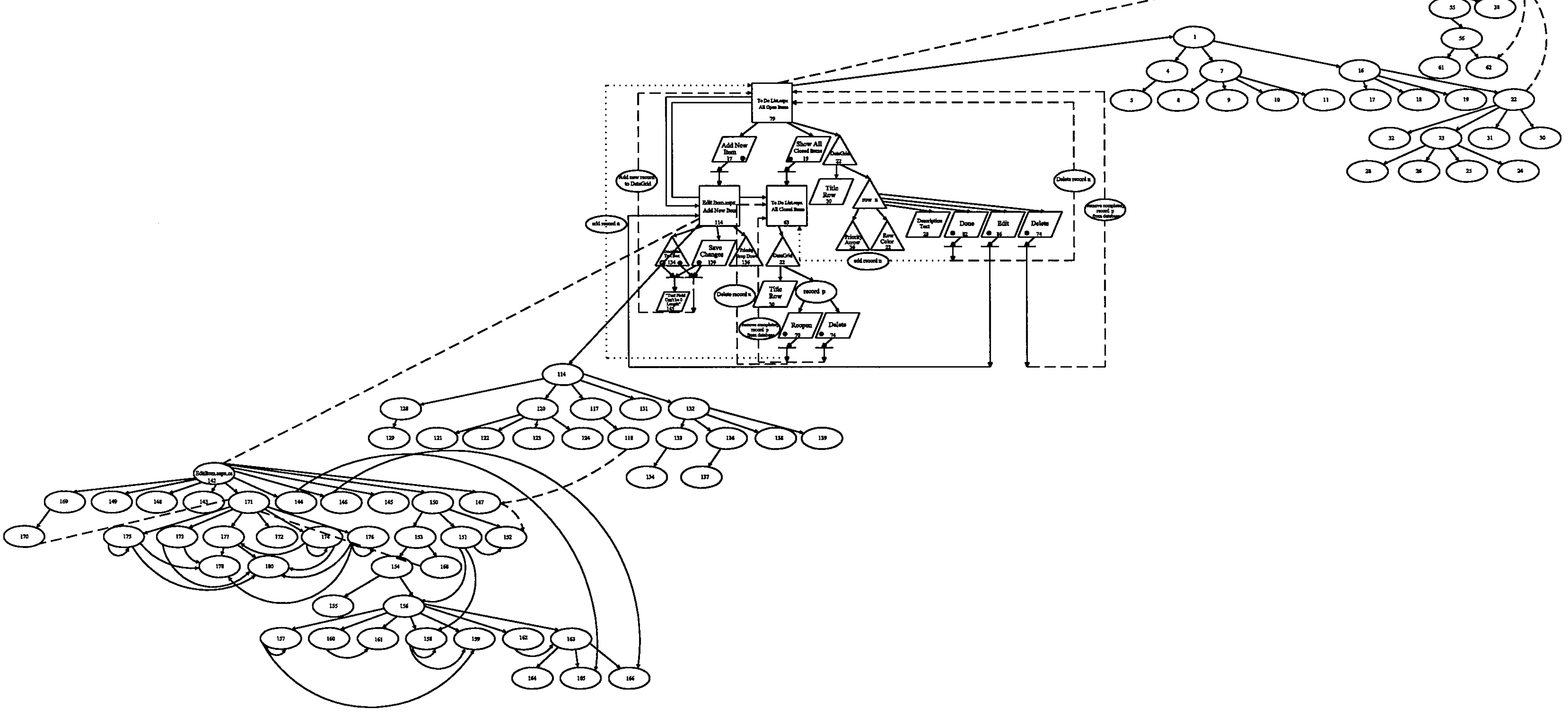


Figure A.10: System Dependence Graph for To do list.