COMPARATIVE STUDY ON CORRECTNESS AND TIME TAKEN OF TEST CASE GENERATION USING EFG AND BXT TECHNIQUES FOR GUI APPLICATION

MOSTAFA NSER BRKA

A dissertation submitted in partial fulfillment of the requirements for the award of the Degree of Master of Computer Science (Software Engineering)

Faculty of Computer Science and Information Technology
Universiti Tun Hussein Onn Malaysia

SEPTEMBER 2015
The previous decade witnessed the increased popularity and rapid development of graphical user interface (GUI). GUI is a type of computer–human interface that allows users to communicate with their computers. This interface has significantly contributed to the popularity of recently developed software applications. Given the importance of working under an error-free GUI, the correctness of such GUI must be tested at all times. GUI testing involves checking the screens for controls, such as menus, buttons, icons and all types of bars, including tool bars, menu bars, dialog boxes and software windows. Event Flow Graph (EFG) and Behavior ExplorerTesting (BXT) are two of the techniques used to generate test cases on GUI components. The correctness and time required for these two techniques to generate GUI test cases are compared in this project by using the techniques in Paint and Present applications. EFG and BXT obtained correctness rates of 9.61% and 36.81% respectively for the paint application and 5.18% and 39.33% respectively for the Present application. Therefore, BXT exhibits more correctness than EFG. In term of elapsed time, EFG and BXT spent 0.16 millisecond (ms) and 0.18 ms in the paint application respectively and 0.14 millisecond (ms) and 0.17 millisecond (ms) in the Present application respectively. Therefore, EFG is slightly faster than BXT in generating test cases. Overall, BXT is better than EFG in generating GUI test cases.
**ABSTRAK**

Dekad yang lepas menyaksikan peningkatan dalam populariti dan pembangunan yang pesat untuk antara muka pengguna grafik (GUI). GUI adalah antara muka manusia komputer yang membolehkan pengguna berkomunikasi dengan komputer mereka. Antara muka ini mempunyai sumbangan besar kepada populariti aplikasi perisian yang dibangunkan. Memandangkan pentingnya GUI bebas dari kesalahan, maka ketepatan kes GUI perlu sentiasa diuji. Pengujian GUI melibatkan pemeriksaan skrin untuk kawalan seperti menu, butang, ikon dan semua jenis bar, termasuk bar alat, bar menu, kotak dialog dan tetingkap perisian. Event Flow Graph (EFG) dan Behavior Explorer Testing (BXT) adalah antara teknik yang digunakan untuk menjana kes-kes ujian untuk komponen GUI. Dalam projek ini, ketepatan dan masa yang diperlukan untuk kedua-dua teknik menjana kes-kes ujian untuk komponen GUI dibandingkan dengan menggunakan dua kajian kes iaitu aplikasi mewarna dan aplikasi persembahan. Dari segi ketepatan, EFG dan BXT menunjukkan kadar ketepatan 9.61% dan 36.81% masing-masing untuk aplikasi mewarna. Sebaliknya, EFG dan BXT masing-masing menunjukkan kadar ketepatan 5.18% dan 39.33% dalam aplikasi persembahan. Oleh itu, teknik BXT menghasilkan kes ujian yang lebih tepat berbanding EFG. Dari segi masa, EFG dan BXT menggunakan 0.16 dan 0.18 millisaat untuk aplikasi mewarna dan 0.14 dan 0.17 millisaat untuk aplikasi persembahan. Oleh itu, EFG adalah sedikit lebih cepat daripada BXT dalam menjana kes-kes ujian. Namun begitu, secara keseluruhan, teknik BXT lebih baik daripada teknik EFG dalam menjana kes-kes ujian.
# CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>TITLE</td>
<td>i</td>
</tr>
<tr>
<td>DECLARATION</td>
<td>ii</td>
</tr>
<tr>
<td>DEDICATION</td>
<td>iii</td>
</tr>
<tr>
<td>ACKNOWLEDGEMENT</td>
<td>iv</td>
</tr>
<tr>
<td>ABSTRACT</td>
<td>v</td>
</tr>
<tr>
<td>ABSTRAK</td>
<td>vi</td>
</tr>
<tr>
<td>LIST OF TABLES</td>
<td>xi</td>
</tr>
<tr>
<td>LIST OF FIGURES</td>
<td>xiv</td>
</tr>
<tr>
<td>LIST OF SYMBOLS AND ABBREVIATIONS</td>
<td>xvii</td>
</tr>
<tr>
<td>LIST OF APPENDICES</td>
<td>xviii</td>
</tr>
</tbody>
</table>

## CHAPTER 1 INTRODUCTION

1. Background of Study
2. Problem Statement
3. Project Objectives
4. Scope of Project
5. Thesis Outline

## CHAPTER 2 LITERATURE REVIEW

1. Introduction
2. Graphical User Interface
3. Software Testing
4. GUI Testing
5. Performance Parameters for Testing
   1. Complexity
   2. Correctness
   3. Magnitude of Relative Error (MRE) for Parameters
4.1 Introduction 36
4.2 Pre-work before applying the techniques 36
  4.2.1 First case study (Paint application) 37
  4.2.2 Second case study (Present application) 38
4.2.3 Generate Test Cases For EFG and BXT Technique in Case Studies 39
4.3 Implementation of the EFG and BXT techniques in case study 1 39
  4.3.1 Implementation of the EFG technique in case study 1 39
  4.3.2 Implementation the BXT technique in case study 1 47
4.4 Implementation EFG and BXT techniques in case study 2 50
  4.4.1 Implementation EFG technique in case study 2 50
  4.4.2 Implementation BXT technique in case study 2 59
4.5 Step 2: Calculate complexity and time taken using EFG and BXT technique in case studies 62
  4.5.1 Step 2.1: Calculate the complexity and time taken using EFG technique in case study 1 62
  4.5.2 Step 2.2: Calculate complexity and time taken using BXT technique in case study 1 64
  4.5.3 Step 2.3: Calculate complexity and time taken using EFG technique in case study 2 67
  4.5.4 Step 2.4: Calculate complexity and time taken using BXT technique in case study 2 69
4.6 Summary 71

CHAPTER 5  RESULTS AND ANALYSIS 72
5.1 Introduction 72
5.2 Magnitude of relative error (MRE) 72
5.3 Analysis of the result in two case studies 72
  5.3.1 Analysis of the results in Paint application 73
  5.3.2 Discussion the results to Case study 1 78
  5.3.3 Analysis of the Results in Present application 79
  5.3.4 Discuss the results to case study 2 84
5.4 Summary 85
# LIST OF TABLES

3.1 Description of terms in algorithm for event flow graph 26
3.2 Description of terms in algorithm for the execute test cases generation 27
3.3 Description of terms in algorithm for testing case execution 29
3.4 Description of terms used in algorithm for BXT 32
3.5 Example date for Complexity and Time Taken 34
4.1 Experimental results for File Menu using EFG technique in Paint application 62
4.2 Experimental results for Edit Menu using EFG technique in Paint application 63
4.3 Experimental results for View Menu using EFG technique in Paint application 63
4.4 Experimental result for Image Menu using EFG technique in Paint application 63
4.5 Experimental results for Filter Menu using EFG technique in Paint application 64
4.6 Experimental results for layer Menu using EFG technique in Paint application 64
4.7 Experimental results for File Menu using BXT technique in Paint application 65
4.8 Experimental results for Edit Menu using BXT technique in Paint application 65
4.9 Experimental results for View Menu using BXT technique in Paint application 65
4.10 Experimental results for Image Menu using BXT technique in Paint application 66
4.11 Experimental results for Filter Menu using BXT technique in 66
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.12</td>
<td>Experimental results for layer Menu using BXT technique in Paint application</td>
<td>66</td>
</tr>
<tr>
<td>4.13</td>
<td>Experimental results for File Menu using EFG technique in Present application</td>
<td>67</td>
</tr>
<tr>
<td>4.14</td>
<td>Experimental results for Edit Menu using EFG technique in Present application</td>
<td>67</td>
</tr>
<tr>
<td>4.15</td>
<td>Experimental result for View Menu using EFG technique in Present application</td>
<td>68</td>
</tr>
<tr>
<td>4.16</td>
<td>Experimental results for Image Menu using EFG technique in Present application</td>
<td>68</td>
</tr>
<tr>
<td>4.17</td>
<td>Experimental results for Filter Menu using EFG technique in Present application</td>
<td>68</td>
</tr>
<tr>
<td>4.18</td>
<td>Experimental results for layer Menu using EFG technique in Present application</td>
<td>69</td>
</tr>
<tr>
<td>4.19</td>
<td>Experimental results for File Menu using BXT technique in Present application</td>
<td>69</td>
</tr>
<tr>
<td>4.20</td>
<td>Experimental results for Edit Menu using BXT technique in Present application</td>
<td>70</td>
</tr>
<tr>
<td>4.21</td>
<td>Experimental results for View Menu using BXT technique in Present application</td>
<td>70</td>
</tr>
<tr>
<td>4.22</td>
<td>Experimental results for Image Menu using BXT technique in Present application</td>
<td>70</td>
</tr>
<tr>
<td>4.23</td>
<td>Experimental results for Filter Menu using BXT technique in Present application</td>
<td>71</td>
</tr>
<tr>
<td>4.24</td>
<td>Experimental results for Layer Menu using BXT technique in Present application</td>
<td>71</td>
</tr>
<tr>
<td>5.1</td>
<td>Experimental results in term of Correctness using EFG technique in Paint application</td>
<td>73</td>
</tr>
<tr>
<td>5.2</td>
<td>Experimental results in terms of Correctness using BXT technique Paint application</td>
<td>74</td>
</tr>
<tr>
<td>5.3</td>
<td>Experimental results in terms of Time Taken using EFG techniques in Paint application</td>
<td>75</td>
</tr>
</tbody>
</table>
5.4 Experimental results in terms of Time Taken using BXT techniques Paint application
5.5 Criteria in Paint application for Correctness
5.6 MRE Criteria in Paint application
5.7 Experimental results in terms of Correctness using EFG technique in Present application
5.8 Experimental results in terms of Correctness using BXT technique in Present application
5.9 Experimental results in terms of Time Taken using EFG technique in Paint application
5.10 Experimental results in terms of Time Taken using BXT technique in Paint application
5.11 Criteria in Present application for Correctness
5.12 MRE Criteria in Present application
### LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>GUI is the front-end to the underlying code</td>
<td>6</td>
</tr>
<tr>
<td>2.2</td>
<td>Simple GUI in EFG</td>
<td>15</td>
</tr>
<tr>
<td>2.3</td>
<td>Event sequence in GUI</td>
<td>16</td>
</tr>
<tr>
<td>2.4</td>
<td>Algorithm 1: EFG</td>
<td>16</td>
</tr>
<tr>
<td>2.5</td>
<td>Algorithm 2 shows the pseudo-code for BXT</td>
<td>18</td>
</tr>
<tr>
<td>3.1</td>
<td>Schematic diagram of the research methodology</td>
<td>23</td>
</tr>
<tr>
<td>3.2</td>
<td>Details of applying EFG technique (Step 1-1)</td>
<td>24</td>
</tr>
<tr>
<td>3.3</td>
<td>Model of generating EFG</td>
<td>25</td>
</tr>
<tr>
<td>3.4</td>
<td>Algorithm 2: Execute Test Cases Generation</td>
<td>27</td>
</tr>
<tr>
<td>3.5</td>
<td>Algorithm 3: Execution Test Case Generation</td>
<td>28</td>
</tr>
<tr>
<td>3.6</td>
<td>Step 1-2 Application of the BXT technique</td>
<td>30</td>
</tr>
<tr>
<td>3.7</td>
<td>BXT Algorithm</td>
<td>31</td>
</tr>
<tr>
<td>4.1</td>
<td>Case studies in the application under test (AUT) folder</td>
<td>36</td>
</tr>
<tr>
<td>4.2</td>
<td>Main interface for Paint application</td>
<td>37</td>
</tr>
<tr>
<td>4.3</td>
<td>Main interface for Present application</td>
<td>38</td>
</tr>
<tr>
<td>4.4</td>
<td>Import Paint application and GUI Tester tool to Eclipse</td>
<td>39</td>
</tr>
<tr>
<td>4.5</td>
<td>Code for creating the graph module in EFG technique</td>
<td>40</td>
</tr>
<tr>
<td>4.6</td>
<td>Code to save the graph module</td>
<td>40</td>
</tr>
<tr>
<td>4.7</td>
<td>Code for export graph to the graph module</td>
<td>41</td>
</tr>
<tr>
<td>4.8</td>
<td>Graph model for components in the Paint application</td>
<td>41</td>
</tr>
<tr>
<td>4.9</td>
<td>Code segment for the load graph module</td>
<td>42</td>
</tr>
<tr>
<td>4.10</td>
<td>GUI Tester tool creating and saving test case</td>
<td>42</td>
</tr>
<tr>
<td>4.11</td>
<td>Generating and saving test case in Paint application</td>
<td>43</td>
</tr>
<tr>
<td>4.12</td>
<td>GUI Tester tool save the test cases in result folder</td>
<td>43</td>
</tr>
<tr>
<td>4.13</td>
<td>Selection of the test case index</td>
<td>43</td>
</tr>
<tr>
<td>4.14</td>
<td>Selection and execution of some of the test cases</td>
<td>44</td>
</tr>
</tbody>
</table>
4.15 XML report is saved by the GUI tester Tool upon execution 44
4.16 Information In XML after the source code is executed 45
4.17 Determination of the location of test cases in the Paint application 45
4.18 Information in the XML file after execution 46
4.19 Result from an XML Report 46
4.20 Importing the Paint application, a GUI Testr tool and BXT 47
4.21 Debugging the GUI Tester tool and running the BXT technique 48
4.22 Information in XML before the source code is executed 48
4.23 Information in XML after the source code is executed 49
4.24 Code to show the result of XML report after execution 49
4.25 Result from an XML report after execution 50
4.26 Importing Present application and GUI Tester tool to Eclipse 51
4.27 Code for creating the graph module in EFG technique 51
4.28 Code for saving the graph module 52
4.29 Code to export graph to graph module 52
4.30 Graph model for the components of the Present application 53
4.31 Code segment to load graph module 54
4.32 GUI Tester tool creating and saving test cases 54
4.33 Generating and saving the test case 55
4.34 GUI tester tool saves the test cases in the result folder 55
4.35 Selection of the test case index 56
4.36 Selection and execution of some of the test cases 56
4.37 GUI Tester tool saves the XML report after execution 57
4.38 Information in the XML after executing the source code 57
4.39 Determination of the location of test cases 58
4.40 Code for showing the result of the XML report before execution 58
4.41 Result from an XML report from the Present application 58
4.42 Importing the Present application, GUI Tester tool and BXT technique 59
4.43 Debugging the GUI Tester tool and running the BXT technique 60
4.44 Result after the BXT technique is executed in the GUI Tester tool 60
4.45 Information in XML after execution in the source code 61
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.46</td>
<td>Code for executing and viewing the result from the XML report</td>
<td>61</td>
</tr>
<tr>
<td>4.47</td>
<td>Result from an XML report after execution</td>
<td>62</td>
</tr>
<tr>
<td>5.1</td>
<td>Average correctness in Paint application</td>
<td>78</td>
</tr>
<tr>
<td>5.2</td>
<td>Average time taken in Paint application</td>
<td>79</td>
</tr>
<tr>
<td>5.3</td>
<td>Average correctness in Present application</td>
<td>84</td>
</tr>
<tr>
<td>5.4</td>
<td>Average time taken in Present application</td>
<td>85</td>
</tr>
<tr>
<td>Symbol</td>
<td>Definition</td>
<td></td>
</tr>
<tr>
<td>--------</td>
<td>------------------------------------------------</td>
<td></td>
</tr>
<tr>
<td>AUT</td>
<td>Application Under Testing</td>
<td></td>
</tr>
<tr>
<td>BXT</td>
<td>Behavior Explore Testing</td>
<td></td>
</tr>
<tr>
<td>CPU</td>
<td>Center Processor Unit</td>
<td></td>
</tr>
<tr>
<td>CLIs</td>
<td>Command-Line Interfaces</td>
<td></td>
</tr>
<tr>
<td>DOS</td>
<td>Disk Operating System</td>
<td></td>
</tr>
<tr>
<td>DOM</td>
<td>Document Object Model</td>
<td></td>
</tr>
<tr>
<td>ESI</td>
<td>Event Semantic Interaction Graphs</td>
<td></td>
</tr>
<tr>
<td>G</td>
<td>Event Flow Graphs</td>
<td></td>
</tr>
<tr>
<td>EIG</td>
<td>Event Interaction Graphs</td>
<td></td>
</tr>
<tr>
<td>GUI</td>
<td>Graphical User Interface</td>
<td></td>
</tr>
<tr>
<td>IDE</td>
<td>Integrated Development Environment</td>
<td></td>
</tr>
<tr>
<td>JDT</td>
<td>Java Development Tools</td>
<td></td>
</tr>
<tr>
<td>MRE</td>
<td>Magnitude of Relative Error</td>
<td></td>
</tr>
<tr>
<td>Ms</td>
<td>Millisecond</td>
<td></td>
</tr>
<tr>
<td>SDLC</td>
<td>Software Development Lifecycle</td>
<td></td>
</tr>
<tr>
<td>STL</td>
<td>Software Testing Lifecycle</td>
<td></td>
</tr>
<tr>
<td>SFG</td>
<td>State Flow Graph</td>
<td></td>
</tr>
<tr>
<td>STS</td>
<td>Seed Test Suite</td>
<td></td>
</tr>
</tbody>
</table>
**LIST OF APPENDICES**

<table>
<thead>
<tr>
<th>APPENDIX</th>
<th>TITLE</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Source code for case studies</td>
<td>93</td>
</tr>
<tr>
<td>B</td>
<td>Test case description for both case studies</td>
<td>136</td>
</tr>
<tr>
<td>C</td>
<td>Import case studies to Eclipse framework</td>
<td>139</td>
</tr>
<tr>
<td>D</td>
<td>Source code for first algorithm EFG ripper strategy</td>
<td>141</td>
</tr>
<tr>
<td>E</td>
<td>Source code for second algorithm EFG test cases generation</td>
<td>151</td>
</tr>
<tr>
<td>F</td>
<td>GUI tester properties</td>
<td>155</td>
</tr>
<tr>
<td>G</td>
<td>Source code for EFG based test case execution strategy</td>
<td>156</td>
</tr>
<tr>
<td>H</td>
<td>Code use to for show the results after execute test cases</td>
<td>169</td>
</tr>
<tr>
<td>I</td>
<td>Source code for BXT based test cases generation</td>
<td>174</td>
</tr>
</tbody>
</table>
CHAPTER 1

INTRODUCTION

1.1 Background of Study

The Palo Alto Research Center of the Xerox Corporation designed the first graphical user interface (GUI) in the 1970s. However, this GUI was not as popular as the GUIs launched by Apple Macintosh in the 1980s. The substantial power consumption of the central processing unit, high-quality graphics, and high cost of GUI slowed down its growth. Since 1997, Xerox Star (Aris, 2007) has developed and expanded the use of GUIs.

A GUI is sometimes referred to as “gee-you-eye” or “gooey” in computing. It facilitates user interaction with electronic devices by using graphical icons and visual indicators, such as secondary notation or visual cues, as opposed to text-based interfaces, keywords, text links, and text navigation designs (Martinez, 2011). GUIs sharply reduce the use of command line interfaces (CLIs) that use the keyboard to type commands (Melchior et al., 2009). The disk operating system (DOS) is an example of the a typical user–computer interface that used a keyboard’s typed commands before GUIs were designed (Nadira & Sani, 2009). The intermediate step in user interfaces between GUI and CLIs was the non-graphical, menu-based interface, where a user interacts with a device using a mouse instead of typing keyboard commands. GUIs are among the crucial parts of a software (Xie et al., 2006).

The design of human–computer interaction, which is the application of programming in software technology, depends on the visual arrangement and temporal behavior of a GUI. The GUI enhances the efficiency and ease of the
underlying logical design of a stored program, a design discipline known as usability. In a user-centered design system, the usage of visual language ensures the efficiency and usability of tasks. Thus, a good user interface depends on the design instead of the system architecture (Shneiderman & Ben, 2003).

In software engineering, testing all the components of GUIs is vital to ensure that the GUI meets written specifications. Using a variety of test cases ensures that the GUI design specifications are fulfilled. Nowadays, software testing is an important stage in software projects (Nah et al., 2001). It is one of the most expensive and time-consuming phases and usually stops after available resources are used or even in the middle of the development process because of the duration of this phase. However, manual testing allows a programmer to pledge each test, interrelate with the test, as well as interpret, investigate and report the collected results.

Software testing is automated with a tester-free mechanism. GUI is used to design various software applications because it interacts directly with users. Thus, the accuracy of the applications can easily meet the quality specification set by users. Other tools, such as capture replay, can be laborious and error-prone in designing software applications. However, GUI is particularly automated for easy management. Hence, substantial research focused on different methodologies to test GUIs, particularly the techniques for automated GUI test case generation (Padmawar & Sarwate, 2014).

Event flow graphs (EFGs) and behavior explorer technique (BXT) are two of the most employed techniques to generate test cases for GUI. All possible event flows within a particular system are provided by EFGs. These graphs are structural in nature, which is a limitation of the subpaths within an event that can increase exponentially. This step is theoretically feasible but is limited to practical applications. The second method called BXT determines the effect of the first event on the subsequent event and is useful in selecting two-way interaction. Hence, this research applies these techniques to test GUIs in the two case studies.
1.2 Problem Statement

GUI is designed to improve the interaction between a user and an electronic system. It involves the use of a mouse to select menu options; decisions are made by clicking screen buttons and programs are launched by clicking icons on screen. GUI must always be error-free and normal users must not experience any difficulty when using this interface. Numerous software applications depend on GUIs to coordinate the interactions of users with their systems. However, testing the correctness of a GUI is difficult because of the numerous possible interactions in the interface, such as the command button, text box, combo box and radio button. The sequence of GUI events can also lead to different states, increase the cost and length of software development and consume more time (Giuseppe & Di, 2012). A test case includes the input and expected output, pass/fail criteria and environment where the test will be conducted. Input refers to the data required to generate a test case. Software products can be tested in two ways. First, tests are performed on each function of a product to determine whether a software is fully operational. Second, the internal mechanisms of a product are tested to determine whether these functions actually occur. Many techniques have been used to generate test cases for GUI applications, including Event Interaction Graph (EIG) (Memon et al., 2005), Event Semantic Interaction Graph (ESIG) (Yuan & Memon 2010), EFG (Memon et al., 2012) and BXT (Bertolini et al., 2009). These techniques improve, the correctness and consumed time of test cases. Based on the software engineering perspective, the correctness refers to the adherence of a GUI to specifications that determine how users can interact with software and how the software must behave when used correctly. If the software behaves incorrectly, then users may spend a considerable amount of time to complete their tasks or may even fail to complete such tasks. Thus, the current research will compare the correctness and time taken of the two techniques, namely, EFG and BXT, in generating test cases.
1.3 Project Objectives

The objectives of this research are as follows:
1. To define steps for generating test cases by using Event Flow Graphs (EFG) and Behavior Explore Testing (BXT) techniques.
2. To apply techniques in (1) to the paint application and Present application.
3. To compare the correctness and time taken of test case generated between both techniques in (1) for the case studies in (2).

1.4 Scope of Project

This research uses two case studies and two techniques to investigate the problem of correctness and time taken in GUI applications. The study compares the result of the two techniques for the two case studies to determine the better technique between the two chosen techniques. The two techniques are EFG and BXT. The case studies of the current research are the paint and Present applications.

1.5 Dissertation Outline

The dissertation includes six chapters. Chapter 1 is an overview of the research and provides the main objectives of the project. The chapter also includes the problem statement and scope of the work covered by this project. Chapter 2 provides the literature review of EFG and BXT, as well as a brief explanation of the general information about generating test cases for GUI applications and some definitions used in this project. Chapter 3 discusses the methodology and tools to achieve all the objectives of this project. Chapter 4 explains the implementation and detailed steps employed in this work. Chapter 5 discusses this project. Finally, Chapter 6 explains the achieved objectives, conclusion of the project and future work.
CHAPTER 2

LITERATURE REVIEW

2.1 Introduction

This chapter provides an overview of previous research on GUI testing. Research on techniques for generating test cases are also presented in this chapter. An introduction of a framework for tools that employ these techniques and some topics related to this study are explained to identify the appropriate approach for investigating the objectives of the project.

2.2 Graphical User Interface

GUIs have three main bases: windows applications, web applications and mobile applications. At present, software GUIs are one of the most commonly used components. A typical GUI provides degrees of freedom and various facilities to an end-user. A test designer handles particular design challenges, such as enormous input interaction space of the GUI, deals with development and examines the test cases (Huang & Lu, 2012).

Currently, almost 70% of software systems are developed using GUIs. GUIs are primarily used to facilitate an end-user to promote the features of GUIs that provide ease and natural interaction between a system and its users.

GUI is hierarchical in producing the deterministic graphical output when a graphical front-end software system accepts an input user-generated and system-generated sequence of events from a fixed set of events. This interface is composed
of graphical objects where every object has a fixed set of properties. In some cases, GUI properties have separate values that contain a set of constituted GUI states during execution (Memon, 2007).

GUIs have become nearly omniPresent by interacting with software systems. Figure 2.1 describes the front-end underlying code for a GUI, where an end-user interacts with the software using the GUI.

![Interactions between the GUI and the underlying code](image)

Figure 2.1: GUI is the front-end to the underlying code (Memon, 2001)

### 2.3 Software Testing

Software testing is an important activity in software engineering. It can help people obtain accurate findings for software quality and diagnose errors in software execution (Ref et al., 2011). Software testing helps achieve the required results by evaluating an attribute or capability of a program or system (Choudhary & Kumar, 2011).

In software testing, analysis is performed to ensure that the quality of the tested product or service meets the specific requirement of stakeholders. Software testing can also provide an objective and independent view of a software to allow a business to escalate and determine the risks of software implementation. In a testing technique, a user can execute a program or application to find software bugs, errors, or other defects but is not limited to these. Software testing can validate and verify a software program application or product (Karnavel & Santhosh, 2013).
2.4 GUI Testing

GUI is crucial in perfecting a software product. Thus, GUI testing ensures software reliability. It is usually performed using a test script that interacts with a GUI through a sequence of actions. Moreover, generating event sequences is always challenging. Thus, the testing phase is divided into GUI testing, logical testing, unit testing and integration testing. These various testing approaches provide effectiveness, efficiency, correctness and accuracy in developing a quality product (Chen et al., 2008).

Testing a GUI is an important and difficult concern in developing quality software. In GUI testing, one of the most challenging considerations is the significantly large or infinite input domain of a non-trivial GUI application. Defining the region of convergence is necessary to help testers select the test cases from the input domain of GUI applications (Zhao & Cai, 2010). GUI testing provides information about the functionality of software for it to meet the design requirements in the GUI and develop the standard required product before being launched in the market.

GUI testing helps ensure that the requirements specified for a particular GUI are met. These specifications contain the navigation path or sequences that will be performed by a normal user and other sequences that a user can freely obtain through the GUI. These sequences can generate faults or failures in the software system. Thus, testing these sequences is crucial (Isabella & Emi, 2012).

2.5 Performance Parameters for Testing

Developers should achieve accuracy, correctness and performance-related issues in developing a program that can rePresent an algorithm. To obtain appropriate design specifications of a software system, developers must confirm the desired quality. This step is achieved using a process called software verification. This process involves verification activities during specification, design and implementation. Software testing is another process used to assess the functionality and correctness of software. The process analyzes the execution of a program (Gregory, 2008).
The performance of a GUI is a vital aspect in software development. Performance depends on choosing an algorithm that runs quickly and uses available computing resources efficiently. In testing the GUI performance, the time taken has been analyzed by a program and how the program can run on desired speed. Various available design methods can be used to design a test case. These design methods help test the application step by step efficiently and successfully. Thus, time and effort are saved by automating the testing process and the productivity of fault detection is increased as well. Automating the testing process involves different steps of planning. These steps can include effort, time, required resources and criteria for terminating a test case, reporting detected errors and evaluating the data collected.

The project plan must contain a schedule for testing milestones in its entire development period. The schedule will determine the time required for each section of testing milestone. This schedule will be designed effectively when it considers that test cases do fail. Hence, a program along with other test cases is required to ensure that bugs are fixed. This process will help determine more bugs with less effort and consumed time. However, it involves allocating time and resources for testing, running tests and collecting and executing test results (Wasif, 2007).

2.5.1 Complexity

The testing complexity of several classes of programs is measured in terms of the number of test cases required to demonstrate program correctness. Even for very restrictive classes of programs, no commonly used test criteria, namely, statement, branch and path executed at least once, are nearly sufficient to guarantee the absence of errors. A study of testing complexity identifies two new test criteria: one for testing a path and the other for testing a program. These new criteria suggest how test data are selected to obtain confidence in program correctness beyond the requirement of each statement, branch, or path tested at least once (Tai, 1980).

Complexity metric is based on the number of decisions in a program. Testers are important because these indicate the number of tests (including reviews) necessary to practically avoid defects. In other words, areas of codes identified as more complex are candidates for reviews and additional dynamic tests. Although
there are many ways to calculate complexity, the easiest method is to add the number of binary decision statements (if, while, for) (Tai, 1980).

2.5.2 Correctness

Testing a GUI is a tedious task because of various reasons. First, vast possibilities of GUI interactions are implied, producing different states based on various sequences of GUI commands. A large number of possible states indicate numerous input permutations. The large number of possible states results in many variations of inputs, which require considerably numerous testing. For instance, Microsoft released approximately 400,000 beta copies of Windows 95 to detect the failures of the program (Boghday et al., 2011).

The safety, authenticity and usability of a software depends on the functional correctness of its GUIs, which are hierarchical in nature. Testing a GUI for functional correctness is a crucial area of research because testing is a laborious job. This testing also involves extensive resources. A total of 50%–60% of the total cost of software development is purely allotted to software testing. At Present , GUI testing is a quick process because the characteristics of GUIs differ from those of traditional software systems. Hence, functional accuracy significantly contributes to software success. GUI testing is required to provide effectiveness, efficiency and accuracy (Kaur et al., 2010).

The accuracy of GUI assures the accuracy of the overall software system. Hence, comprehensive GUI testing is needed. This process involves all possible sequences of actions that need to be performed by widgets or the end-users using GUIs (Zacharias, 2012). GUI testing is considered a subdomain of software testing; for many reasons, testing the perfectness of a GUI is not an easy task (Memon, 2001). Many studies have identified the problems encountered in testing GUIs. The following are some of the findings of these studies:

(i) Numerous possible interactions are involved in a GUI state; thus, GUI testing requires enormous effort (Navarro et al., 2010).

(ii) Numerous input possibilities originate from the large number of possible GUI states; hence, achieving complete coverage by a test suite is a very complicated task (Huang & Lu, 2012).
(iii) GUI testing is a very complicated task because validating GUI states requires the selection of objects and their properties (Navarro et al., 2010).

(iv) GUI states are produced because of events performed in a particular sequence. Thus, not every sequence will produce the same state.

(v) The development and execution of test cases for testing GUIs include actions for validating input event sequences.

(vi) The mechanism for determining the successful execution of a software involves a particular test case for GUI (Memon, 2004).

2.5.3 Magnitude of Relative Error (MRE) for Performance Parameters

To illustrate the problem of magnitude of relative error (MRE), consider two prediction models A and B. If the MRE of model B is significantly lower than the MRE of model A, one can conclude that model B is better than model A (B is “more accurate” than A in the current software engineering terminology). To determine whether model A or model B is the best model between the two, the model evaluation metric must select the model that is closest to the actual value, considering that MRE ≤ 0.25 is acceptable for prediction MRE ≤ 0.25 as acceptable for prediction models and then obtain the MRE according to the equation (Tron et al., 2002).

\[
MRE = \frac{1}{N} \sum_{i=1}^{n} \left| \frac{Y - Y_i}{Y_i} \right| \tag{2.1}
\]

where \( N \) is number of values, \( Y \) is predicted value and \( Y_i \) is actual value.
2.6 Test Case Definition

Bunin and Schneider (2009) define a test case as “a set of conditions or variables under which a tester will determine whether a system under test satisfies requirements or works correctly.” In developing a test case, problems in the requirements and design of an application can also be identified (Bunin & Schneider, 2009).

2.6.1 Test Case Generation

Test case generation is a critical step in testing because the species of a test case is the effectiveness of the object being tested and the environment of the object also depends on the test inputs or its condition. The collection of test cases is known as a test suite, which is typically associated with an implementation dependency or testing goal. According to Liu (2001), test case generation is not limited to generating only input to implement the software, but also the corresponding output properties.

In 2011, Boghdady et al. proposed that test case generation is considered as the base or fundamental stage of any testing process. However, test data are still required to execute the generated test cases that enhance the test data generation, which is important compared with the test case generation.

2.6.2 Test Case Generation and Execution

Test cases for testing GUIs can be generated and executed in various ways. These methods, which are based on a fundamental idea, are indicated below (Eldh, 2011).

(i) Fully Manual: A test case is written manually to describe the step-by-step testing procedure. This process is time consuming and a test writer may miss some of the steps involved in software testing by mistake.

(ii) Automatically generating test cases and running manually: In this approach, testers integrate testing tools with a software to generate test cases automatically. Once the test cases are generated, the tester runs the tests manually.
Generating test cases manually and running them automatically: This approach allows a user to write test cases manually and testers run the cases automatically to save time and resources in testing. The manually generated test cases are mounted on any testing server that represents GUI testers.

Fully automated: In this approach, test cases are generated and executed automatically. This process is done by mapping GUI interfaces with testing tools for generating test cases. The testing tools execute test cases automatically.

Generating test cases is a crucial part of software testing because testing results depend on the test cases generated using any of the approaches mentioned above. In this study, the test cases are generated automatically using EFG and BXT techniques.

2.7 Automated GUI Testing

In this method of generating test cases, software testing tools execute automated test cases in the form of scripts of user actions. The purpose of this testing is to apply intensive testing in a short period of time with minimum resources. This process is significantly more convenient and useful compared with numerous quality assurance (QA) team resources. In testing, the automated testing tools execute tests, generate testing results and compare the results with previous test runs. Automated testing can also be performed any time and repeatedly (Fernando et al., 2014).

The following are the significant benefits of automated software testing:

(i) Software development lifecycle (SDLC) is shortened, which facilitates the marketing of software products as early as possible.
(ii) Optimum utilization of hardware resources and efficient testing are achieved.
(iii) Utilization of QA manpower resources is reduced, saving software cost.
(iv) Software stability and reliability are improved through enhanced testing.
(v) Software accuracy is improved.
(vi) Automated software testing is a convenient way to test software with minimum effort that produces high-quality results.
(vii) Contrary to an overloaded testing team, automatic software testing reduces the burden of testing of the QA department (Rauf & Alanazi, 2014).
Automated software testing is becoming widely accepted because it helps reduce human interaction, repetitive testing and releases. Automated testing is very practical and easy to use. In this method, testing tools execute the script of testing commands that replicate test procedures and fully control software testing (Yu et al., 2011).

Automated software testing allows testers to implement the entire software testing lifecycle (STL) to improve the efficiency of STL and make it effective using optimizing efforts throughout the entire STL. A major challenge of automated software testing is automating the integration and system testing efforts. The overall objective is to design, develop and deliver an automated test. This process ensures the capability of re-testing to enhance testing efficiency. Compared with traditional testing, if implemented and executed successfully, automated testing reduces the cost of software testing and the overall cost of a software product. This testing not only reduces cost, but also optimizes testing resources and improves product quality at the same time (Karnavel et al., 2013).

The major concern in testing GUI automatically is the need for functionally accurate GUI components. This step includes automating the testing efforts, which are highly resource intensive and error prone. Automating the GUI testing, which is accurate and reliable, saves cost as opposed to manual GUI testing (Latiu et al., 2013). However, whenever test cases are automated, the cases must be prepared according to the latest version of the GUI application. Any changes made in the application should also be replicated in the scripts of the automated test cases.

Test automation is gaining wide acceptance in software development communities. Hence a large number of test automation tools, such as QUITAR, Selenium and GUI Tester, have become available.

2.7.1 Automated GUI Testing Technique

Although various methods and techniques to generate test cases for GUI applications are available, such as module-based testing, unit testing and integration testing, not all these methods work under similar GUI applications, operating systems, or programming languages. Some methods and techniques perform well on GUI
applications and operating systems, such as Windows, DOS and Linux, as well as on programming languages, such as C++, Java and Visual Basic.Net.

2.7.2 Challenges in Test Automation

Completely automation of software testing is impossible because softwares are very complex in nature. However, more than 80% automation of GUI testing can be achieved. The challenges of automating GUI testing are as follows:

i) The name of Windows changes dynamically.

ii) Widgets like text boxes and combo boxes do not have unique identifying names.

iii) Produced GUI controls use managed and unmanaged codes.

iv) Applications are developed in different programming languages and operating systems.

v) Applications are developed using third-party controls that include various visual controls such as grid controls and schedule controls (Gandhi et al., 2014).

2.8 Model-Based Testing (MBT)

This technique is based on modules, where a structured testing approach is employed and testing results are based on the models of the system. These models reflect the structure of the application. Thus, when testing any application using MBT, one test case run shows results of the effect on the whole application and the other test case run illustrates the effects on a particular model being tested, such as Event Flow Graphs (EFG), Event Interaction Graphs (EIG) and Event Semantic Interaction Graphs (ESIG). At the end of the testing process, results are verified by comparing the two executions or runs and viewing the overall execution plan of the model. Thus, any difference in the results of both runs is attributed to the defect in application (Silva et al., 2008).

2.8.1 Event Flow Graph (EFG)

EFG shows all possible event sequences that can be performed on a GUI of a software. In this graph, the nodes represent GUI events and edges. The actions of a
GUI are represented as events in this graph. These actions can be clicking on a button or selecting an option in a list box, whereas edges represent the association between events. For example, T is an event that follows event S. Event T needs to be performed immediately after event S. The initial event is “click on button Exit,” which initiates another GUI that consists of events (Yes, No) for confirmation from the main page, as shown in Figure 2.2.

![Simple GUI in EFG](image)

**Figure 2.2: Simple GUI in EFG (Kanchan & Sharma, 2014)**

The entity flow diagram provides a way to map GUI for generating test cases systematically and automatically. The event sequence of a test case is generated by walking on the EFG with a graph traversal algorithm (Gautam & Sharma, 2014). The EFG diagram shows events and their association with other events and trigger vector. A dedicated event for initiating the flow of events is called a start event, whereas an event for terminating an application is called an exit event.

In an EFG, circles represent events, while arrows denote the dependencies among events. This research proposes an EFG that is different from the testing GUI. The proposed EFG shows all possible events along with their interdependencies among interesting events, as shown in Figure 2.3. EFG is built using user interactions after the events and their associations are analyzed (Singh, 2011).

Figure 2.3 depicts the components in the GUI illustrated in Figure 2.2. These components are create, select, reset and exit.
Figure 2.3: Event sequence in GUI (Singh, 2011)

Figure 2.4 shows the step-by-step algorithm of creating EFG for an AUT. In this algorithm, vertex $G=(V,E)$ represents a state where an event $e_i$ follows $e_j$, which may be performed immediately after $e_i$. Edge $(v_i, v_j) \in E$. The event $v_i$ follows the event $v_j$.

```
Input       : an application p
Output    : an event flow graph $G=(V,E)$ and a set of initial nodes I
1  $V \leftarrow \{ \}$
2  $E \leftarrow \{ \}$
3  $\text{SeenEvents} \leftarrow \{ \}$
4  $\text{WorkList} \leftarrow \text{new Queue}( )$
5  $\text{WorkList}.\text{add}(<<>)$
6  While $\text{WorkList}$ is not empty do
7    EventSeq = < $e_1, e_2, \ldots, e_{n-1}, e_n$ > $\leftarrow$ $\text{WorkList}$.remove( )
8    $\text{AvailableEvents}_n \leftarrow \text{RunNCapture}(p, \text{EventSeq})$
9    For each $e \neq \epsilon \in \text{AvailableEvents}_n$ do
10       $V \leftarrow V \cup e$
11       If EventSeq =< > then
12          $I \leftarrow I \cup e$
13       else
14          $E \leftarrow E \cup (e_n, e)$
15       end
16       If $e \notin \text{SeenEvents}$ then
17          $\text{SeenEvents} \leftarrow \text{SeenEvents} \cup e$
18          $\text{WorkList}.\text{add}(<<>)$
19       end
20   end
21 Return $(G=(V,E),I)$
```

Figure 2.4: Algorithm1: EFG (Bae et al., 2012)
This research also uses an EFG that employs an algorithm and applies the step-by-step algorithm in the two case studies. This process is done to generate various test cases and test all the events in a GUI.

2.9 Dynamic Event Extraction Technique

This technique, which is based on GUI testing, does not require a model creation phase. Moreover, it does not separate the phases of test case suite generation and execution. In this technique, test cases are generated for a test case suite and are simultaneously executed and generated (Bertolini et al., 2009).

2.9.1 Behavior Explorer Technique (BXT)

Figure 2.5 describes the pseudo-code algorithm for BXT in two main parts. The first part includes the decision to be made for selecting the screen to focus and the second part involves the stresses of the application from a selected screen. The second part performs the various steps for a given number of times or until the application fails or crashes because of the reasons stated below:

i) Identification of enabled events displayed on the current screen, such as the events that active components can process.

ii) Random selection of one event from all possible events.

iii) Data are developed for the current event and sent to the GUI where the data will be checked for failure or crash. This sequence corresponds to the code fragment in the line range
Figure 2.5: Algorithm 2 shows the pseudo-code for BXT (Bertolini et al., 2009)

A testing technique based on dynamic event extraction (DEE) will be used in this study (Bertolini et al., 2010).

2.10 GUI Tester Tool

This approach is employed to perform comparative experiments on GUI testing techniques that can exploit existing tools and the implementation for these techniques. However, configuring an available tool to execute environments and support fair comparisons is difficult because tools often use different types of events or have different efficiencies owing to their implementations and detailed configurations, which affect coverage and cost. The aim of this study is to compare the relationship between the two classes of techniques and not the tools. The tool is used to compare the ability of capturing events for equivalent GUI states with GUI testing techniques in the same working environment. The research framework used in this study is an open-source code framework that supports both DEE-based GUI testing and model-based GUI testing techniques (Xie et al., 2006).
2.11 Eclipse Framework

Eclipse framework is an integrated development environment (IDE) that contains an extensible plugin system for customizing the environment and a base workspace. Eclipse is mainly defined in Java language for it to be used in developing applications with various plugins. This framework may be helpful in developing various applications in other programming languages, such as C, Ada, Fortran, ABAP, C++, JavaScript, COBOL, Natural, Haskell, PHP, Lasso, Ruby (including Ruby on Rails framework), Perl, Prolog, Python, R, Scala, Clojure, Groovy, Scheme and Erlang. The software package Mathematica is also one of the framework’s developed program. Eclipse Java development tools (JDTs) are used as development environments, which includes Eclipse CDT for C or C++, Eclipse PDT for PHP, Java, Scala and many others (DesRivieres & Wiegand, 2004). Java language has numerous advantages when used with Eclipse IDE. These advantages are as follows:

i) With the assistance provided by IDE, writing Java code is easier than in a text editor. Hence, more time can be allotted for learning Java than for typing and looking up documentation.

ii) For a detailed execution of the Java code, the IDE debugger and scrapbook provide excellent output, which helps users to “see” the objects and easily understand the working steps of the Java program.

iii) IDE provides full support to users for the development practices of agile software (Dexter, 2007). An example is Test Driven Development.

2.12 Cobertura Tool

Cobertura is a free Java tool that determines the percentage of code accessed by tests and identifies gaps in test coverage in the Java program based on JCoverage. The main features of Cobertura includes the following: it can be executed from a common line or ant, instruments Java byte code after its compilation, reports generation in HTML or XML file type and describes the line percentage and branches (which covers the test case and overall project for each package). Cobertura can show the complexity of each class in McCabe Cyclomatic code and test cases, average Cyclomatic code complexity for each package and overall product.
Moreover, it can sort HTML results in ascending or descending order based on the percent of lines covered, class name and percent of branches covered.

2.13 Related work

Different approaches are used for GUI testing; however, the model-based testing is better than the other approaches and such an automated technique can be used to generate test cases. This automated technique presents a model for GUIs, named the event flow model, which is mostly used for the different aspects of GUI testing. For the effectiveness of GUI faults, the event flow model helps generate the GUI test cases in a large number quickly and efficiently. Additionally, this model can be recycled or re-used because it can be converted to generate additional test cases for the same or modified version of GUI (Kanchan & Sharma, 2014).

By contrast, an AJAX application user interface and transitions described by dynamic analysis are used to construct a state flow graph (SFG) between the models of two states. An equivalent static pages set is generated from this model, which can be used for different applications, such as the application of search engines to their content and state-based test performance. The approach to detect the clickables is a Crawljax that is completely reliant on a heuristically based approach, such as the elements of DOM, which can correspond to active user interface components and crawls application randomly exercising these clickables (Mesbah & Bozdag, 2008).

Yuan and Memon (2010) presented an autonomous model-driven technique to generate test cases for GUI-based applications. By using the structural EIG model of the GUI, a feedback technique was automatically generated from the execution of the seed test suite (STS). New test cases could automatically be generated during the STS execution, resulting in the runtime effect of each GUI event relationships on all other event pinpoints, such as event semantic interaction (ESI). Two studies on eight applications for the feedback-based technique demonstrated that (1) these applications could significantly improve the existing techniques and help the software identify serious problems and (2) the GUI state yield test suites captured ESI relationships, which identified more faults compared to their code, event and event interaction coverage equivalent counterparts.
Dynamic analysis can construct an SFG that models the states of the user interface and interstate transitions of the AJAX application. From this model, a set of equivalent static pages can be generated for use in various applications (e.g., applying search engines to their content and performing state-based testing). Crawljax relies on a heuristically based approach to detect clickables, which are elements of the document object model (DOM) that may correspond to active user interface components and crawl the application by exercising the clickables in random order (Mesbah & Bozdag, 2008).

2.14 Chapter Summary

The background study of GUI testing reviewed the testing techniques related to previous works. Eclipse framework and GUI tester tool were discussed in this chapter. To overcome the highlighted problem and achieve the research objectives, the designed research methods are described in the next chapter.
CHAPTER 3

METHODOLOGY

3.1 Introduction

This chapter describes the research methodology of applying EFG and BXT techniques to test the graphical user interface (GUI) for the two case studies analyzed in this research. This chapter is organized into four subsections. Section 3.2 presents an overall view of the research methodology and Section 3.3 thoroughly describes each procedure. Section 3.4 discusses the two case studies explored in the investigation and Section 3.5 summarizes this chapter.

3.2 Research Methodology

The research methodology consists of two primary steps that are completed to achieve the objectives of this study stated in Chapter 1. Figure 3.1 illustrates these steps. The first step involves the application of the EFG and BXT techniques in the GUI tester tool. The core processes of these techniques are used to examine the case studies stored in the application under testing (AUT) folder. From these, the XML report is generated. The second step includes the measurement and determination of the time and correctness of the generated test cases based on the XML report. Finally, the results of the previous step are analyzed and compared to obtain the most accurate technique.
The steps shown in Figure 3.1 are thoroughly explained in the succeeding sections.

3.3 **Step 1: Application of the EFG and BXT techniques**

Step 1 consists of two procedures. The first one is the application of the EFG technique and the second is the adoption of BXT.
3.3.1 Step 1-1: Application of the EFG technique

As previously mentioned, the preliminary step of the research methodology is the application of the EFG technique to the case studies, which are the Paint and Present applications of the GUI tester tool working in the Java language platform. This step, however, incorporates four other processes, as shown in Figure 3.2.
REFERENCES


Xie, Q., & Memon, A. M. (2006,). Studying the characteristics of a" Good" GUI test suite. *In Software Reliability Engineering, 2006. ISSRE’06. 17th International Symposium on* (pp. 159-168). IEEE.

Wasif Afzal. (2007). Metrics in Software Test Planning and Test Design Processes. *School of Engineering Blekinge Institute of Technology*


