Graphic User Interface Modelling and Testing Automation

A thesis submitted in fulfilment of the requirements of the degree of

Doctor of Philosophy

Xuebing Yang

Supervisor: Professor Yuan Miao

School of Engineering and Science
Victoria University

May 2011

DECLARATION

I, Xuebing Yang, declare that the PhD thesis entitled *Graphic User Interface Modelling and Testing Automation* is no more than 100,000 words in length including quotes and exclusive of tables, figures, appendices, bibliography, references and footnotes. This thesis contains no material that has been submitted previously, in whole or in part, for the award of any other academic degree or diploma. Except where otherwise indicated, this thesis is my own work.

Signature

Date

Xuebing Yang
May 2011

PUBLICATIONS

1. Y. Miao, **X. Yang**. "An FSM based GUI Test Automation Model", the 11th International Conference on Control, Automation, Robotics and Vision, ICARCV 2010. Singapore, December 7-10, 2010
2. **X. Yang**, Y. Miao and Y. Zhang. Model-driven GUI automation for efficient information exchange between heterogeneous electronic medical record systems, 19th International Conference on Information Systems Development, ISD 2010. Prague, Czech Republic, August 25 - 27, 2010
3. **X. Yang**, Y. Miao. Distributed Agent Based Interoperable Virtual EMR System for Healthcare System Integration. Journal of Medical Systems. August 2009.
4. **X. Yang**, Y. Miao and Y. Zhang. GP eConnect: Extends e-referrals exchange to healthcare providers' collaborations. IADIS eHealth 2009. Algarve, Portugal, 21-23 June 2009.

ACKNOWLEDGMENTS

First and foremost, I would like to express my deep gratitude to my supervisor, Professor Yuan Miao, for his exceptional support, encouragement, and patience during all stages of my research for this dissertation. This dissertation would not have been possible without his invaluable advice and guidance. I feel blessed to have had Professor Miao as my supervisor. I will be forever in his debt.

I also wish to thank my co-supervisor, Professor Yanchun Zhang, for his constant help, discussions, and many constructive suggestions throughout the course of my doctoral study. I have taken the great pleasure in working with my colleagues in the Centre for Applied Informatics Research (CAI) at Victoria University. I would like to thank them for their valuable suggestions and discussions during the process. I thoroughly enjoyed their fruitful collaboration, and I gained invaluable skills by working with them.

I am grateful to the School of Engineering and Science for supplying a very good research environment, and the staff members who frequently offered assistance, particularly the School Postgraduate Coordinator, Dr. Gitesh Raikundalia, A/Prof. Xun Yi, Dr. Fuchun Huang, Professor Pietro Cerone, A/Prof. Hao Shi, and the Scholarship Coordinator for Postgraduate Research, Ms. Lesley Birch.

I am also grateful to our research partner organization Westgate General Practice Network (WGPN) for providing me with a practical working environment, and the staff members who encouraged me through all the stages of my study, particularly the Chief Executive Officer Dr. Corinne Siebel, the Information Management Officer Mr. Manfred Queteschiner, and the Psychologist Ms. Sandra Plant.

I wish to extend my deepest gratitude to my parents for their love, support and encouragement. I thank my be-loved wife, Hong Tao. Without her endless love and

support, I could not possibly have come this far. I would also like to thank my son, Zeming Yang, for constantly encouraging me and making everything I do meaningful.

ABSTRACT

A Graphical User Interface (GUI) is the most widely used method whereby information systems interact with users. According to ACM Computing Surveys, on average, more than 45% of software code in a software application is dedicated to the GUI. However, GUI testing is extremely expensive. In unit testing, 10,000 cases can often be automatically tested within a minute whereas, in GUI testing, 10,000 simple GUI test cases need more than 10 hours to complete.

To facilitate GUI testing automation, the knowledge model representing the interaction between a user and a computer system is the core. The most advanced GUI testing model to date is the Event Flow Graph (EFG) model proposed by the team of Professor Atif M. Memon at the University of Maryland. The EFG model successfully enabled GUI testing automation for a range of applications. However, it has a number of flaws which prevent it from providing effective GUI testing. Firstly, the EFG model can only model knowledge for basic GUI test automation. Secondly, EFGs are not able to model events with variable follow-up event sets. Thirdly, test cases generation still involves tremendous manual work.

This thesis effectively addresses the challenges of existing GUI testing methods and provides a unified solution to GUI testing automation. The three main contributions of this thesis are the proposal of the Graphic User Interface Testing Automation Model

(GUITAM), the development of GUI Defect Classification and the proposal of the Long Use Case Closure Envelope Model.

Graphic User Interface Testing Automation Model. This research proposed a GUI testing automation model (GUITAM), proved that GUITAM is not only able to automate all testings that EFG can automate, but also able to model a series of important scenarios which EFG cannot. The efficiency of GUITAM, in terms of storage and computational complexity, was also proved to be at least as good as that of the EFG model.

GUI Defect Classification. This research systematically established, for the first time, a GUI defect classification, which includes criteria of classifying defects, distributions of defects and classification directed test case generation. Defect classification allows test cases to be designed for specific classes of defects, thus effectively avoids large unnecessary permutations in existing models.

Long Use Case Closure Envelope Model. This research proposed a knowledge model called the Long Use Case Closure Envelope Model, for representing user experience of interacting with the GUI, and generating task-oriented test cases automatically. By using a use case as the backbone, an envelope model was developed to encapsulate all possible branches of states and events related to a given task. Highly efficient and effective task-oriented test cases can be automatically generated from the envelope.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 1

Introduction

With the development of Information Technology (IT), computers and intelligent devices have become ubiquitous in our society. Computers and intelligent devices provide functions by running software systems. Software systems are present in virtually all aspects of modern society, aeroplanes and cars have computer boards, banks manage user accounts and relevant information with banking systems, trains are scheduled by coordinating systems, doctors use Electronic Medical Record (EMR) systems to record patient information, people pay for goods and services electronically, and shopping can be done on the Internet. Increasing implantation of software systems makes our daily lives more dependent on their functioning without errors. For instance, a slight error in an airline coordinating system may cause air crashes, and minor leaks of Internet Banking may lead to misplacement of customers' money. Since software systems are simplified models of our real lives, no one can claim that they are perfect and free of defects. The correctness of the functions of a system depends not only on the exact, unambiguous and complete capture of the customer requirements, but also on how thoroughly the system is tested before being put into use. It goes without saying that software testing is critical for providing quality-software related products. Nowadays, a Graphical User Interface (GUI) is widely used as a way for users to

interact with software systems. More than 45% of the total source code is used for implementing a GUI which makes GUI testing inevitable. Due to the characteristics of the GUI, GUI testing is much more difficult than conventional software testing. This thesis focuses principally on providing solutions to tackle the difficulties in GUI testing. Before presenting our work, this chapter will introduce conventional software testing and the state-of-art of GUI testing researches.

This chapter is organized as follows. First, the background of conventional software testing is reviewed in Section 1.1. In Sections 1.2 and 1.3, outlines of research on GUI testing and automation are given. Section 1.4 lists the issues in GUI testing automation and summarises our solution. The structure of this thesis will be presented in Section 1.5.

1.1 Conventional software testing.

Software testing is one of the most important parts of the life cycle of software system development and costs more than any other part. Software testing is labour and resource intensive. Usually software testing accounts for 50-60% of the total cost of software development [1]. As shown in table 1.1[2], the earlier the faults are found, the less the repair will cost.

Table 1.1 Relative cost of repairing software faults.

Stage	Relative cost of repair
Requirements	0.1 ~ 0.2
Design	0.5

Coding	1
Unit testing	2
Acceptance testing	5
Maintenance	20

It is impossible to exhaustively test an application. The reasons for this noted by Kaner, Falk, and Hguyen [3] are as follows: (1) the domain of program inputs is too large; (2) there are too many possible input paths, and (3) design and specification issues are difficult to test. In software testing, only a small percentage of possible input combinations can be selected to generate test cases, and these are selected based on certain coverage criteria. Test cases are executed either manually or automatically and check whether the outputs conform to the software specifications.

A conventional software testing procedure encapsulates a number of steps. These steps include test planning, test case generation, test check with oracles, and analysis of the results. Figure 1.1 shows the typical process of conventional software test case execution.

In Figure 1.1, input test cases are usually generated according to certain coverage criteria. Manually executing the test cases one by one is laborious and very inefficient. In conventional software testing, testing tools (executors) can be used to perform the execution automatically. An automatic executor may perform thousands of test cases in one minute. The expected outputs are supposed to be worked out for each test case, normally by analysing the specifications. After each test case is executed, the test oracle will compare the real output with the expected output and report the results.

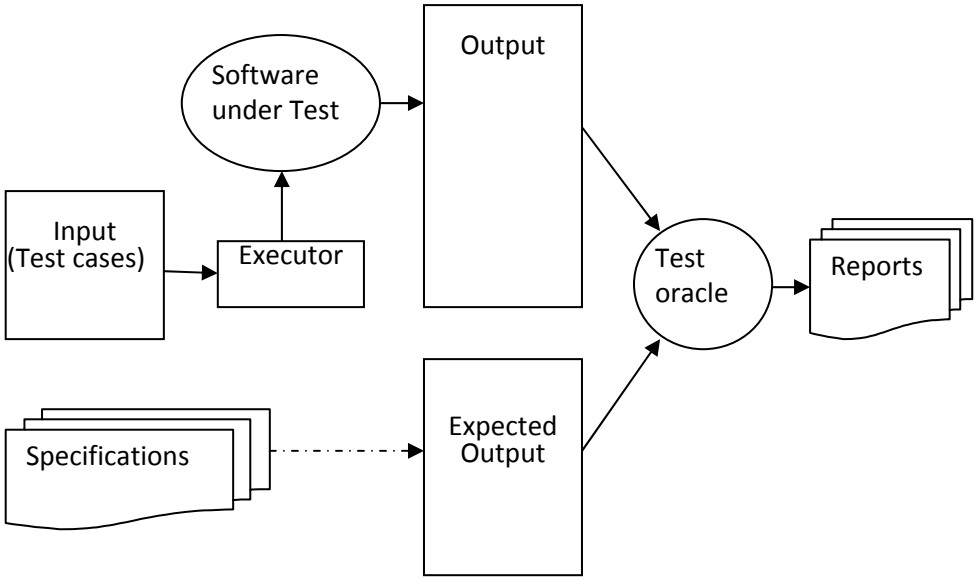


Figure 1.1 Process of conventional software test case execution

1.2 GUI Testing

Today, most software products provide GUIs as the main interface for users to access their functions. GUIs have become dominant in comparison with other kinds of interfaces such as command line based consoles. A GUI is a type of user interface that allows users to interact with programs in more ways. A GUI offers graphical icons, and visual indicators to fully represent the information and actions available to a user. The actions are usually performed through direct manipulation of graphical elements. In a typical GUI, instead of laboriously typing commands to tell a computer what to do, a user can simply choose commands by activating or manipulating the pictures. For example, a user may click on a button or drag an icon with an input device such as a mouse. GUIs are intended to make computers "user friendly" by simplifying tasks and

decisions, and by creating a visual representation of a computer system to which people can more easily relate. A significant aspect of GUIs is that they are not merely different to look at, but can increase the efficiency of learning and usage over text-based interfaces. GUIs can also lead to higher productivity because they lend themselves better to performing multiple tasks simultaneously. Well-designed GUIs not only represent files, programs, and procedures visually, but also provide streamlined methods for completing tasks and take into account users' needs and expectations.

The ease of using computers through GUIs makes computers ubiquitous in our daily lives. People with little knowledge of computers can thereby use them to perform tasks with only a small amount of training, or without any training at all. In order to provide quality GUIs, testing them before they are put into use therefore becomes crucial. Recognizing the importance of the GUI, today's software developers are dedicating an increasingly large portion of software code to implementing them. On the one hand, GUIs provide users with a convenient and intuitive way of operating software and richer information on limit display area. They facilitate users' experiences of these applications. On the other hand, to avoid defects, software providers have to dedicate much greater effort to testing the GUIs and GUI related codes. Because GUIs face the terminal users directly, slight inconveniences or small faults will lead to complaints to refusal to use the software. Obviously, to ensure the software quality and usability,

comprehensive testing of GUIs must be implemented before the software is delivered to the terminal users.

Unlike conventional non-GUI software testing, the distinct characteristics of a GUI make testing it much more difficult. While some GUI development frameworks such as Swing make GUI development easier, they unfortunately make GUI testing much harder. In contract to traditional non-GUI applications, GUI based applications are written in an event-based style, where the application needs to handle a diverse set of events representing user inputs such as mouse movements, object manipulations, menu selections, and opening or closing of windows. A GUI is implemented as a single large program handling all user interactions, and is also connected to the underlying business logic. When a user interacts with the software by using the GUI, the underlying code will be executed to perform the functionality (see Figure 1.2). Sometimes different views of GUI may be related to the same code. For example, a "Save" button may share the same background codes with a "Save" menu item. Objects can also be linked to each other by underlying code without explicit visual connection. Normally when an interface appears, widgets on the GUI such as buttons, menu items, check boxes, and radio buttons will be exposed to the user at the same time. A user may trigger the events in any sequence. The number of permutations of the event sequences is infinite or extremely large which makes it impossible to perform exhaustive testing of the GUI even on one single and simple window. A GUI test case includes a series of events which can only be performed one after another

with time spans for the GUI to react. Execution of test cases on GUIs is time-consuming. As compared to unit logic, in which 10,000 cases can often be automatically tested within a minute - 10,000 simple GUI test cases need more than 10 hours, which at a cost of 600 times of the former.

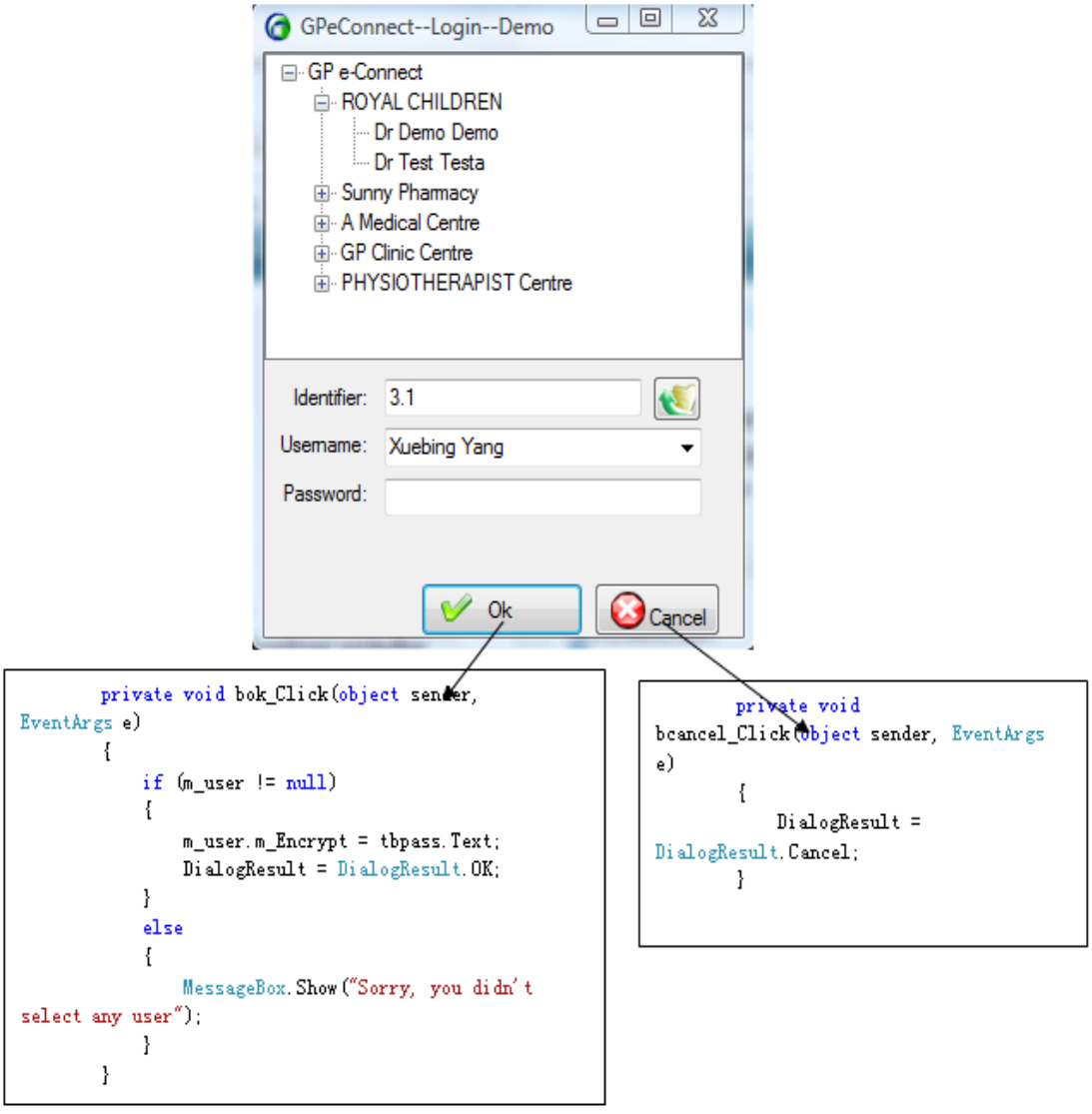


Figure 1.2 GUI and its underlying codes

Due to the characteristics of GUIs, which are different from those of traditional software, techniques typically applied to software testing are not adequate for GUI testing. In comparison with conventional software testing, GUI testing involves more effort, such as event simulation. A typical GUI test case execution process is shown in Figure 1.3.

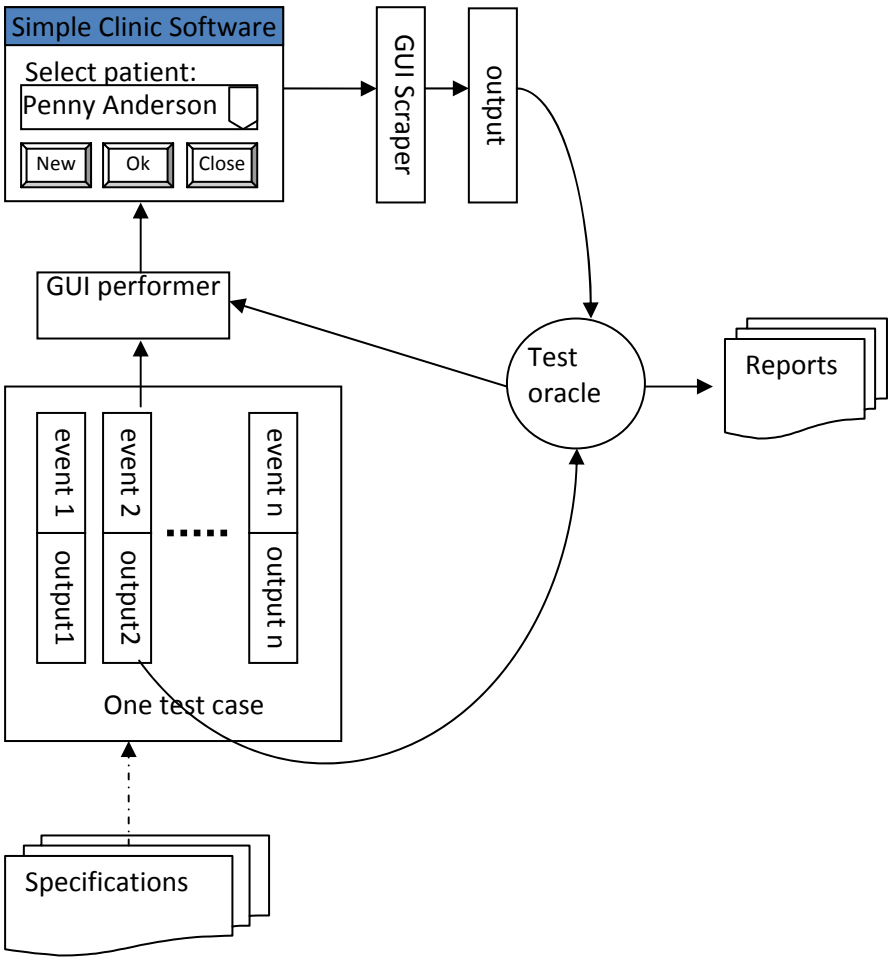


Figure 1.3 Process of GUI test case executions

In Figure 1.3, in contrast to the conventional test case execution shown in Figure 1.1, each test case includes a series of events. Each event needs to be performed on the given GUI by simulating a user's operation. Unlike conventional software, GUI based software takes time to react to each event. After the GUI finishes the reaction, a GUI scraper reads the status of all the components and compares them with the pre-stored expected information linked to the event by using a test oracle. Any mismatches in the results may be fed-back to the GUI performer so that it can decide whether the other events of this test case need to be performed or not. All mismatches will be put into a defects report for later analysis. If the GUI performer works manually, each test case needs a long time to execute, normally more than 15 seconds. If the result checking is also done manually, about another 20 seconds are requied. Supposing there are 10,000 cases, the total testing time is about 100 hours. If the test cases are executed automatically by mimicking user operations, it takes about 3 seconds for each test case to be performed and about another 1 second for the oracle check. 10,000 cases need about 10 hours to be executed. Apparently, automatic GUI testing is about 10 times more efficient than manual GUI testing and saves human resources. GUI testing automation has therefore been attracting more and more researchers in recent years.

1.3 GUI Testing Automation

GUI testing automation involves several steps. Firstly, a set of test cases needs to be generated automatically according to certain coverage criteria. Because of the difficulties, especially the huge number of possible test cases, it is impossible to

perform exhaustive testing. Many GUI objects, properties and events are related to each other. Without the knowledge of how these parts are connected and how they work, test cases can only be generated manually. In other words, to automatically generate test cases, comprehensive models are needed to model the whole GUI system of an application. Secondly, after the test cases are generated, they need to be executed automatically. Each test case comprises an event sequence with GUI state and object information linked to them. An executor is used to mimic a user to perform all the events automatically in sequence. Thirdly, the aim of executing the test cases is to check whether the GUI performs as intended. This needs proper mechanisms called 'test oracles' to check and report the incongruences. Automatically generating test oracle information for the checking is very difficult for GUI applications. The oracle information can either come from application specifications or from a base version of the application under test. Regressive GUI testing may automatically generate oracle information by executing the same test cases on the base version and recording the corresponding GUI states for comparisons. Some defects may cause incorrect GUI states which can also lead to unexpected events, which can also make further execution of the test cases useless. Consequently, the execution of the test case must be terminated when an error is detected [6]. Furthermore, the test results need to be analysed and reported. The major steps of GUI testing automation are as follows [13].

Step 1: Coverage criteria design. Because it is impossible to carry out exhaustive testing, certain coverage criteria are needed to determine how to select test cases and to

what extent the test will be done. In GUIs, a user may click on a window randomly and a different order of clicks may lead to different results. Since the number of all permutations is extremely huge, only a small number of the sequences may be selected as test cases.

Step 2: Test cases generation. Test cases in GUIs are actually sequences of user inputs. Each test case is represented as a series of events, such as clicks, keys, or menu selections. According to the given criteria, test cases are selected to perform actions on the GUIs of an application.

Step 3: Test oracles development. Without a mechanism to predict the right output, one can never tell whether there are faults after performing a test case. Test oracles are used to check the output after each test case is executed. Developing test oracles is tedious and time consuming.

Step 4: Test case execution and output verification. Executing test cases can be done automatically with a proper GUI state model by simulating a user's operations on a given GUI. After each step of a test case is executed, the GUI state is retrieved by a GUI Scraper and compared by the oracles. Any faults will be recorded for later analysis.

Step 5: Test results analysis. Once all the test cases have been executed and the comparisons have been performed between the expected outputs and actual outputs, the report of the test will be analysed. Not every test suite can test all the parts of the

software. Which parts of the software are tested and which are not tested should be reported as well.

Step 6: Regression testing. Regression testing is used to help ensure the correctness of the modified parts of the software as well as to establish confidence that changes have not adversely affected previously tested parts.

All the six steps mentioned above are indispensible to GUI testing. It is so difficult to fulfil GUI testing automation for all the steps that many researchers have been trying partially GUI testing automation [7-13].

GUI testing automation is traditionally through the Capture and Replay (CR) technique [14-15]. CR tools provide a basic automation solution by recording mouse coordinates and user actions as scripts. A major problem of using mouse coordinates is that the scripts can break with even minor changes to the GUI layout [42].

To overcome the difficulties associated with recording mouse actions and coordinates, a series of modern CR tools have been developed. Among these are the popular Quick Test Professional (QTP) [16], Abbot [17], Selenium [18], Rational Functional Tester (RFT) [19], Win runner [20], SilkTest [21], and IBM Rational Robot [22]. These CR and visual test tools capture values of various properties of GUI objects rather than mouse coordinates. The recorded events are connected to GUI objects (widgets such as Textbox, Button, etc.) by using unique names. Unique names can be identified with collections of values of the properties of GUI objects. When interacting with the

application, the unique name will be used to obtain the reference to the real object in the GUI and recorded events will be performed on the designated object.

CR tools are very useful, but inadequate for performing true automation tasks such as GUI testing. They can only record user actions on the GUIs of the given applications and replay these actions. Test automation requires knowledge of the logic or workflow of the GUIs. Researchers have developed a series of techniques for GUI test automation [23-27]. However, these proposals are based on models manually created from the application's specifications [10, 28-30] which involve much time and resources. This specifications based GUI automation has been proven to be impractical and not really feasible [31-32].

To automate the process of GUI testing, many researchers have tried to use model based GUI testing [6, 9, 24, 26, 33-41]. Of these models, a graph-traversal model, the event flow graph (EFG), and its later version, the event interaction graph (EIG) [28-29, 37, 42-51] and the event sequence graph (ESG) [6, 33, 52] have been known to be successful to some extent in recent years in generating sequences of events for creating test cases. The EFG based test automation [37] has been claimed as the first practical GUI automated smoke test and much subsequent research was based on EFG [28, 29, 31, 32, 37, 46, 47, 49, 53-55]. This was followed by research on automated black-box GUI testing. In this research, the event flow graph (EFG) was proposed as the core-enabling model. In EFG, each vertex represents an event. All events which can be executed immediately after this event are connected with directed edges from it. A

path in EFG is a legal executable sequence which can be used as a test case. EFGs can be generated automatically using a tool called GUI Ripping [44]. Traversing the EFG with certain strategy can generate test cases.

The Event Sequence Graph (ESG) [56] is also an event focused model. ESG represents the system behaviour and the facilities from the user's point of view while interacting with the system. ESGs are directed graphs; their nodes represent events and edges represent valid, correct sequences of events. Two pseudo vertices, '[' and ']', symbolize entry and exit where any node can be reached by entry, and any node can reach the exit. Any sequence of vertices connected by an edge is called a legal event sequence (ES). Two events connected by an edge are called an event pair (EP). An ES starting at the pseudo vertex '[' and ending with the pseudo vertex ']' is called a complete event sequence (CES). CESs are considered to perform successful runs through the ESG, i.e., they are expected to arrive at the exit of the ESG that models an application. In other words, they deliver desirable events. For (positive) testing, CESs are used as test inputs [6].

Both EFGs and ESGs are event-oriented models which ignore the actual state of the GUI. All these models can only provide basic GUI automation functions which focus on part of the GUI testing automation steps. To the best of my knowledge, no solutions or techniques can fully automate the entire GUI testing automation steps mentioned above. Current GUI testing techniques are still incomplete and labour-intensive.

1.4 Challenges in GUI Testing Automation

The characteristics of GUI make GUI testing very difficult. Firstly, event-driven architecture ensures the uncertainty of user inputs. In conventional command line software, inputs can be simply described as strings. In GUI applications, inputs are much more complex. A user may click on any pixel on the screen in any order. Key presses may happen while the mouse is being clicked or pressed. The input space is huge. Secondly, an automatic test suite has to simulate these events somehow. To automate, an automation tool needs to be able to mimic a user performing events. Thirdly, the state of the GUI is a combination of the states of all its components. Even the simplest components have a large number of attributes and methods. A distinct set of a combination of all the attributes constitutes a GUI state. The number of states increases exponentially as the number of components in a GUI increases.

Due to the difficulties of GUI testing, GUI testing automation therefore faces a number of challenges:

1) GUI states explosion. A GUI state comprises of a set of objects and their property values. Any difference in number of objects or property values may mean a different state. Some property values have huge or even infinite domains of possible distinct values which make the number of GUI states in turn huge or infinite. Without a proper method to limit the explosion of GUI states, it is infeasible to perform testing automation for GUIs.

2) Test case generation. Given the combinatorial explosion due to arbitrary event interleaving, selecting a feasible number of event sequences is paramount. To reduce the number of test cases, some methods try to limit the length of test cases to a certain number of steps. How to automatically generate efficient test cases is a big challenge for all existing GUI Testing Automation models.

3) Oracles development. In GUI testing, the outputs are manifested by the values of properties of widgets in the GUI. The expected values need to be prepared before testing. One of the difficulties is how to arrive at the expected values. Besides, one test case may contain a number of events. After each event is executed, the GUI state needs to be checked. If all the values of the properties of all the widgets are selected, the storage needed is remarkable. How to select values of properties of the widgets for user collecting and checking is another difficulty in oracle development.

4) Coverage of test case suite: Conventional coverage methods are not suitable for GUI testing. GUI behaviours are represented by components statuses and events. The challenges include how to decide what part of the GUI, what kind of behaviour of the GUI or what group of events to be covered and tested. How to define coverage criteria is also a challenge.

5) Regression testing: Regression testing is used to help ensure the correctness of the modified parts of the software as well as to establish confidence that changes have not adversely affected previously tested parts. How to make use of old test cases for

generating new test suites and how much the previous versions of applications can be used is very important.

This thesis effectively addresses the challenges of existing GUI testing methods and provides a unified solution to GUI testing automation. The three main contributions of this research are the proposal of the Graphic User Interface Testing Automation Model (GUITAM), the development of the GUI Defect Classification and defect classification directed test case generation, and the proposal of the Long Use Case Closure Envelope Model for task-oriented long test case generation.

1.5 Thesis structure

The remainder of this thesis is organized as follows. Chapter 2 introduces some basic concepts and definitions used in this thesis, and surveys some technologies which are related to both conventional and GUI testing. Chapter 3 presents the novel Graphic User Interface Testing Automation Model (GUITAM) to automate the procedure of GUI testing. Chapter 4 presents defect classification and defect classification directed automatic test case generation for focused GUI testing. Chapter 5 develops the Long Use Case Closure Enveloping Model which makes use of use cases to automatically generate task-oriented test cases. Finally, Chapter 6 concludes the thesis and outlines possible future work.

Chapter 2

Background and Related Work

The research presented in this thesis focuses on providing a unified solution to GUI modelling and testing automation, which includes GUI representation, GUI testing automation models, defect classification and a knowledge model for task oriented test case generation. This chapter will introduce the background, relevant terminologies, and related research in the area of software testing and GUI testing automation.

2.1 Software Testing Principles

To better understand the whole process of software testing and avoid ambiguity in presentation, some underlying principles in software testing will be introduced in this section.

2.1.1 Terminologies

The purpose of software testing is to reveal software ***faults*** in order to correct ***errors*** made during the implementation of the application under test (AUT) and to ensure the quality of the AUT [58]. We say that a program's execution is ***correct*** when its behaviour matches the functional and non-functional requirements in the AUT's specifications [57]. An ***error*** is a mistake made by a programmer during the implementation of a software system [58]. If the implementation is not as described in the specifications of the AUT, this is an error implementation. A ***fault*** is a collection of

program source code statements that cause a ***failure.*** A ***failure*** of an application is an external, incorrect behaviour of a program [58]. A d***efect*** generally refers to any of these concepts, including *faults*, *errors* and *failures*.

According to A.M. Memon, M.E. Pollack, G.M. Kapfhammer and M.L. Soffa [43, 59], test suites are used to assess the quality of an AUT. A test suite includes a series of test cases and states.

Definition 2.1[43,49]**:** A ***test suite*** T is a triple $(\Delta_0, <T_1,...,T_e>, <\Delta_1,..., \Delta_e>)$, consisting of an initial external test state, Δ_0, a test case sequence $<T_1,..., T_e>$for state Δ_0, and expected external test states $<\Delta_1,..., \Delta_e>$,

where $\Delta_f = T_f(\Delta_{f-1})$ for $f = 1,...,e$. $\Delta_f = \{(var_\Delta, val_\Delta) \in U_\Delta \times V_\Delta \mid value\ (var_\Delta, f) = val_\Delta\}$

Δ_f denotes the externally visible state of the AUT. Δ_f can be viewed as a set of pairs where the first of each pair is a variable name and the second is a value. U_Δ and V_Δ denote the universe of valid variable names and variable values respectively.

Definition 2.2[43,49]**:** A ***test case*** $T_f \in <T_1,...,T_e>$ is a triple$<\delta_0, <o_1,...,o_g>, <\delta_1,..., \delta_g>>$ consisting of an initial internal test state, δ_0, a test operation sequence $<o_1,...,o_g>$, for state δ_0, and expected internal test $<\delta_1,..., \delta_g>$, where $\delta_h = o_h(\delta_{h-1})$ for $h = 1, ... , g$. $\delta_h = \{(var_\delta, val_\delta) \in U_\delta \times U_\delta \mid value\ (var_\delta, h) = val_\delta\}$

$T_f \in <T_1,...,T_e>$ can be viewed as a sequence of test operations that cause the AUT to enter into states that are only visible to T_f. δ_h denotes the internal state that is created after the execution of T_f's test case operation o_h.

Definition 2.3[60]**:** A test suite *T* is ***independent*** if and only if for all $\gamma \in \{1 \ldots e\}$, $\Delta_\gamma = \Delta_0$. *Independent suite* is a restricted type of suite, where each test case returns the AUT back to the initial state, Δ_0, before it terminates.

Software testing can be divided into two categories: execution based software testing and non-execution-based testing. Execution-based software testing techniques are either program-based, or specification-based, or combined [61, 62]. Non-execution-based software testing can be performed through software inspections [63]. During a software inspection, software engineers manually examine the source code of a system and any document that accompanies the system. The inspection can be guided by a software inspection checklist [64] or by using scenario-based reading techniques [65, 66]. Figure 2.1 shows the types of software testing.

Non-execution based software testing is usually done manually which is not our focus. The executions in execution-based software testing can be automated to some extent. Many testing automation models are for execution-based testing.

2.1.2 Representation of program source code

Conventional program source codes are made up of a set of methods (procedures or functions). Before generating test cases, the test case adequacy criteria need to be analysed. To analyse the adequacy of the AUT, the structure of the program needs to be modelled. There are many different graph-based representations for programs. For example, the class control flow graph (CCFG) represents the static control flow

between the methods within a specific class [67-70]. Inter-procedural Control Flow Graph (ICFG) [71] represents the control flows of each method within an AUT. Different representations for the AUT influence the measurement of the quality of existing test suites and the generation of new tests. These graph-based representations can be generated automatically by scanning the source codes. Here we just introduce the ICFG.

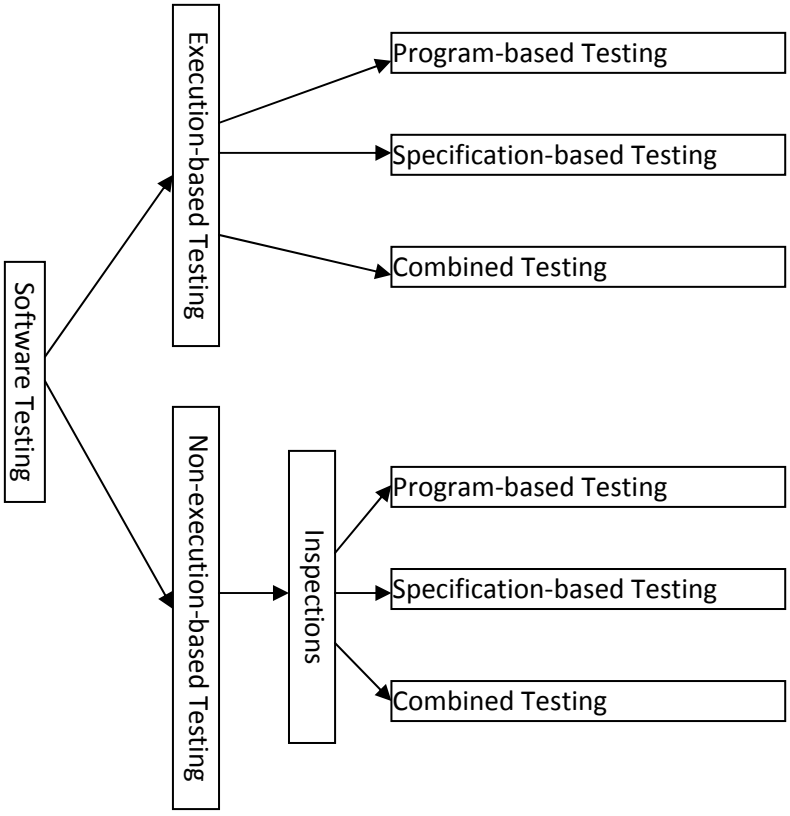


Figure 2.1 Types of software testing

An ICFG is a collection of control flow graphs (CFGs) $G_1, G_2, \ldots, G_u$ which correspond with the CFGs for the program's methods $m_1, m_2, \ldots, m_u$, respectively. We define

control flow graph G_v so that $G_v = (N_v, E_v)$ and we use N_v to denote a set of CFG nodes and E_v to denote a set of CFG edges. Furthermore, we assume that each $n \in N_v$ represents a statement in method m_v and each $e \in E_v$ represents a transfer of control in method m_v. Also, we require each CFG G_v to contain unique nodes $entry_v$ and $exit_v$ that demarcate the entrance and exit points of method m_v, respectively. We use the sets $pred(n_\tau) = \{ n_\rho \mid (n_\rho, n_\tau) \in E_v \}$ and $succ(n_\rho) = \{ n_\tau \mid (n_\rho, n_\tau) \in E_v \}$ to denote the set of predecessors and successors of node n_τ and n_ρ, respectively. Finally, we require $N = \cup \{N_v \mid v \in [1, u]\}$ and $E = \cup \{E_v \mid v \in [1, |u]\}$ to contain all of the nodes and edges in the inter-procedural control flow graph for a program.

```
1   class Testing
2   {
3       int MaxOfTwo(int x, int y)
4       {
5           int max = x;
6           if (x < y)
7           {
8               max = y;
9           }
10          return y;
11      }
12      int MaxOfThree(int x, int y, int z)
13      {
14          int max = MaxOfTwo(x, y);
15          max = MaxOfTwo(max, z);
16          return max;
17      }
18  }
```

Figure 2.2 A sample program

From the sample program in Figure 2.2, an ICFG can be created for the program which is shown in Figure 2.3.

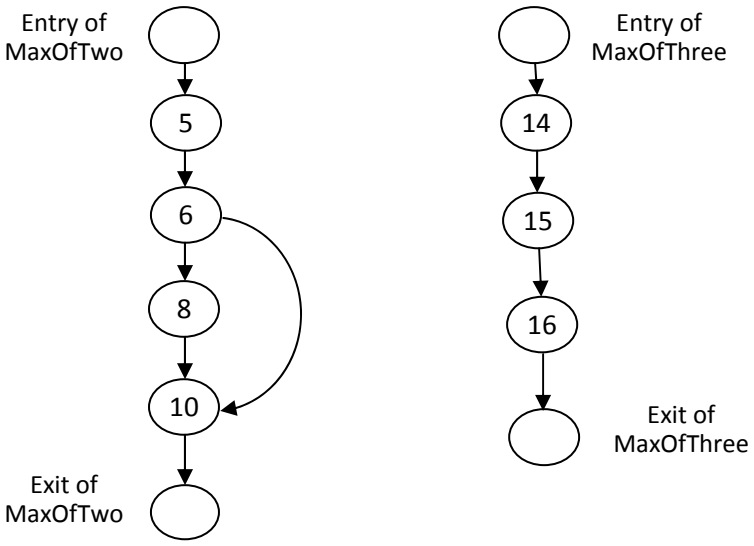


Figure 2.3 ICFG of the sample program in Figure 2.2

2.1.3 Coverage Criteria

Selection of representation of the AUT source code affects the definition of coverage criteria. For reasons of simplicity, in this section ICFG is used as source code representation.

Definition 2.4[74]: A **test suite** T for control flow graph $G_v = (N_v, E_v)$ satisfies the ***all-nodes*** test coverage criterion if and only if the tests in T create a set of complete paths Π_{Nv} that include all $n \in N_v$ at least once.

Definition 2.5[74]: A test suite T for control flow graph $G_v = (N_v, E_v)$ satisfies the ***all-edges*** test coverage criterion if and only if the tests in T create a set of complete paths Π_{Ev} that include all $e \in E_v$ at least once.

Definition 2.6[74]: A test suite T for control flow graph $G_v = (N_v, E_v)$ satisfies the ***all-paths*** test coverage criterion if and only if the tests in T create a set of complete paths Π_v that include all the execution paths beginning at the unique entry node $entry_v$ and ending at the unique exit node $exit_v$.

In a standard program, the occurrence of a variable on the left-hand side of an assignment statement is called a ***definition*** of this variable. The occurrence of a variable on the right hand side of an assignment statement is called a computation-use (or ***c-use***) of this variable. When a variable appears in the predicate of a conditional logic statement or an iteration construct, this is called a predicate-use (or ***p-use***) of the variable. A ***definition clear path*** for variable var_v is a path $<n_\rho, ..., n_\tau>$ in G_v, such that none of the nodes $n_\rho, ..., n_\tau$ contain a definition or undefinition of program variable var_v.

Def-c-use association is a triple $<n_d, n_{c\text{-}use}, var_v>$ where a definition of variable var_v occurs in node n_d and a c-use of var_v occurs in node $n_{c\text{-}use}$. ***Def-p-use*** association as the two triples $<n_d, (n_{p\text{-}use}, t), var_v>$, and $<n_{p\text{-}use}, f), var_v>$ where a definition of variable var_v occurs in node n_d and a p-use of var_v occurs during the true and false evaluations of a predicate at node $n_{p\text{-}use}$ [72, 73, 75-77].

All-du-paths coverage criterion requires the coverage of all the paths from the definition to a usage of a program variable [76, 77].

Definition 2.7[76,77]**:** A test suite T for control flow graph $G_v = (N_v, E_v)$ satisfies the ***all-c-uses*** test coverage criterion if and only if for each association $<n_d, n_{c\text{-}use}>$,

where $var_v \in U_v$ and $n_d, n_{c\text{-}use} \in N_v$, there exists a test $_{Tf} \in {}_{<T1,..,\ Te>}$ to create a complete path π^{var_v} in G_v that covers the association.

Definition 2.8[76,77]: A test suite T for control flow graph $G_v = (N_v, E_v)$ satisfies the ***all-p-uses*** test coverage criterion if and only if for each association $(n_d, (n_{p\text{-}use}, t), var_r)$ and $(n_d, (n_{p\text{-}use}, f), var_v)$,

where $var_v \in U_v$ and $n_d, n_{p\text{-}use} \in N_v$, there exists a test $T_f \in <T_1, \ldots, T_e>$ to create a complete path π^{var_v} in G_v that covers the association.

Definition 2.9[76,77]: A test suite T for control flow graph $G_v = (N_v, E_v)$ satisfies the ***all-uses*** test coverage criterion if and only if for each association $<n_d, n_{c\text{-}use}, var_v>$, $<(n_d, (_{np\text{-}use}, t), var_v>$ and $<n_d, (n_{p\text{-}use}, f), var_v>$,

where $var_v \in U_v$ and $n_d, n_{c\text{-}use}, n_{p\text{-}use} \in N_v$, there exists a test $T_f \in <T_1, \ldots, T_e>$ to create a complete path π^{var_v} in G_v that covers the association.

Definition 2.10[76,77]*:* A test suite T for control flow graph $G_v = (N_v, E_v)$ satisfies the ***all-du-paths*** test coverage criterion if and only if for each association $< n_d, n_{c\text{-}use}, var_v>$, $<n_d, (n_{p\text{-}use},t), var_v>$ and $<n_d, (n_{p\text{-}use}, f), var_v>$,

where $var_v \in U_v$ and $n_d, n_{c\text{-}use}, n_{p\text{-}use} \in N_v$, the tests in T create a set of complete paths $\Pi_v^{var_v}$ that include all of the execution paths that cover the associations.

2.1.4 Test Case Generation

The generation of test cases can be performed in a manual or automated fashion. Manual test generation involves the construction of test cases by analysing the source code and specifications. Test cases generated manually are usually more purpose intended in finding certain kind of faults. Manual test case generation involves strenuous labour as software systems are becoming more and more complex. This makes manual test cases generation infeasible. Alternatively, in some GUI based software, test cases can be "recorded" or "captured" by simply using the AUT and monitoring the actions that are taken during usage [78].

An automated solution to the test case generation problem attempts to automatically create a test suite that will fulfil selected coverage criterion when it is used to test an AUT. By using certain coverage criteria, algorithms may be developed to traverse the structured model of the AUT, such as ICFG, and generate corresponding test case suites.

2.1.5 Test Execution

The execution of a test suite can occur in a manual or automated fashion. For example, the test case descriptions that are the result of the test selection process could be manually executed against the AUT.

2.1.6 Regression Testing

Regression testing can be used to determine whether there are any changes that introduce defects after an AUT is updated for bug fixing or adding additional

functionality. Regression test suites help to ensure that the evolution of an application does not result in lower quality software. Regression testing often has a strong positive influence on software quality [79]. Regression testing is also costly. A complete regression testing of a 20,000 line software system required about seven weeks of continuous execution [70]. Selecting an appropriate subset of the existing test suite, prioritizing the execution of a regression test suite and regression test distribution help reduce the cost of regression testing by [69, 80, 81].

2.2 GUI Testing

It is estimated that an average of 48% of the application code and 50% of the time spent with implementation are dedicated to the user interface [82]. The testing phase of the software life cycle may consume around 50% of the total time of the project [83 - 85]. Checking an AUT can be performed by static or dynamic analysis. Static analysis is usually by way of code review and formal analysis such as model checking and formal proofs. This is based only on the experience and sensibility of the tester, which makes the process unsystematic, unmanageable and *ad hoc*. Dynamic analysis is performed by executing the application under test. Given that the specification is formal, the construction and execution of the test cases can be automated and the overall process becomes more systematic.

In general, conventional testing strategies are applicable to GUI testing. However, the characteristics of the GUI, such as time constraints, test case explosion problems, the

need for combining testing techniques, and test automation raises specific challenges. GUIs are becoming more and more complex, which makes manual GUI testing impractical.

2.2.1 Manual GUI Testing

Even though manual GUI testing is becoming more and more impractical, in the initial stage, it is useful to find errors from either real users or trained specialists. The bugs found can provide hints for finding other bugs, i.e., the tests can be adapted to look for bugs similar to the ones found (adaptability). Besides the real users, trained specialists can use formal methods to do the manual testing. These methods include inspection, inquiry, and usability tests [128].

Manual tests are appropriate for finding usability problems and making general assessments about usability [86], but the results/errors found by manual tests are very dependent on the capabilities of the tester. Human errors can also be injected into the results. Constructing, executing, and analysing the results of the test cases involves too much human effort. This research mainly focuses on automated GUI testing.

2.2.2 Automated GUI Testing

GUI testing represents a significant amount of the overall testing effort. To automate GUI testing, several kinds of testing tools have been developed. These tools vary from those that only support the automatic execution of test cases, to those that support test

case execution, test case generation, and construction of the GUI model by a reverse engineering process.

1) Capture/Replay (C/R)

A GUI is constituted from widgets. Each widget has some properties and related events. This information about GUI components is programmatically readable. By hooking the system event handling, actions taken by a user can be captured and the corresponding information about the GUI widgets can be read. The captured sequences of the actions can be replayed many times on the same GUI of an AUT. Some tools, such as WinRunner [20] and Rational Robot [22], have been developed for this purpose. Test scripts can be constructed by interacting with the AUT but capture/replay tools give no support for their design and coverage criteria analysis. The lack of structure of the scripts makes their maintenance very difficult. Some researchers have tried to solve the problems by the adoption of methodologies that entail more structure in the test scripts [87-91].

C/R technology has many advantages in other applications such as demonstrations, remote support, analysis of user behaviour, macro functionality, and educational scenarios. However, for testing purposes, it is still subject to severe criticism. The disadvantages of C/R in GUI testing are

- C/R tools can be used only when the GUI, or part of it, is already available.

- The whole process of a test case needs to be re-captured if a mistake or a failure happens in the middle of the capture. All that is being tested are things that already work [92].
- Test case design and evaluation are not supported according to coverage criteria.
- Minor changes to the implementation usually require the re-capturing of all affected test scripts.
- Low level of abstraction, such as mouse positions, may be hard coded in generated scripts. A small change on the layout of the user interface might invalidate all test cases.

2) Random Input Testing

Generating test cases is difficult for GUI testing. In the early era of GUI testing research, inputs were generated randomly for crash testing. Random input testing is also referred to as monkey testing [93]. Mouse movements, clicks and keys are randomly generated and performed on the GUI. Microsoft reported that 10-20% of the bugs in their software projects are found by a monkey test tool [94]. Besides finding defects which crash the system, this method cannot even recognize a software error without knowledge of the system, which makes it not particularly useful.

The coverage of random input testing is very weak. Due to the huge space of the input domain, important actions can be selected with very low probability. Although some

errors can be found by this approach, it is rather arbitrary and does not provide reliable coverage criteria [95].

3) Unit Testing Framework

By using certain framework such as JUnit [96] and NUnit [97], test cases can be constructed / programmed manually with high-level flexibility. This unit testing is particularly suitable for API testing, but not for GUI testing. Strenuous labour of testers is involved to adequately test GUI behaviour. In unit testing, because the test action sequences are usually written manually, the sequences tend to be too short to uncover bugs which need long particular sequences of actions. Thus, these kinds of errors are very likely to be missed. There are some GUI libraries, such as Abbot [17], or Jemmy [127], which provides methods to simulate user actions, but GUI testing still requires a lot of extra programming effort to be effective.

4) Model based GUI Testing

Model-based GUI testing tools normally focus on the GUI testing automation process. To test GUI automatically, the GUI states and events are usually described with certain kinds of model. With the models, test cases can be generated automatically to some extent. The generation of test cases can be either random or according to certain coverage criteria. Test cases execution and output checking can also be automatic to some extent. Automatic test cases and oracle generation usually needs knowledge

models to take advantage of the information in AUT's specifications. The next section will introduce some popular existing GUI testing models.

2.3 Existing GUI testing models

2.3.1 Event Sequence Graph

Belli et al [33] proposed an event sequence graph (ESG) model, and introduced decision tables to refine a node of the ESG where the test cases are generated according to the rules of the decision table.

Definition 2.11[98]: An event sequence graph $ESG = (V, E, I, \Gamma)$ is a directed graph, where $V \neq \emptyset$ is a finite set of vertices (nodes), $E \subseteq V \times V$ is a finite set of arcs (edges), $I, \Gamma \subseteq V$ are finite sets of distinguished vertices with $\xi \in I$, and $\gamma \in \Gamma$, called entry nodes and exit nodes, respectively, wherein $\forall v \in V$ there is at least one sequence of vertices $<\xi, v_0, \ldots, v_k>$ from each $\xi \in I$ to $v_k = v$ and one sequence of vertices $<v_0, \ldots, v_k, \gamma>$ from $v_0 = v$ to each $\gamma \in \Gamma$ with $(v_i, v_{i+1}) \in E$, for $i = 0, \ldots, k-1$ and $v \neq \xi, \gamma$.

$I(ESG), \Gamma(ESG)$ represent the entry nodes and exit nodes of a given ESG, respectively. To mark the entry and exit of an ESG, all $\xi \in I$ are preceded by a pseudo vertex '$\lceil$' $\notin V$ and all $\gamma \in \Gamma$ are followed by another pseudo vertex '$\rfloor$' $\notin V$. The semantics of an ESG is as follows: any $v \in V$ represents an event. For two events $v, v' \in V$, the event v' must be enabled after the execution of v *iff* $(v, v') \in E$. The operations on identifiable components of the GUI are controlled and/or perceived by input/output devices, i.e.,

elements of windows, buttons, lists, checkboxes, etc. Thus, an event can be a user input or a system response; both of them are elements of *V* and lead interactively to a succession of user inputs and expected desirable system outputs. To illustrate the model, RealJukebox (RJB) has been selected, more precisely the basic, English version of RJB 2 (Build: 1.0.2.340) of RealNetworks. Figure 2.4 shows the GUI of RJB, and Figure 2.5 is an example of ESG representation of RJB. Table 2.1 shows the RJB system functions as responsibilities of the system to interact with the user.

Figure 2.4 GUI of Real Juke Box [56]

Table 2.1 Real Juke Box System functions [56]

1. Play and record a CD or track 2. Create and play a playlist 3. Edit playlists and/or autoplaylists 4. View lists and/or tracks 5. Edit a track 6. Visit the sites	7. Visualization 8. Skins 9. Screen sizes 10. Different views of 11. Find music 12. Configure RJB

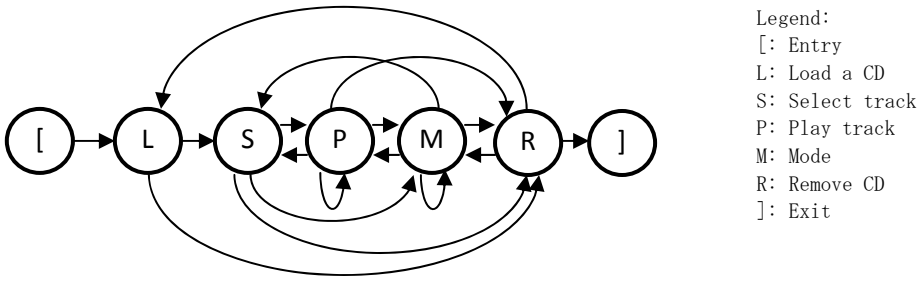


Figure 2.5 ESG representation of Play and Record a CD system function [56]

Definition 2.12[98]**:** Let *V, E* be defined as in Definition 2.11. Then any sequence of vertices $<v_0,...,v_k>$ is called an *event sequence (ES) iff* $(v_i,v_{i+1}) \in E$, for $i=0,...,k-1$. Moreover, an ES is *complete* (or, it is called a *complete event sequence, CES*), *iff* $v_0 \in I$ and $v_k \in \Gamma$.

Note that the pseudo vertices *'[', ']'* are not included in ESs. An $ES = <v_i,v_k>$ of length 2 is called an event pair (EP). A CES may invoke no interim system responses during user-system interaction, i.e., it may consist of consecutive user inputs and a final system response.

Graphically speaking, missing edges of the ESG represent undesirable user-system interactions, i.e., faulty event pairs (FEP). FEPs can systematically be constructed by either (1) adding arcs in the opposite direction wherever only one-way arcs exist, or (2) adding two-way arcs between vertices wherever no arcs connect them, or finally, (3) adding self-loops to vertices wherever none exist.

Definition 2.13[98]: Let $ES = \langle v_0,\ldots,v_k\rangle$ be an event sequence of length $k+1$ of an *ESG* and $FEP = \langle v_k,v_m\rangle$ a faulty event pair. The concatenation of the *ES* and *FEP* then forms a faulty event sequence $FES = \langle v_0,\ldots,v_k,v_m\rangle$. *FES* is complete (or, it is called a faulty complete event sequence, FCES) *iff* $v_0 \in I$. The *ES* as part of a *FCES* is called a *starter*.

According to Definition 2.13, the red doted lines shown in figure 2.6 are FEPs, *CES* and *FCES* form test cases to the *SUC*. The *SUC* is supposed to accept test inputs described by *CESs* in the specified order whereas test inputs described by *FCESs* should result in a warning.

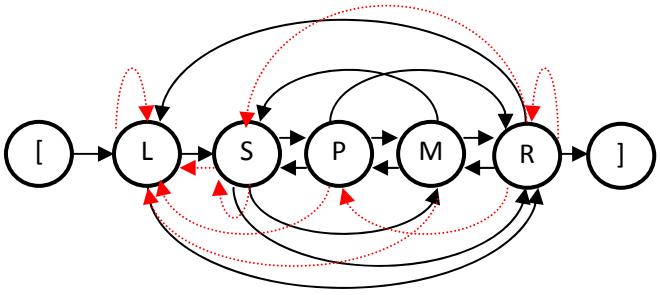
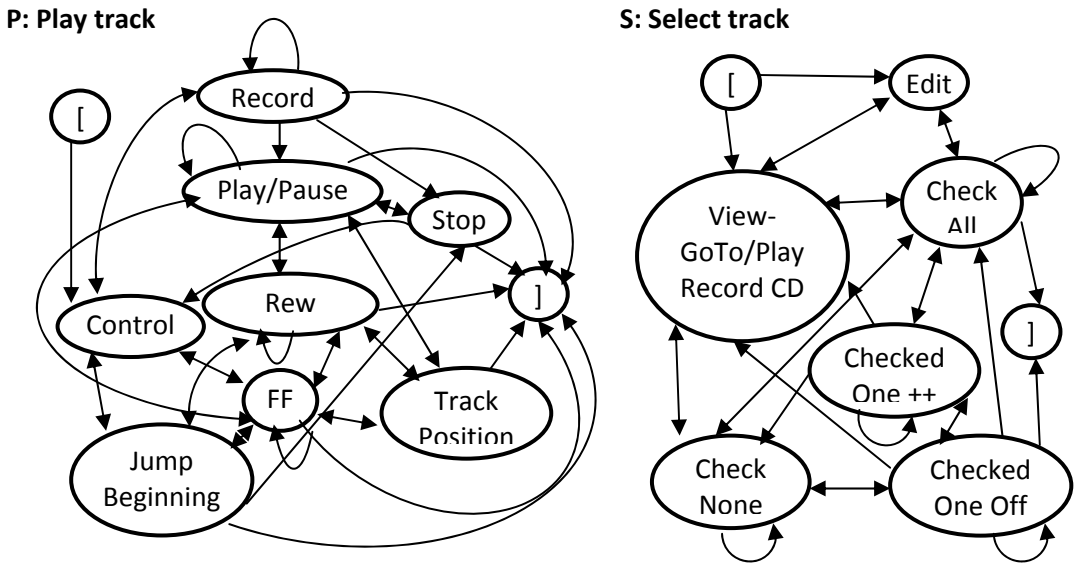


Figure 2.6 ESG with FP

Modelling input data, especially concerning causal dependencies between each other as additional nodes, inflates the ESG model. To avoid this, decision tables are introduced to refine a node of the ESG. Such refined nodes are double-circled.

Definition 2.14[56]: Given an *ESG*, say $ESG_1 = (V_1,E_1)$, a vertex $v \in V_1$, and an *ESG*, say $ESG_2 = (V_2,E_2)$, then replacing v by ESG_2 produces a refinement of ESG_1, say $ESG_3 = (V_3,E_3)$ with $V_3 = V_1 \cup V_2 \setminus \{v\}$, and $E_3 = E_1 \cup E_2 \cup E_{pre} \cup E_{post} \setminus E_{1replaced}$ (\ is the set difference operation), wherein $E_{pre} = N^-(v) \times I(ESG_2)$ (connections of the predecessors of v with the entry nodes of ESG_2), $E_{post} = \Gamma(ESG_2) \times N^+(v)$ (connections of exit nodes of ESG_2 with the successors of v), and $E_{1replaced} = \{(v_i, v), (v, v_k)\}$ with $v_i \in N^-(v)$ and $v_k \in N^+(v)$ (replaced arcs of ESG_1).

Figure 2.7 shows the refinement of the vertices S, P, and M of the ESG in Figure 2.5



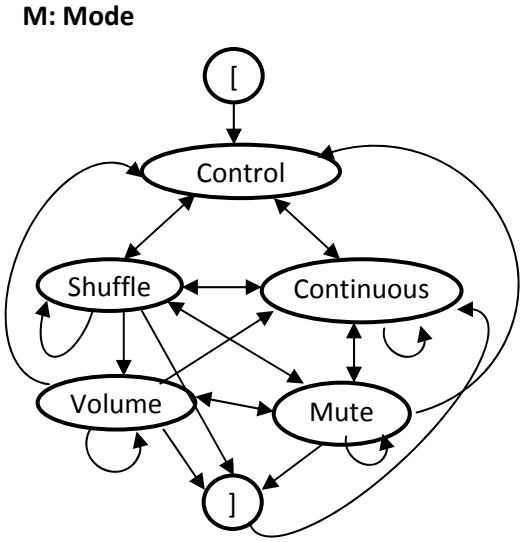


Figure 2.7 Refinement of the vertices S, P, and M of the ESG in Figure 2.4 [56]

Definition 2.15[98]: A Decision Table *DT = {C, A, R}* represents actions that depend on certain constraints where:

- $C \neq \emptyset$ is the set of constraints
- $A \neq \emptyset$ is the set of actions
- $R \neq \emptyset$ is the set of rules that describe executable actions depending on a certain combination of constraints.

Decision tables [99] are popular in information processing and are also used for testing, e.g., in cause and effect graphs. A decision table logically links conditions (”if”) with actions (”then”) that are to be triggered, depending on combinations of conditions (”rules”) [100].

Definition 2.16[98]: Let R be defined as in Definition 15. Then a *rule* $R_i \in R$ is defined as $R_i = (C_{True}, C_{False}, A_x)$ where:

- $C_{True} \subseteq C$ is the set of constraints that have to be resolved to be "true"
- $C_{False} = C \backslash C_{True}$ is the set of constraints that have to be resolved to be "false"
- $A_x \subseteq A$ is the set of actions that should be executable if all constraints $t \in C_{True}$ are resolved to be "true" and all constraints $f \in C_{False}$ are resolved to be "false" .

Note that $C_{True} \cup C_{False} = C$ and $C_{True} \cap C_{False} = \emptyset$ under regular circumstances. In certain cases it is inevitable to remark conditions with a don't care (symbolized with a '-' in *DT*), i.e., such a condition is not to be considered in a rule and $C_{True} \cup C_{False} \subset C$. A *DT* is used to refine data input of GUI's.

The most important contribution of the ESG to GUI testing is that it takes into account not only the desirable behaviour, but also the undesirable behaviour of a GUI. That is to say, it tests GUIs not only through exercising them by means of test cases which show that the GUI is working properly during routine operation, but also exercising potentially illegal events to verify that the GUI behaves satisfactorily in exceptional situations. However, the ESG model still faces a number of limitations for real GUI automation. The major limitations of the ESG model include:

- Model is manually created by analysing the specifications and source code, which involves enormous labour;

- States explosion. With the algorithm proposed in the ESG model, the vertices and states may increase drastically, especially when taking into account of concurrency;
- Procuring test oracle information involves intensive labour;
- Unable to model events which have uncertain follow-ups events.

2.3.2 Event Flow Graph (EFG) and Event-Interaction Graph (EIG)

The event flow graph (EFG), and its later version, the event interaction graph (EIG) were recently proposed by the team of Professor Atif M. Memon in the University of Maryland [37, 43, 44, 47]. The EFG-based test automation [37] has been claimed to be the first practical GUI automated smoke test. This was followed by research on automated black-box GUI testing. In these researches, the event flow graph (EFG) was proposed as the core-enabling model. In the EFG, each vertex represents an event. All events which can be executed immediately after this event are connected with directed edges from it. A path in the EFG is a legal executable sequence which can be seen as a test case. EFGs can be generated automatically using a tool called GUI Ripping [44]. Traversing an EFG with a certain strategy can generate test cases.

The EFG was first proposed in 2001 [43]. The definition of the EFG is as follows.

Definition 2.17 [37]: An *event-flow graph* for a component C is a quadruple $<V, E, B, I>$ where:

1. V is a set of vertices representing all the events in the component. Each $v \in V$ represents an event in C;

2. $E \subseteq V \times V$ is a set of directed edges between vertices. Event e_i follows e_j *iff* e_j may be performed immediately after e_i. An edge $(v_x, v_y) \in E$ *iff* the event represented by v_y follows the event represented by v_x ;

3. $B \subseteq V$ is a set of vertices representing those events of C that are available to the user when the component is firstly invoked; and

4. $I \subseteq V$ is the set of restricted-focus events of the component.

In the definition, a GUI component C is an ordered pair $<RF, UF>$, where RF represents a model window in terms of its events and UF is a set whose elements represent modeless windows also in terms of their events. Each element of UF is invoked either by an event in UF or RF. Figure 2.8 shows an example of an EFG for Notepad.

To generate the test cases automatically, events are classified into 5 groups:

1. Restricted-focus events open modal windows;
2. Unrestricted-focus events open modeless windows;
3. Termination events close modal windows;
4. Menu-open events are used to open menus; and

5. System-interaction events interact with the underlying software to perform some actions.

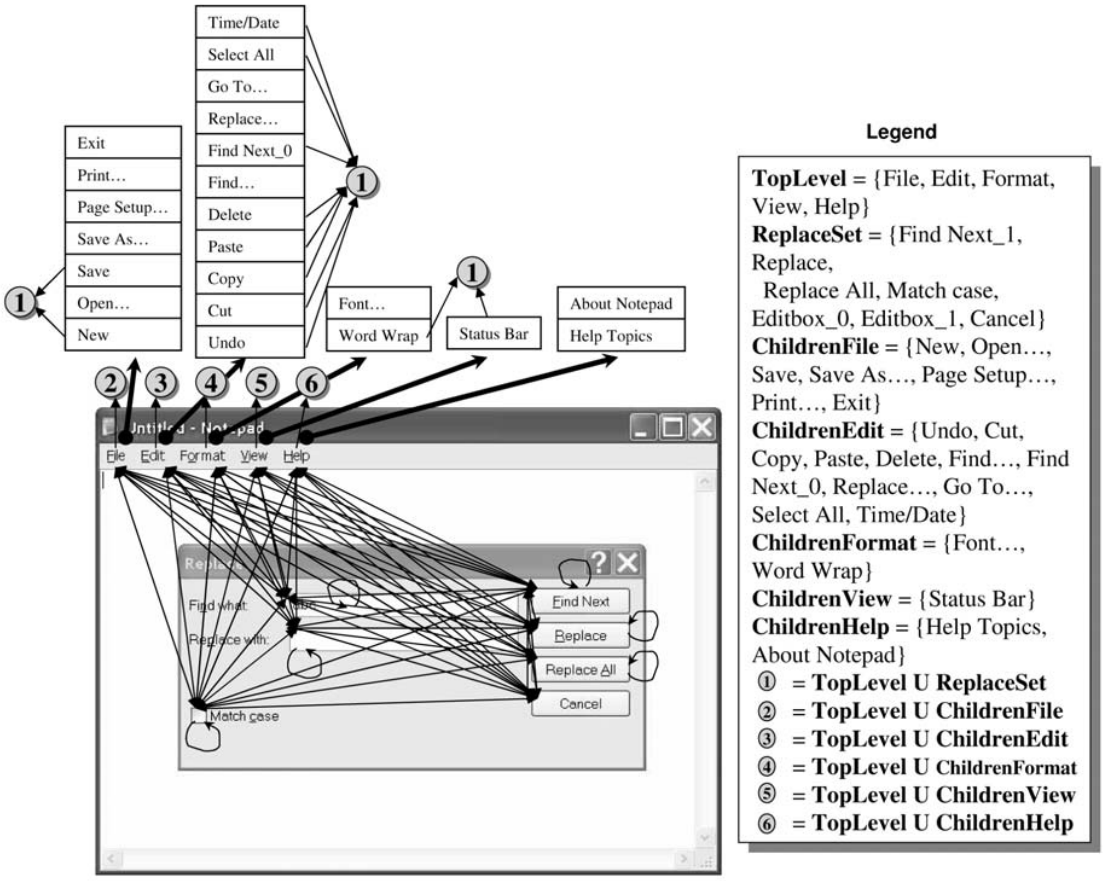


Figure 2.8 An example of an EFG [37]

Algorithm GetFollows [43].

```
1. GetFollows( v: Vertex or Event)
2. {
3.     if(EventType(v)=menu-open ){
4.         if v ∈ B of the component that contains v
5.             return ( MenuChoices(v) ∪{v} ∪B);
```

```
6.          else
7.              Return
8.                  (MenuChoices(v) ∪{v} ∪B  ∪(Parent(v)));
9.  }
10. if(EventType(v)=system-interaction)  return (B);
11. if( EventType(v)=termination)
12.       return (B of Invoked component);
13. if (EventType(v)=unrestricted-focus)
14.       return   (B ∪ B of Invoked Model Dialogue)
15. if (EventType(v)=restricted-focus)
16.       return (B of Invoked component);
17. }
```

Figure 2.9 Algorithm GetFollows

To create an EFG automatically, finding the follow-up events of each event is critical. This can be done using an algorithm called GetFollows [37]. Figure 2.9 shows the algorithm GetFollows.

The set of follows (v) can be determined using the algorithm GetFollows for each vertex v. The recursive algorithm contains a switch structure that assigns follows (v) according to the type of each event. If the type of the event v is a menu-open event (line 3) and v $\in$ B (recall that B represents events that are available when a component is invoked) then the user may either perform v again, its sub-menu choices, or any event in B (line 5). However, if $v \in B$ then the user may either perform all sub-menu choices of v, v itself, or all events in follows *(parent (v))* (line 8); *parent (v)* is defined as any event that makes v available. If v is a system-interaction event, then after

performing *v*, the GUI reverts to the events in *B* (line 10). If *v* is a termination event, i.e., an event that terminates a component, then follows (*v*) consists of all the top-level events of the invoking component (line 12). If the event type of *v* is an unrestricted-focus event, then the available events are all top-level events of the invoked component available as well as all events of the invoking component (line 14). Lastly, if *v* is a restricted-focus event, then only the events of the invoked component are available.

Since an EFG models all possible event interactions, it cannot be used directly for rapid testing. To effectively generate test cases, a new event-interaction graph (EIG) was introduced in 2005 [37]. System-interaction events are those that interact with the underlying software, including non-structural events and those that close windows. In EIG, only system interaction events are selected. An EIG can be transferred from an EFG.

Definition 2.18: There is an *event-flow-path* from node n_x to node n_y *iff* there exists a (possibly empty) sequence of nodes $n_j; n_{j+1}; n_{j+2}; \ldots; n_{j+k}$ in the event-flow graph *E* such that $\{ (n_x, n_j), (n_{j+k}, n_y)\} \subseteq edges(E)$ and $\{(n_{j+i}, n_{j+i+1}) \text{ for } 0 \leq i \leq (k-1) \} \subseteq edges(E)$ [37].

Definition 2.19: An event-flow-path $< n_1; n_2; \ldots; n_k >$ is *interaction- free iff* none of $n_2; \ldots; n_{k\_1}$ represent system-interaction events. [37]

Definition 2.20: A system-interaction event e_x *interacts-with* system-interaction event e_y *iff* there is at least one interaction-free event-flow-path from the node n_x (that represents e_x) to the node n_y (that represents e_y). [37]

The *interacts-with* relationship is used to create the EIG which contains nodes, one for each *system-interaction event* in the GUI. An edge in EIG from node n_x (that represents e_x) to node n_y (that represents e_y) means that e_x *interacts-with* e_y.

An EIG can be converted from an EFG. Algorithm GenerateEIG in Figure 2.10 is used to convert an EFG to an EIG.

Algorithm GenerateEIG [37]

```
1.  Ṅ /* Nodes set of EIG */
2.  Ê /* Edges set of EIG */
3.  GenerateEIG( EFG(N,E))
4.  {
5.    Ṅ = N;
6.    Ê = E;
7.    For all n ϵ N
8.    {
9.      Start(n) = { n_i | (n, n_i) ϵ E, and n ≠ n_i }
10.     End(n)={n_i | (n_i, n) ϵ E, and n ≠ n_i }
11.   }
12.   For all n ϵ N
13.   {
14.     If( EventType(n) ≠ System-interaction)
15.     {
16.       For all n_x ϵ end(n)
17.         For all n_y ϵ start(n)
18.         {
19.           Ê = Ê ∪ (n_x, n_y)
20.           If (n_x ≠ n_y)
21.           {
22.             Start(n_x) = start (n_x) ∪ {n_y}
23.             End(n_y)=end(n_y) ∪ {n_x}
24.           }
```

```
25.              }
26.           For all n_x ϵ end(n)
27.              Remove n from start(n_x)
28.           For all n_y ϵ start(y)
29.              Remove n from end(n_y)
30.           Remove n from Ṅ
31.           Remove all edges (n, n_i) from Ė
32.           Remove all edges(n_i, n) from Ė
33.        }
34.     }
35. }
```

Figure 2.10 Algorithm GenerateEIG

The algorithm GenerateEIG takes as input an *EFG*, represented as a set of nodes *N* and a set of edges *E*. It removes all non-system-interaction event nodes and their associated edges from the given *EFG*. At the termination of the procedure, the event-interaction graph is obtained, represented as a set of nodes $\dot{N}$ and a set of edges $\dot{E}$. $\dot{N}$ *and* $\dot{E}$ are initialized to *N* and *E* (lines 5-6). When traversing all edges of the *EFG*, a list of nodes *start (n)* on the edges that start from the node *n* (except itself) is obtained for all nodes. Similarly, a list of nodes *end (n)* that end with the node *n* (except itself) for all nodes (lines 9-10) is computed. For each node *n* of the EFG (line 12), all new *edges* (n_x, n_y) are added to $\dot{E}$ if there is an interaction-free path $< n_x; n; n_y >$ in the EFG (lines 14-19); *start*(n_x) and *end*(n_y) are updated to add n_y and n_x in the lists, respectively, if n_x and n_y are not the same node (lines 20-23). Accordingly, *n* is removed from the start and end lists (lines 26-29). Finally, *n* is removed from $\dot{N}$ (line 30); all edges associated with *n* are removed from $\dot{E}$ (lines 31-32). The space of event sequences in EIG can be reduced considerably since only the system interaction event interactions are modelled in this graph. Figure 2.11 is the EIG converted from the EFG in Figure 2.8.

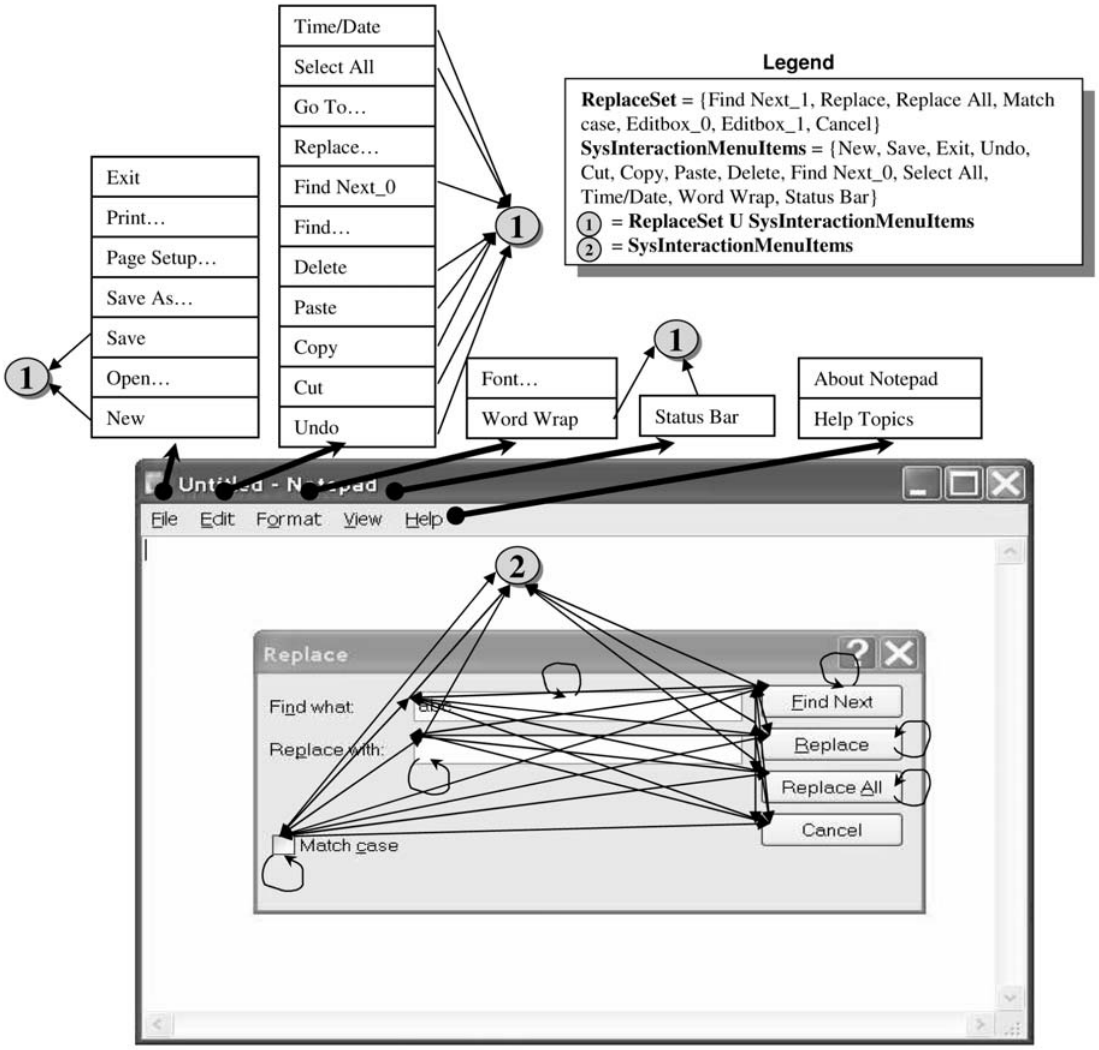


Figure 2.11 EIG for the EFG in Figure 2.7 [37]

The EFG model is so far the most popular and practical GUI testing automation model. The research team led by Professor Atif M. Memon also provided a unified solution to GUI testing automation. The solution includes automatic EFG generation with a tool called GUI Ripping, architecture of smoking test which is called Daily Automated

Regression Tester (DART), and Automatic test oracle generation in regression testing. However, the EFG model still faces a number of limitations. These limitations include:

- The EFG cannot model all GUI behaviours. In an EFG, one event has a fixed set of follow-up events. In fact, very commonly, many events such as button clicks may have uncertain follow-up events when the ambient conditions change.
- To avoid the explosion of test cases, test case generation with EFG usually reduces the number of test cases by limiting the length of each test case.
- Lack of user's knowledge makes long test cases generation impractical.
- The EFG model focuses on events instead of GUI states, which limits the ability of characterizing the full feature of GUIs.

2.4 Conclusion

This chapter has presented an overview of the research serving as the foundation for some of the concepts developed in this thesis. Conventional testing solutions do not fully suit GUI based software testing, but can be extended to GUI testing environments. The code coverage criterion of generating test cases and adequacy evaluation can be extended to events and the states coverage criterion in GUIs. Although test case execution in GUI testing is quite different from in conventional software testing, new GUI operation tools help to perform test cases on GUIs. These tools provide basic operation functions such as reading widget information, recording user actions,

performing mouse clicks, etc. To automatically test GUI based software, various models are used to model the GUI events or states so that test cases generation, oracle information generation, test case execution and evaluation can be automated to some degree. Among the existing models, the event sequence graph (ESG) proposed by Belli and his team [6, 33, 34, 56, 57, 100, 101,102], event flow graph (EFG) proposed by Memon and his team are most popular for GUI testing automation. However, an ESG needs to be generated from an AUT's specification and the generation of the model involves too much human effort. The EFG is the most practical GUI automation model so far. An EFG can be generated automatically by a tool called GUI Ripping [44]. Both the ESG and EFG models provide basic GUI testing automation, but because the models are all entirely focused on events rather than on GUI states, especially that each event is represented as a node and all the follow-ups are connected to it with directed edges, they are not able to model scenarios in which one event may have uncertain follow-ups.

This thesis will present a GUI testing automation model which models both the GUI states and events and provides a unified solution to GUI testing automation. Details are to be presented in the next chapters.

Chapter 3

Graphic User Interface Testing Automation Model

GUIs are very different from conventional command-based interfaces. They present a much more complex structure and more complex event-driven behaviour. GUI testing is laborious, boring, and time and resource consuming. The approaches and tools available to aid the testing process are not satisfactory. As compared to the conventional source code modelling, a proper representation of GUI is needed to model its behaviour.

3.1 What is a GUI?

Instead of using a command line, most of today's software comes with a graphic user interface which users can interact with. A GUI uses a collection of objects (widgets) which are familiar from real life such as, buttons and menus items which make the system more user-friendly. These objects include elements such as windows, pull-down menus, buttons, scroll bars, iconic images, and wizards. Instead of using a key board, as with command line software, GUIs support point devices, such as mouses and touch screens, which allow users to operate any part of a window on the screen in any sequence. Software users can perform tasks by manipulating GUI objects as they do in the real world. Dragging an item, discarding an object by dropping it in a trash-

can, and selecting items from a menu are all familiar actions available in today's GUI. Figure 3.1 shows a set of GUIs of simplified clinic software.

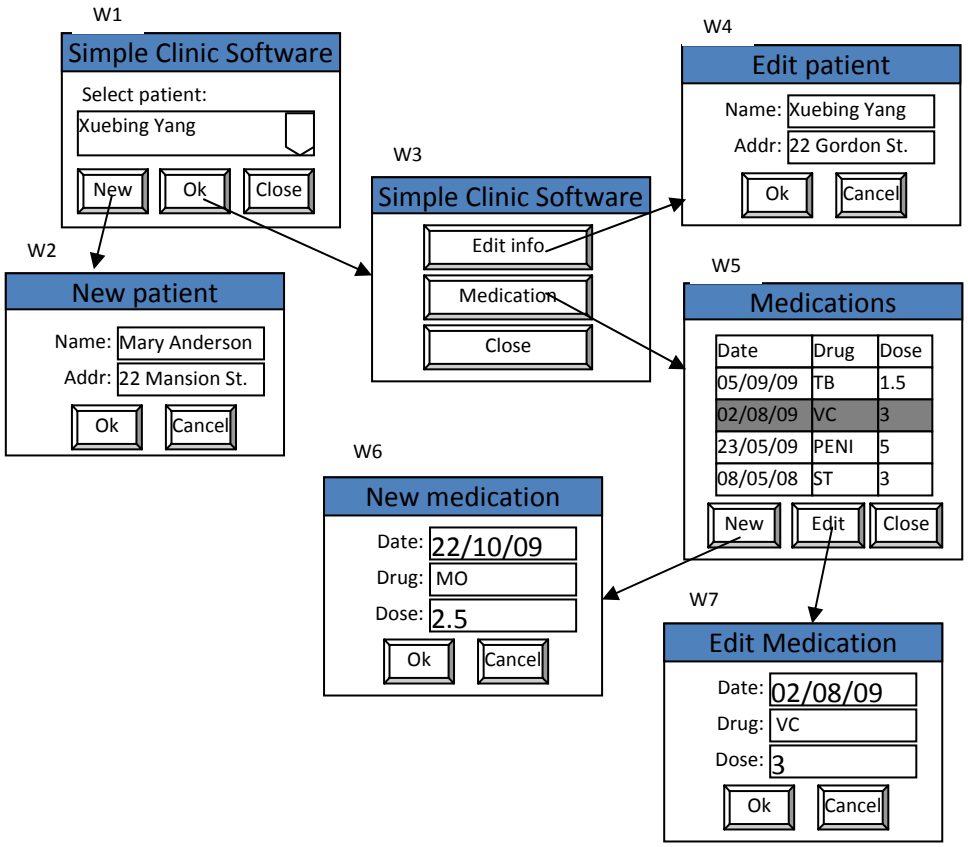


Figure 3.1 GUIs of Simple Clinic Software

The actions performed by users are called events. These events cause deterministic changes to the state of the software that may be reflected by a change in the appearance of one or more GUI objects. For example, in Figure 3.1, clicking on the 'New' button in window w_1 will lead to the opening of window w_2; When w_2 is open,

and the 'Name' Editbox is focused, typing a string will cause the text to change in the edit area of the object.

At any given moment, a GUI is a certain collection of windows and objects built in the windows. Windows and their objects are hierarchically organized. Each object has its own collection of properties, exposes a collection of events and owns a collection of sub-objects. Property values of an object constitute a state of the object. All states of objects constitute the state of the GUI. Figure 3.2 shows the hierarchically organized objects of w_1 in Figure 3.1 and some of their property values.

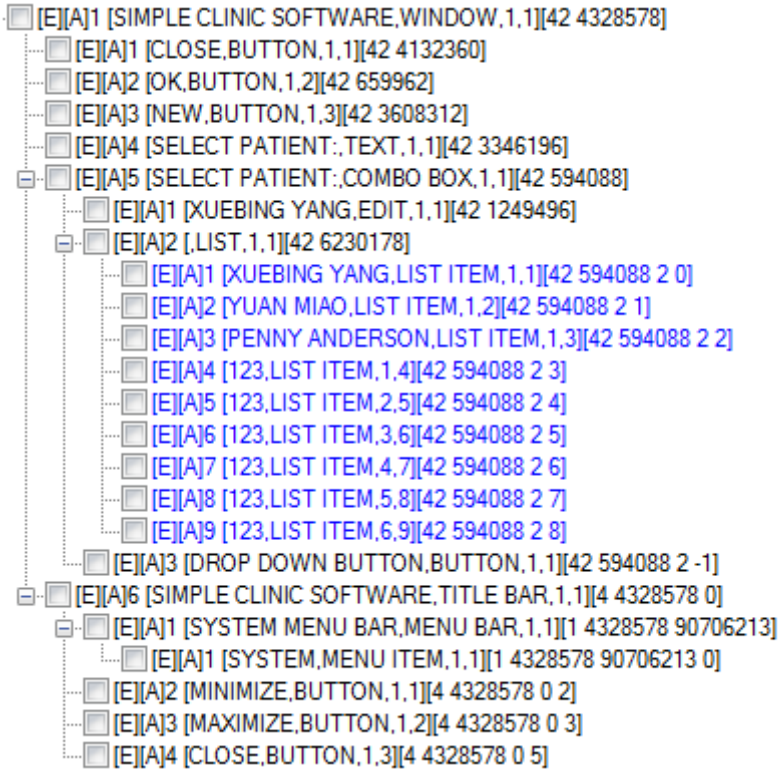


Figure 3.2 Hierarchical objects of w_1 in Figure 3.1

The important characteristics of GUIs include their graphical orientation, event driven input, hierarchical structure, the objects they contain, and the properties (attributes) of those objects. Formally, a GUI can be defined as follows:

Definition 3.1: A *Graphical User Interface (GUI)* is a hierarchical, graphical front-end to a software system that accepts as input user-generated and system-generated events which are from a fixed set of events, and produces deterministic graphical output. A GUI contains graphical objects; each object has a fixed set of properties. At any time during the execution of the software, these properties have discrete values, the set of which constitutes the state of the GUI.

The above definition specifies a class of GUI which have a fixed set of events with deterministic outcomes which can be performed on objects with discrete valued properties.

3.2 GUI states

There are many ways to define the states of a GUI application. To facilitate GUI testing automation, we shall focus on GUI-related state and state transitions. The graphical user interface of a given application is treated as a series of interfaces. Each interface may be regarded as a state. The states are to be used to construct a finite state machine for GUI test automation.

Definition 3.2: A ***GUI state*** is modelled as a set of opened windows and the set of objects (labels, buttons, edits, etc.) contained in each window. Hence, at a particular

time *t*, the GUI can be represented by its constituent windows $W = \{w_1, w_2, \ldots, w_n\}$ and their objects $O = \{ O_1, O_2, \ldots, O_n \}$,

where

$O_i = \{o(i,1), o(i,2), \ldots o(i, m_i)\}$, $i = 1, 2, \ldots, n$;

Each object contains properties

$P = \{ P(1,1), P(1,2), \ldots, P(1, m_1),$

$P(2,1), P(2,2), \ldots, P(2, m_2),$

….,

$P(n,1), P(n,2), \ldots., P(n, m_n)\}$,

where

$P(i,j) = \{ p(i,j,1), p(i,j,2), \ldots., p(i,j, k_{ij}) \}$; $i = 1, 2, \ldots, n;$ $j = 1, 2, \ldots., m_i$;

and their corresponding values $V(i,j) = \{ V(i,j,1), V(i,j,2), \ldots., V(i,j, k_{ij}) \};$

where

$V(i,j,k) = \{ v(i,j,k,1) , v(i,j,k,2) , \ldots, v(i,j,k, L_{ijk}) ,\},$

i=1, 2, …, n;

j= 1, 2, …., m_i ;

k= 1, 2, …., k_{ij} ;

At a certain time *t*, the set of windows and their objects constitute the state of the GUI. All the objects are organized as a forest. A GUI state is then modelled as a quadruple *(W,O, P, V)*. Events $\{e_1, e_2 \ldots e_q\}$ performed on the GUI may lead to state transitions. The function notation $S_j = e_i(S_i)$ is used to denote that S_j is the state resulting from the execution of event e_i at state S_i. Such a state and transition can be considered as a finite state machine. However, such an FSM would contain too many states and transitions which would make the test automation impractical. Traditional FSM differentiates all the minor changes of the states. It leads to exponential increased number of GUI states. Figure 3.3 shows a GUI editing interface in Medical Director 3 [110], of a type which is very common employed in many other applications. In Figure 3.3, there are 15 radio buttons and two editboxes. Even we confine ourselves to the radio buttons, each can have one of two statuses (checked or unchecked). This simple form can be 2^{15} = 32768 GUI states. If combined with other interfaces, the number of statuses for the whole application will be prodigious. Therefore, the test cases for full permutation of all the states can easily require over a million years to execute. However, because none of the value changes of the radio buttons will affect other GUI states, there is no need to see the different sets of values of this form as different states. To tackle this problem, in the next section a GUI Testing Automation Model is proposed, within which object and property selection criteria are used to differentiate GUI states in order that the number of states can be drastically reduced.

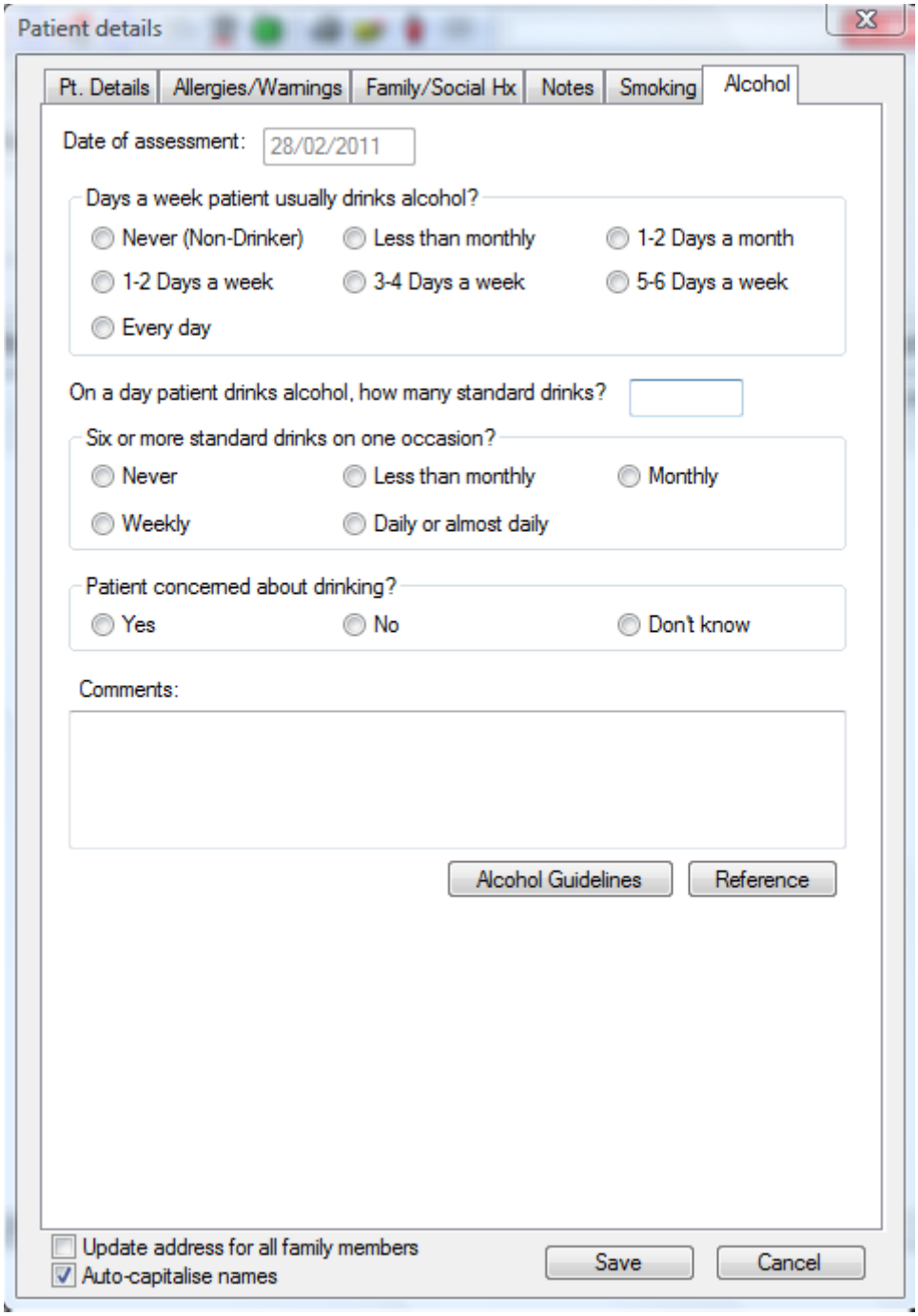


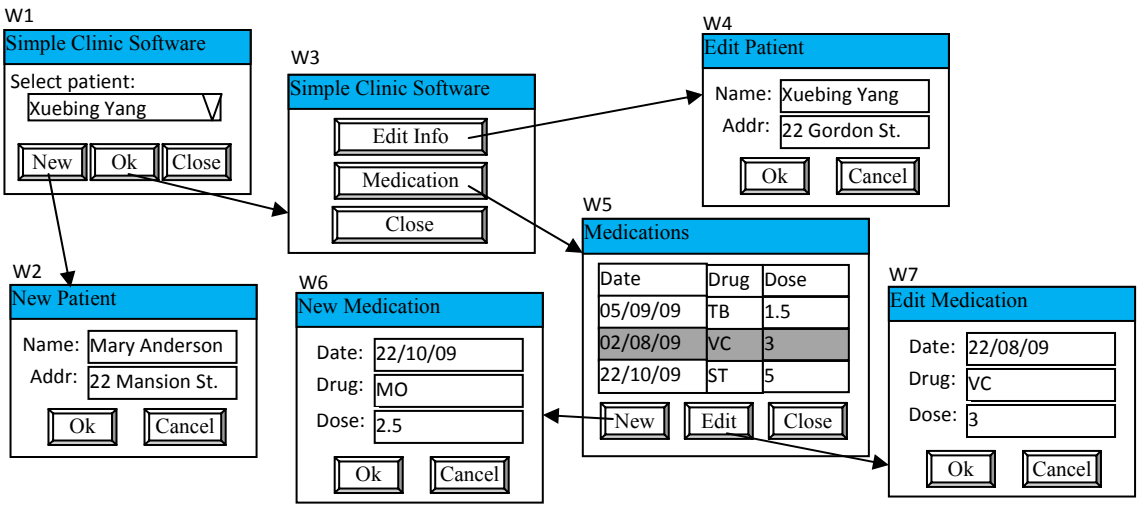
Figure 3.3 Patient details editing GUI in Medical Director 3

3.3 GUI Testing Automation Model

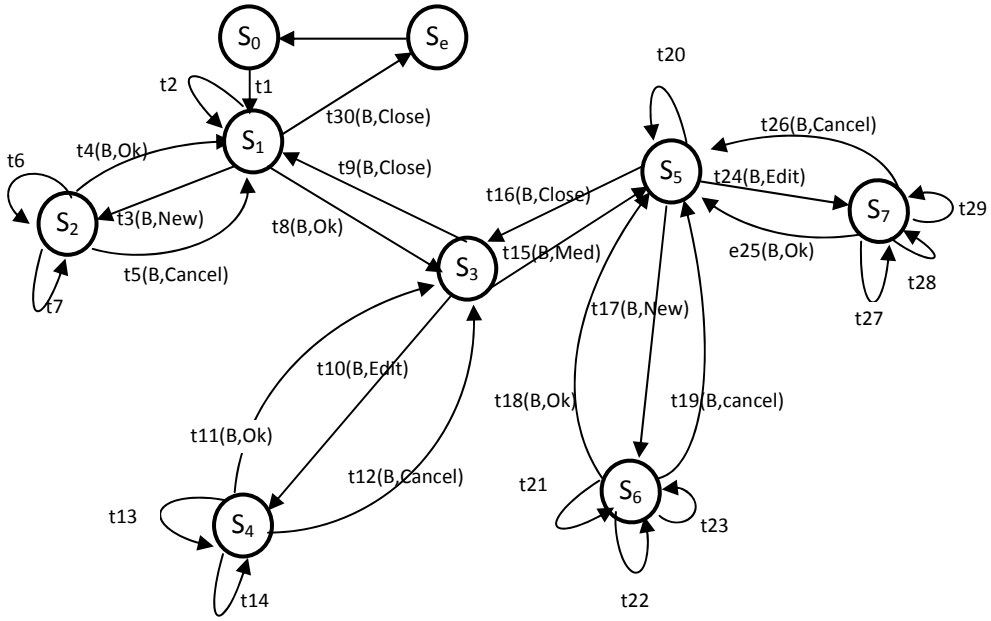
Definition 3.3: A GUI *Testing Automation Model (GUITAM)* is a quadruple *(Σ, S, s_0, T),* where:

- Σ is a finite, non-empty set of all possible input events of the application under test (AUT),
- S is a finite, non-empty set of GUI states of the AUT,
- s_0 is an initial state, an element of S, and
- T is the state-transition function set: $T: S \times \Sigma \rightarrow S$.

In this model, all possible events of an AUT constitute the input set Σ, and s_0 is the first state of the AUT when it is invoked. Sometimes, use s_0 to denote the state when an application is not started yet (Before Start). S is composed of all possible states of the AUT. For each $s \in S$ is a tuple $<Q,E>$, where Q is the GUI state *(W,O, P, V)* as defined in definition 3.2 and E is the set of possible events in this state, $E \subseteq \Sigma$. For each $t \in T$, $s'=t(s, \hat{e})$, where s is the current state, s' is the next state, $\hat{e}$ is either an event or a small set of elementary events in s. $\hat{e}$ is also called an operator. We can also simply use $<s,s', \hat{e}>$ to describe a transition. By performing event $\hat{e}$ on s, the state will be changed to s'. It is possible for $s' \neq s$, and also for $s'=s$. Figure 3.4 (b) shows the GUITAM states of the simple clinic software shown in Figure 3.1 (or Figure 3.4 (a)). In Figure 3.4 (b), many unimportant events are ignored to ensure simplicity.



(a)



(b)

Figure 3.4 GUITAM states

In Figure 3.4 (b), s_0 stands for the state before the application is started, s_e stands for the state after the application is quitted. Actually, in this case, s_e is the same state as s_0. Each other state stands for a set of windows. Each window has its objects (see Figure 3.2). Events such as clicks on a button may lead to a state transition but some may not. For example, if the content of the edit box is not used for distinguishing states, typing a string in an edit box doesn't cause a state transition. t_6 and t_7 in Figure 3.4 just lead to transition to the same state. Table 3.2 shows the events and states transition information. Table 3.1 shows the states and their corresponding windows. Because the simple clinic software is just a simplified dialog-based demonstration application, when a dialog opens, the window or dialog that opens it is still open.

Table 3.1 GUITAM states and their corresponding windows in simple clinic software

State	Open windows
S_0	
S_1	w_1
S_2	w_1, w_2
S_3	w_1,w_3
S_4	w_1,w_3,w_4
S_5	w_1,w_3,w_5
S_6	w_1,w_3,w_5,w_6
S_7	w_1,w_3,w_5,w_7
S_8	

Table 3.2 Transitions (events) description for Figure 3.4

Transition	Event	From	To	Description
t_1	e_1	S_0	S_1	Start application
t_2	e_2	S_1	S_1	Select patient
t_3	e_3	S_1	S_2	Click button 'New'

t_4	e_4	S_2	S_1	Click button 'Ok'
t_5	e_5	S_2	S_1	Click button 'Cancel'
t_6	e_6	S_2	S_2	Type on Edit 'Name'
t_7	e_7	S_2	S_2	Type on Edit 'Addr'
t_8	e_8	S_1	S_3	Click button 'Ok'
t_9	e_9	S_3	S_1	Click button 'Close'
t_{10}	e_{10}	S_3	S_4	Click button 'Edit'
t_{11}	e_{11}	S_4	S_3	Click button 'Ok'
t_{12}	e_{12}	S_4	S_3	Click button 'Cancel'
t_{13}	e_{13}	S_4	S_4	Type on Edit 'Name'
t_{14}	e_{14}	S_4	S_4	Type on Edit 'Addr'
t_{15}	e_{15}	S_3	S_5	Click button 'Medication'
t_{16}	e_{16}	S_5	S_3	Click button 'Close'
t_{17}	e_{17}	S_5	S_6	Click button 'New'
t_{18}	e_{18}	S_6	S_5	Click button 'Ok'
t_{19}	e_{19}	S_6	S_5	Click button 'Cancel'
t_{20}	e_{20}	S_5	S_5	Select grid item
t_{21}	e_{21}	S_6	S_6	Type on Edit 'Date'
t_{22}	e_{22}	S_6	S_6	Type on Edit 'Drug'
t_{23}	e_{23}	S_6	S_6	Type on Edit 'Dose'
t_{24}	e_{24}	S_5	S_7	Click button 'Edit'
t_{25}	e_{25}	S_7	S_5	Click button 'Ok'
t_{26}	e_{26}	S_7	S_5	Click button 'Cancel'
t_{27}	e_{27}	S_7	S_7	Type on Edit 'Date'
t_{28}	e_{28}	S_7	S_7	Type on Edit 'Drug'
t_{29}	e_{29}	S_7	S_7	Type on Edit 'Dose'
t_{30}	e_{30}	S_1	S_e	Click button 'Close'

Depending on the definition of states, there can be a widely different numbers of states for the same AUT. For GUI testing automation, ignoring all properties will leave many cases where the resulting state of some states is not well-defined (not unique). However, if we differentiate all different property values as different states, the number of states is too large to be computationally feasible. For example, in Figure 3.1, if the content of the edit box is considered to distinguish states, the state number

becomes unlimited due to the unlimited space of possible typing sequences. This thesis uses selected set of property values to differentiate states. The state selection in the proposed GUITAM is at a similar level to the EFG model, which is also practical in terms of storage and computational complexity.

3.4 Automatic construction of GUITAM.

Runtime GUI information is programmatically readable, which provides an opportunity to automatically generate a GUITAM for an application by traversing its GUI states - Automatically generating a GUITAM requires reading the widgets (objects) and performing the events on the GUI of the given AUTs. We have built a set of fundamental tools that read the widgets (objects), check states and perform events on the GUI. ReadState reads the current state of a given AUT (see Figure 3.2), Existing(*s, S*) tells whether *s* is contained in *S* according to certain criteria, GetEquivalentState(*s,S*) returns the state in *S* which is equivalent to *s* according to certain criteria, and MoveTerminateEventsToBottom(*E*) moves all the termination events to the bottom of the events collection, which is used when a new state is found. TravelTo(s_1, s_2) will navigate the AUT from state s_1 to s_2. In each state *s*, we used a variable '*nextEventIndex*' to record the next events to be performed. Algorithm AutoGenerateGUITAM recursively constructs the GUITAM for a given AUT.

Algorithm AutoGenerateGUITAM

1. AutoGenerateGUITAM (*ps* : GUIState, *pe* : Event, *M*: GUITAM)
2. {

```
3.   s =ReadState();//read the current state
4.   s' = GetEquivalentState(s,M.S);
5.   if( s'=null ){ // s is a new state
6.     s.E=s.GenPossibleEventList(); s.nextEventIndex=0;
7.     M.S = M.S=M.S ∪{s}; M. Σ = M. Σ ∪s.E;
8.     If(M.S=Ø) M.s0=s;
9.    }
10.   else Merge(s, s')
11.   if(ps ≠ null && pe ≠ null){ //Add transition from ps to s.
12.     M.T = M.T ∪{<ps, s, pe>};
13.   }
14.   while (s.HasMoreEvent())
15.   {
16.      e=s.GetNextEvent;
17.      Perform(e);
18.      AutoGenerateGUITAM (s,e,M); //Recursively gen for next state
19.      s'=readState()
20.      if(ts!=s) if(!NavigateTo(s, M))  return;
21.   }
22. }
```

Figure 3.5 Algorithm AutoGenerateGUITAM

AutoGenerateGUITAM Algorithm is a recursive function. It takes 3 parameters: *ps* is the previous state; *pe* is the previously performed event that belongs to *ps*. Both *ps* and *pe* are null when the algorithm is first invoked. *M* is a GUITAM model instance. *M* is empty when the algorithm is first invoked, and is the final GUITAM when the algorithm finishes. Before the algorithm is first called, the AUT information, such as file name and path, is supposed to have been set and the AUT is closed (before start).

Line 3 reads the current runtime state of the AUT. If the AUT has not been started, ReadState returns a state marked as 'BeforeStart'. Line 4 is used to check whether the current state is a new state or a state which has previously occurred. If GetEquivalentState returns a null, it means current runtime state *s* is a new state. GetEquivalentState uses certain state distinguishing criteria to compare the states. Different criteria may lead to a different collection of states. Lines 6 – 9 initialize the new state *s* by generating a list of all possible events in this state and set the *nextEventIdex* to *0*. This new state is also added to the GUITAM state collection, *M.S*, and all the possible events are added to the input collection, *M.* $\sum$. When a state is marked as 'BeforeStart', the only possible event on this state is 'StartApplication'. If the state *s* is already in the collection, *s* is to be merged with *s'* by line 10 for having both the runtime status and other information. If the previous state *ps* and previous event *pe* are not null, that means the current state is transited from the previous state and the corresponding transition <*ps, s, pe*> is added to the transition collection. Line 14 – 20 check all possible events in the current state and perform them in order. After each event is performed, the algorithm recursively checks the next state by using the same procedure. After the recursive call to AutoGenerateGUITAM, the runtime state may or may not be the same state as *s*. Line 19 re-reads the runtime state and checks it. If it is not the same state, a NavigateTo procedure is invoked to navigate the AUT to the state recorded in *s*. Figure 3.6 shows the details of the algorithm NavigateTo. The

failure of NavigateTo means a failure of the AUT and the algorithm can't go any further.

The output of this algorithm is an instance *M* of GUITAM model for the given AUT.

Algorithm NavigateTo

```
1.  bool NavigateTo(s: GUIState)
2.  {
3.      cs=ReasdState();
4.      path=FindPath(cs, s);
5.      if(path=null){
6.          path=FindPath(s0,s);
7.          if(path=null) return false;
8.          KillAUTProcess();
9.      }
10.     ts=NavigateAlongPath(path)
11.     if(ts!=s) return false;
12.      else return true;
13. }
```

Figure 3.6 Algorithm NavigateTo

In the GUITAM, the states and transitions constitute a graph of which the nodes represent states and the edges represent transitions. FindPath is a normal graph path search method. The algorithm NavigateTo first reads the current runtime state and searches whether there is a path from the runtime state (*cs*) to the given state (*s*) (lines 3 – 4). If there is a path, it then navigates the AUT from *cs* to *s* along the path (line 10). A path is a series of edges which record the corresponding events to be performed. If there is no path found from *cs* to *s*, it then quits the AUT by killing the process and finds the path from s_0 (before start) to *s* (there is at least one path from s_0 to *s*) (lines 5 –

9). Line 10 navigates the AUT to *s*. If this navigation is not successful, it means the AUT crashes or a failure occurs which stops the navigation. Figure 3.7 illustrates the detailed procedure of automatically generating a GUITAM for the simple clinic software shown in Figure 3.4 (a) and generating the GUITAM shown in Figure 3.4 (b).

(Nil,Nil,M) M={}
M={s_0}, s_0.E={<$e_{0,0}$, "StartApplication">}
∑={$e_{0,0}$},
T={}
(s_0,$e_{0,0}$,M)
M={s_0,s_1},
s_1.E={<$e_{1,0}$, "TypeString", Edit1>, <$e_{1,1}$, "New", Button>, <$e_{1,2}$,"Ok",Button>, <$e_{1,3}$, "Close", Button>}
∑={$e_{0,0}$, $e_{1,0}$, $e_{1,1}$, $e_{1,2}$,$e_{1,3}$},
T={<s_0,s_1,$e_{0,0}$>}
(s_1,$e_{1,0}$,M)
M={s_0,s_1},
T={<s_0,s_1,$e_{0,0}$>,<s_1,s_1,$e_{1,0}$>}
(s_1,$e_{1,1}$,M)
M={s_0,s_1,s_2},
s_2.E={<$e_{2,0}$,"TypeString","Name", Edit>, <$e_{2,1}$>, "TypeString","Addr",Edit>,<$e_{2.2}$, "Ok",Button>, <$e_{2,3}$, "Cancel", Button>}
∑={$e_{0,0}$, $e_{1,0}$, $e_{1,1}$, $e_{1,2}$,$e_{1,3}$, $e_{2,0}$, $e_{2,1}$,$e_{2,2}$, $e_{2,3}$}
T={<s_0,s_1,$e_{0,0}$>,<s_1,s_1,$e_{1,0}$>, <s_1,s_2,$e_{1,1}$>}
(s_2,$e_{2,0}$,M)
M={s_0,s_1,s_2},
T={<s_0,s_1,$e_{0,0}$>, <s_1,s_1,$e_{1,0}$>, <s_1,s_2,$e_{1,1}$>, <s_2,s_2,$e_{2,0}$>}
(s_2,$e_{2,1}$,M)
M={s_0,s_1,s_2},
T={<s_0,s_1,$e_{0,0}$>, <s_1,s_1,$e_{1,0}$>, <s_1,s_2,$e_{1,1}$>, <s_2,s_2,$e_{2,0}$>, <s_2,s_2, $e_{2,1}$>}
(s_2,$e_{2,2}$,M)
M={s_0,s_1,s_2},
T={<s_0,s_1,$e_{0,0}$>, <s_1,s_1,$e_{1,0}$>, <s_1,s_2,$e_{1,1}$>, <s_2,s_2,$e_{2,0}$>, <s_2,s_2, $e_{2,1}$>, <s_2,s_1,$e_{2,2}$>}
(s_1,$e_{1,2}$,M)
M={s_0,s_1,s_2,s_3},
s3.E={<e3,0,"Edit info",Button>, <e3,1,"Medication", Edit>, <e3,2,"Close", Button>}
∑={ $e_{0,0}$, $e_{1,0}$, $e_{1,1}$, $e_{1,2}$,$e_{1,3}$, $e_{2,0}$, $e_{2,1}$,$e_{2,2}$, $e_{2,3}$, $e_{3,0}$, $e_{3,1}$, $e_{3,2}$}
T={<s_0,s_1,$e_{0,0}$>, <s_1,s_1,$e_{1,0}$>, <s_1,s_2,$e_{1,1}$>, <s_2,s_2,$e_{2,0}$>, <s_2,s_2, $e_{2,1}$>, <s_2,s_1,$e_{2,2}$>, <s_1,s_2,$e_{1,2}$>}
(s_3,$e_{3,0}$,M)
M={s_0,s_1,s_2,s_3, s_4},
s_4.E={<$e_{4,0}$,"TypeString","Name", Edit>, <$e_{4,1}$,"TypeString","Addr", Edit>, <$e_{4,2}$,"Ok", Button>, <$e_{4,3}$,"Cancel",Button>}
∑={ $e_{0,0}$, $e_{1,0}$, $e_{1,1}$, $e_{1,2}$,$e_{1,3}$, $e_{2,0}$, $e_{2,1}$,$e_{2,2}$, $e_{2,3}$, $e_{3,0}$, $e_{3,1}$, $e_{3,2}$, $e_{4,0}$,$e_{4,1}$,$e_{4,2}$,$e_{4,3}$}
T={<s_0,s_1,$e_{0,0}$>, <s_1,s_1,$e_{1,0}$>, <s_1,s_2,$e_{1,1}$>, <s_2,s_2,$e_{2,0}$>, <s_2,s_2, $e_{2,1}$>, <s_2,s_1,$e_{2,2}$>, <s_1,s_2,$e_{1,2}$>, <s_3,s_4,$e_{3,0}$>}
(s_4,$e_{4,0}$,M)
M={s_0,s_1,s_2,s_3, s_4},
T={<s_0,s_1,$e_{0,0}$>, <s_1,s_1,$e_{1,0}$>, <s_1,s_2,$e_{1,1}$>, <s_2,s_2,$e_{2,0}$>, <s_2,s_2, $e_{2,1}$>, <s_2,s_1,$e_{2,2}$>, <s_1,s_2,$e_{1,2}$>, <s_3,s_4,$e_{3,0}$>, <s_4,s_4,$e_{4,0}$>}
(s_4,$e_{4,1}$,M)
M={s_0,s_1,s_2,s_3, s_4},

T={<s_0,s_1,$e_{0,0}$>, <s_1,s_1,$e_{1,0}$>, <s_1,s_2,$e_{1,1}$>, <s_2,s_2,$e_{2,0}$>, <s_2,s_2, $e_{2,1}$>, <s_2,s_1,$e_{2,2}$>, <s_1,s_2,$e_{1,2}$>, <s_3,s_4,$e_{3,0}$>, <s_4,s_4,$e_{4,0}$>, <s_4,s_4,$e_{4,1}$>}

(s_4,$e_{4,2}$,M)

M={s_0,s_1,s_2,s_3, s_4},

T={<s_0,s_1,$e_{0,0}$>, <s_1,s_1,$e_{1,0}$>, <s_1,s_2,$e_{1,1}$>, <s_2,s_2,$e_{2,0}$>, <s_2,s_2, $e_{2,1}$>, <s_2,s_1,$e_{2,2}$>, <s_1,s_2,$e_{1,2}$>, <s_3,s_4,$e_{3,0}$>, <s_4,s_4,$e_{4,0}$>, <s_4,s_4,$e_{4,1}$>, <s_4,s_3,$e_{4,2}$>}

(s_3,$e_{3,1}$,M)

M={s_0,s_1,s_2,s_3, s_4,s_5},

s_5.E={<$e_{5,0}$,"SelectRow",Grid>, <$e_{5,1}$,"New",Button>,<$e_{5,2}$,"Edit", Button>, <$e_{5,3}$,"Close",Button>}

∑={ $e_{0,0}$, $e_{1,0}$, $e_{1,1}$, $e_{1,2}$,$e_{1,3}$, $e_{2,0}$, $e_{2,1}$,$e_{2,2}$, $e_{2,3}$, $e_{3,0}$, $e_{3,1}$, $e_{3,2}$, $e_{4,0}$,$e_{4,1}$,$e_{4,2}$,$e_{4,3}$, $e_{5,0}$, $e_{5,1}$, $e_{5,2}$,$e_{5,3}$}

T={<s_0,s_1,$e_{0,0}$>, <s_1,s_1,$e_{1,0}$>, <s_1,s_2,$e_{1,1}$>, <s_2,s_2,$e_{2,0}$>, <s_2,s_2, $e_{2,1}$>, <s_2,s_1,$e_{2,2}$>, <s_1,s_2,$e_{1,2}$>, <s_3,s_4,$e_{3,0}$>, <s_4,s_4,$e_{4,0}$>, <s_4,s_4,$e_{4,1}$>, <s_4,s_3,$e_{4,2}$>, <s_3,s_5,$e_{3,1}$>}

(s_5,$e_{5,0}$,M)

M={s_0,s_1,s_2,s_3, s_4,s_5},

T={<s_0,s_1,$e_{0,0}$>, <s_1,s_1,$e_{1,0}$>, <s_1,s_2,$e_{1,1}$>, <s_2,s_2,$e_{2,0}$>, <s_2,s_2, $e_{2,1}$>, <s_2,s_1,$e_{2,2}$>, <s_1,s_2,$e_{1,2}$>, <s_3,s_4,$e_{3,0}$>, <s_4,s_4,$e_{4,0}$>, <s_4,s_4,$e_{4,1}$>, <s_4,s_3,$e_{4,2}$>, <s_3,s_5,$e_{3,1}$>, <s_5,s_5,$e_{5,0}$>}

(s_5,$e_{5,1}$,M)

M={s_0,s_1,s_2,s_3, s_4,s_5,s_6},

s_6.E={<$e_{6,0}$,"TypeString","Date",Edit">, <$e_{6,1}$,"TypeString","Drug",Edit">, <$e_{6,2}$, "TypeString", "Dose", Edit">, <$e_{6,3}$,"Ok",Button>, <$e_{6,4}$,"Cancel",Button>}

∑={ $e_{0,0}$, $e_{1,0}$, $e_{1,1}$, $e_{1,2}$,$e_{1,3}$, $e_{2,0}$, $e_{2,1}$,$e_{2,2}$, $e_{2,3}$, $e_{3,0}$, $e_{3,1}$, $e_{3,2}$, $e_{4,0}$,$e_{4,1}$,$e_{4,2}$,$e_{4,3}$, $e_{5,0}$, $e_{5,1}$, $e_{5,2}$,$e_{5,3}$, $e_{6,0}$,$e_{6,1}$, $e_{6,2}$,$e_{6,3}$,$e_{6,4}$}

T={<s_0,s_1,$e_{0,0}$>, <s_1,s_1,$e_{1,0}$>, <s_1,s_2,$e_{1,1}$>, <s_2,s_2,$e_{2,0}$>, <s_2,s_2, $e_{2,1}$>, <s_2,s_1,$e_{2,2}$>, <s_1,s_2,$e_{1,2}$>, <s_3,s_4,$e_{3,0}$>, <s_4,s_4,$e_{4,0}$>, <s_4,s_4,$e_{4,1}$>, <s_4,s_3,$e_{4,2}$>, <s_3,s_5,$e_{3,1}$>, <s_5,s_5,$e_{5,0}$>, <s_5,s_6,$e_{5,1}$>}

(s_6,$e_{6,0}$,M)

M={s_0,s_1,s_2,s_3, s_4,s_5,s_6},

T={<s_0,s_1,$e_{0,0}$>, <s_1,s_1,$e_{1,0}$>, <s_1,s_2,$e_{1,1}$>, <s_2,s_2,$e_{2,0}$>, <s_2,s_2, $e_{2,1}$>, <s_2,s_1,$e_{2,2}$>, <s_1,s_2,$e_{1,2}$>, <s_3,s_4,$e_{3,0}$>, <s_4,s_4,$e_{4,0}$>, <s_4,s_4,$e_{4,1}$>, <s_4,s_3,$e_{4,2}$>, <s_3,s_5,$e_{3,1}$>, <s_5,s_5,$e_{5,0}$>, <s_5,s_6,$e_{5,1}$>, <s_6,s_6, $e_{6,0}$>}

(s_6,$e_{6,1}$,M)

M={s_0,s_1,s_2,s_3, s_4,s_5,s_6},

T={<s_0,s_1,$e_{0,0}$>, <s_1,s_1,$e_{1,0}$>, <s_1,s_2,$e_{1,1}$>, <s_2,s_2,$e_{2,0}$>, <s_2,s_2, $e_{2,1}$>, <s_2,s_1,$e_{2,2}$>, <s_1,s_2,$e_{1,2}$>, <s_3,s_4,$e_{3,0}$>, <s_4,s_4,$e_{4,0}$>, <s_4,s_4,$e_{4,1}$>, <s_4,s_3,$e_{4,2}$>, <s_3,s_5,$e_{3,1}$>, <s_5,s_5,$e_{5,0}$>, <s_5,s_6,$e_{5,1}$>, <s_6,s_6, $e_{6,0}$>, <s_6,s_6, $e_{6,1}$>}

(s_6,$e_{6,2}$,M)

M={s_0,s_1,s_2,s_3, s_4,s_5,s_6},

T={<s_0,s_1,$e_{0,0}$>, <s_1,s_1,$e_{1,0}$>, <s_1,s_2,$e_{1,1}$>, <s_2,s_2,$e_{2,0}$>, <s_2,s_2, $e_{2,1}$>, <s_2,s_1,$e_{2,2}$>, <s_1,s_2,$e_{1,2}$>, <s_3,s_4,$e_{3,0}$>, <s_4,s_4,$e_{4,0}$>, <s_4,s_4,$e_{4,1}$>, <s_4,s_3,$e_{4,2}$>, <s_3,s_5,$e_{3,1}$>, <s_5,s_5,$e_{5,0}$>, <s_5,s_6,$e_{5,1}$>, <s_6,s_6, $e_{6,0}$>, <s_6,s_6, $e_{6,1}$>, <s_6,s_6,$e_{6,2}$>}

(s_6,$e_{6,3}$,M)

M={s_0,s_1,s_2,s_3, s_4,s_5,s_6},

T={<s_0,s_1,$e_{0,0}$>, <s_1,s_1,$e_{1,0}$>, <s_1,s_2,$e_{1,1}$>, <s_2,s_2,$e_{2,0}$>, <s_2,s_2, $e_{2,1}$>, <s_2,s_1,$e_{2,2}$>, <s_1,s_2,$e_{1,2}$>, <s_3,s_4,$e_{3,0}$>, <s_4,s_4,$e_{4,0}$>, <s_4,s_4,$e_{4,1}$>, <s_4,s_3,$e_{4,2}$>, <s_3,s_5,$e_{3,1}$>, <s_5,s_5,$e_{5,0}$>, <s_5,s_6,$e_{5,1}$>, <s_6,s_6, $e_{6,0}$>, <s_6,s_6, $e_{6,1}$>, <s_6,s_6,$e_{6,2}$>, <s_6,s_5, $e_{6,3}$>}

(s_5,$e_{5,2}$,M)

M={s_0,s_1,s_2,s_3, s_4,s_5,s_6, s_7},

T={<s_0,s_1,$e_{0,0}$>, <s_1,s_1,$e_{1,0}$>, <s_1,s_2,$e_{1,1}$>, <s_2,s_2,$e_{2,0}$>, <s_2,s_2, $e_{2,1}$>, <s_2,s_1,$e_{2,2}$>, <s_1,s_2,$e_{1,2}$>, <s_3,s_4,$e_{3,0}$>, <s_4,s_4,$e_{4,0}$>, <s_4,s_4,$e_{4,1}$>, <s_4,s_3,$e_{4,2}$>, <s_3,s_5,$e_{3,1}$>, <s_5,s_5,$e_{5,0}$>, <s_5,s_6,$e_{5,1}$>,

<s_6,s_6, $e_{6,0}$>, <s_6,s_6, $e_{6,1}$>, <s_6,s_6,$e_{6,2}$>, <s_6,s_5, $e_{6,3}$>, <s_5,s_7,$e_{5,2}$>}
S_7.E={<$e_{7,0}$,"TypeString","Date",Edit">, <$e_{7,1}$,"TypeString","Drug",Edit">, <$e_{7,2}$,"TypeString","Dose",Edit">, <$e_{7,3}$,"Ok",Button>, <$e_{7,4}$,"Cancel",Button>}
∑={ $e_{0,0}$, $e_{1,0}$, $e_{1,1}$, $e_{1,2}$,$e_{1,3}$, $e_{2,0}$, $e_{2,1}$,$e_{2,2}$, $e_{2,3}$, $e_{3,0}$, $e_{3,1}$, $e_{3,2}$, $e_{4,0}$,$e_{4,1}$,$e_{4,2}$,$e_{4,3}$, $e_{5,0}$, $e_{5,1}$, $e_{5,2}$,$e_{5,3}$, $e_{6,0}$,$e_{6,1}$, $e_{6,2}$,$e_{6,3}$,$e_{6,4}$,$e_{7,0}$,$e_{7,1}$, $e_{7,2}$,$e_{7,3}$,$e_{7,4}$}

(s_7,$e_{7,0}$,M)
M={s_0,s_1,s_2,s_3, s_4,s_5,s_6, s_7},
T={<s_0,s_1,$e_{0,0}$>, <s_1,s_1,$e_{1,0}$>, <s_1,s_2,$e_{1,1}$>, <s_2,s_2,$e_{2,0}$>, <s_2,s_2, $e_{2,1}$>, <s_2,s_1,$e_{2,2}$>, <s_1,s_2,$e_{1,2}$>, <s_3,s_4,$e_{3,0}$>, <s_4,s_4,$e_{4,0}$>, <s_4,s_4,$e_{4,1}$>, <s_4,s_3,$e_{4,2}$>, <s_3,s_5,$e_{3,1}$>, <s_5,s_5,$e_{5,0}$>, <s_5,s_6,$e_{5,1}$>, <s_6,s_6, $e_{6,0}$>, <s_6,s_6, $e_{6,1}$>, <s_6,s_6,$e_{6,2}$>, <s_6,s_5, $e_{6,3}$>, <s_5,s_7,$e_{5,2}$>,<s_7,s_7,$s_{7,0}$>}

(s_7,$e_{7,1}$,M)
M={s_0,s_1,s_2,s_3, s_4,s_5,s_6, s_7},
T={<s_0,s_1,$e_{0,0}$>, <s_1,s_1,$e_{1,0}$>, <s_1,s_2,$e_{1,1}$>, <s_2,s_2,$e_{2,0}$>, <s_2,s_2, $e_{2,1}$>, <s_2,s_1,$e_{2,2}$>, <s_1,s_2,$e_{1,2}$>, <s_3,s_4,$e_{3,0}$>, <s_4,s_4,$e_{4,0}$>, <s_4,s_4,$e_{4,1}$>, <s_4,s_3,$e_{4,2}$>, <s_3,s_5,$e_{3,1}$>, <s_5,s_5,$e_{5,0}$>, <s_5,s_6,$e_{5,1}$>, <s_6,s_6, $e_{6,0}$>, <s_6,s_6, $e_{6,1}$>, <s_6,s_6,$e_{6,2}$>, <s_6,s_5, $e_{6,3}$>, <s_5,s_7,$e_{5,2}$>,<s_7,s_7,$s_{7,0}$>, <s_7,s_7,$s_{7,1}$>}

(s_7,$e_{7,2}$,M)
M={s_0,s_1,s_2,s_3, s_4,s_5,s_6, s_7},
T={<s_0,s_1,$e_{0,0}$>, <s_1,s_1,$e_{1,0}$>, <s_1,s_2,$e_{1,1}$>, <s_2,s_2,$e_{2,0}$>, <s_2,s_2, $e_{2,1}$>, <s_2,s_1,$e_{2,2}$>, <s_1,s_2,$e_{1,2}$>, <s_3,s_4,$e_{3,0}$>, <s_4,s_4,$e_{4,0}$>, <s_4,s_4,$e_{4,1}$>, <s_4,s_3,$e_{4,2}$>, <s_3,s_5,$e_{3,1}$>, <s_5,s_5,$e_{5,0}$>, <s_5,s_6,$e_{5,1}$>, <s_6,s_6, $e_{6,0}$>, <s_6,s_6, $e_{6,1}$>, <s_6,s_6,$e_{6,2}$>, <s_6,s_5, $e_{6,3}$>, <s_5,s_7,$e_{5,2}$>,<s_7,s_7,$s_{7,0}$>, <s_7,s_7,$s_{7,1}$>, <s_7,s_7,$s_{7,2}$>}

(s_7,$e_{7,3}$,M)
M={s_0,s_1,s_2,s_3, s_4,s_5,s_6, s_7},
T={<s_0,s_1,$e_{0,0}$>, <s_1,s_1,$e_{1,0}$>, <s_1,s_2,$e_{1,1}$>, <s_2,s_2,$e_{2,0}$>, <s_2,s_2, $e_{2,1}$>, <s_2,s_1,$e_{2,2}$>, <s_1,s_2,$e_{1,2}$>, <s_3,s_4,$e_{3,0}$>, <s_4,s_4,$e_{4,0}$>, <s_4,s_4,$e_{4,1}$>, <s_4,s_3,$e_{4,2}$>, <s_3,s_5,$e_{3,1}$>, <s_5,s_5,$e_{5,0}$>, <s_5,s_6,$e_{5,1}$>, <s_6,s_6, $e_{6,0}$>, <s_6,s_6, $e_{6,1}$>, <s_6,s_6,$e_{6,2}$>, <s_6,s_5, $e_{6,3}$>, <s_5,s_7,$e_{5,2}$>,<s_7,s_7,$s_{7,0}$>, <s_7,s_7,$s_{7,1}$>, <s_7,s_7,$s_{7,2}$>, <s_7,s_5,$e_{7,3}$>}

(s_5,$e_{5,2}$,M)
M={s_0,s_1,s_2,s_3, s_4,s_5,s_6, s_7},
T={<s_0,s_1,$e_{0,0}$>, <s_1,s_1,$e_{1,0}$>, <s_1,s_2,$e_{1,1}$>, <s_2,s_2,$e_{2,0}$>, <s_2,s_2, $e_{2,1}$>, <s_2,s_1,$e_{2,2}$>, <s_1,s_2,$e_{1,2}$>, <s_3,s_4,$e_{3,0}$>, <s_4,s_4,$e_{4,0}$>, <s_4,s_4,$e_{4,1}$>, <s_4,s_3,$e_{4,2}$>, <s_3,s_5,$e_{3,1}$>, <s_5,s_5,$e_{5,0}$>, <s_5,s_6,$e_{5,1}$>, <s_6,s_6, $e_{6,0}$>, <s_6,s_6, $e_{6,1}$>, <s_6,s_6,$e_{6,2}$>, <s_6,s_5, $e_{6,3}$>, <s_5,s_7,$e_{5,2}$>,<s_7,s_7,$s_{7,0}$>, <s_7,s_7,$s_{7,1}$>, <s_7,s_7,$s_{7,2}$>, <s_7,s_5,$e_{7,3}$>, <s_5,s_3,$e_{5,2}$>}

(s_3,$e_{3,2}$,M)
M={s_0,s_1,s_2,s_3, s_4,s_5,s_6, s_7},
T={<s_0,s_1,$e_{0,0}$>, <s_1,s_1,$e_{1,0}$>, <s_1,s_2,$e_{1,1}$>, <s_2,s_2,$e_{2,0}$>, <s_2,s_2, $e_{2,1}$>, <s_2,s_1,$e_{2,2}$>, <s_1,s_2,$e_{1,2}$>, <s_3,s_4,$e_{3,0}$>, <s_4,s_4,$e_{4,0}$>, <s_4,s_4,$e_{4,1}$>, <s_4,s_3,$e_{4,2}$>, <s_3,s_5,$e_{3,1}$>, <s_5,s_5,$e_{5,0}$>, <s_5,s_6,$e_{5,1}$>, <s_6,s_6, $e_{6,0}$>, <s_6,s_6, $e_{6,1}$>, <s_6,s_6,$e_{6,2}$>, <s_6,s_5, $e_{6,3}$>, <s_5,s_7,$e_{5,2}$>,<s_7,s_7,$s_{7,0}$>, <s_7,s_7,$s_{7,1}$>, <s_7,s_7,$s_{7,2}$>, <s_7,s_5,$e_{7,3}$>, <s_5,s_3,$e_{5,2}$>, <s_3,s_1,$e_{3,2}$>}

(s_1,$e_{1,3}$,M)
M={s_0,s_1,s_2,s_3, s_4,s_5,s_6, s_7},
T={<s_0,s_1,$e_{0,0}$>, <s_1,s_1,$e_{1,0}$>,
<s_1,s_2,$e_{1,1}$>, <s_2,s_2,$e_{2,0}$>, <s_2,s_2, $e_{2,1}$>,
<s_2,s_1,$e_{2,2}$>, <s_1,s_2,$e_{1,2}$>, <s_3,s_4,$e_{3,0}$>,
<s_4,s_4,$e_{4,0}$>, <s_4,s_4,$e_{4,1}$>, <s_4,s_3,$e_{4,2}$>,
<s_3,s_5,$e_{3,1}$>, <s_5,s_5,$e_{5,0}$>, <s_5,s_6,$e_{5,1}$>,
<s_6,s_6, $e_{6,0}$>, <s_6,s_6, $e_{6,1}$>,
<s_6,s_6,$e_{6,2}$>, <s_6,s_5, $e_{6,3}$>,
<s_5,s_7,$e_{5,2}$>,<s_7,s_7,$s_{7,0}$>, <s_7,s_7,$s_{7,1}$>,
<s_7,s_7,$s_{7,2}$>, <s_7,s_5,$e_{7,3}$>, <s_5,s_3,$e_{5,2}$>,
<s_3,s_1,$e_{3,2}$>, <$\underline{s_1,s_e,e_{1,3}}$>}

(Not s_7, re-open application and navigate to s_7)
(s_7,$e_{7,4}$,M)
M={s_0,s_1,s_2,s_3, s_4,s_5,s_6, s_7},
T={<s_0,s_1,$e_{0,0}$>, <s_1,s_1,$e_{1,0}$>, <s_1,s_2,$e_{1,1}$>, <s_2,s_2,$e_{2,0}$>,
<s_2,s_2, $e_{2,1}$>, <s_2,s_1,$e_{2,2}$>, <s_1,s_2,$e_{1,2}$>, <s_3,s_4,$e_{3,0}$>,
<s_4,s_4,$e_{4,0}$>, <s_4,s_4,$e_{4,1}$>, <s_4,s_3,$e_{4,2}$>, <s_3,s_5,$e_{3,1}$>,
<s_5,s_5,$e_{5,0}$>, <s_5,s_6,$e_{5,1}$>, <s_6,s_6, $e_{6,0}$>, <s_6,s_6, $e_{6,1}$>,
<s_6,s_6,$e_{6,2}$>, <s_6,s_5, $e_{6,3}$>, <s_5,s_7,$e_{5,2}$>,<s_7,s_7,$s_{7,0}$>,
<s_7,s_7,$s_{7,1}$>, <s_7,s_7,$s_{7,2}$>, <s_7,s_5,$e_{7,3}$>, <s_5,s_3,$e_{5,2}$>,
<s_3,s_1,$e_{3,2}$>, <$\underline{s_1,s_e,e_{1,3}}$>, <$s_7$,$s_5$,$e_{7,4}$>}

(Not s_6, navigate to s_6)
(s_6,$e_{6,3}$,M)
M={s_0,s_1,s_2,s_3, s_4,s_5,s_6, s_7},
T={<s_0,s_1,$e_{0,0}$>, <s_1,s_1,$e_{1,0}$>, <s_1,s_2,$e_{1,1}$>, <s_2,s_2,$e_{2,0}$>, <s_2,s_2,
$e_{2,1}$>, <s_2,s_1,$e_{2,2}$>, <s_1,s_2,$e_{1,2}$>, <s_3,s_4,$e_{3,0}$>, <s_4,s_4,$e_{4,0}$>,
<s_4,s_4,$e_{4,1}$>, <s_4,s_3,$e_{4,2}$>, <s_3,s_5,$e_{3,1}$>, <s_5,s_5,$e_{5,0}$>, <s_5,s_6,$e_{5,1}$>,
<s_6,s_6, $e_{6,0}$>, <s_6,s_6, $e_{6,1}$>, <s_6,s_6,$e_{6,2}$>, <s_6,s_5, $e_{6,3}$>,
<s_5,s_7,$e_{5,2}$>,<s_7,s_7,$s_{7,0}$>, <s_7,s_7,$s_{7,1}$>, <s_7,s_7,$s_{7,2}$>, <s_7,s_5,$e_{7,3}$>,
<s_5,s_3,$e_{5,2}$>, <s_3,s_1,$e_{3,2}$>, <$\underline{s_1,s_e,e_{1,3}}$>, <$s_7$,$s_5$,$e_{7,4}$>, <$s_6$,$s_5$,$e_{6,4}$>}

(Not s_4, navigate to s_4)
(s_4,$e_{4,3}$,M)
M={s_0,s_1,s_2,s_3, s_4,s_5,s_6, s_7},
T={<s_0,s_1,$e_{0,0}$>, <s_1,s_1,$e_{1,0}$>, <s_1,s_2,$e_{1,1}$>, <s_2,s_2,$e_{2,0}$>, <s_2,s_2, $e_{2,1}$>, <s_2,s_1,$e_{2,2}$>,
<s_1,s_2,$e_{1,2}$>, <s_3,s_4,$e_{3,0}$>, <s_4,s_4,$e_{4,0}$>, <s_4,s_4,$e_{4,1}$>, <s_4,s_3,$e_{4,2}$>, <s_3,s_5,$e_{3,1}$>,
<s_5,s_5,$e_{5,0}$>, <s_5,s_6,$e_{5,1}$>, <s_6,s_6, $e_{6,0}$>, <s_6,s_6, $e_{6,1}$>, <s_6,s_6,$e_{6,2}$>, <s_6,s_5, $e_{6,3}$>,
<s_5,s_7,$e_{5,2}$>,<s_7,s_7,$s_{7,0}$>, <s_7,s_7,$s_{7,1}$>, <s_7,s_7,$s_{7,2}$>, <s_7,s_5,$e_{7,3}$>, <s_5,s_3,$e_{5,2}$>,
<s_3,s_1,$e_{3,2}$>, <$\underline{s_1,s_e,e_{1,3}}$>, <$s_7$,$s_5$,$e_{7,4}$>, <$s_6$,$s_5$,$e_{6,4}$>, <$s_4$,$s_3$,$e_{4,3}$>}

(Not s_2, navigate to s_2)
(s_2,$e_{2,3}$,M)
M={ s_0,s_1,s_2,s_3, s_4,s_5,s_6, s_7},
T={<s_0,s_1,$e_{0,0}$>, <s_1,s_1,$e_{1,0}$>, <s_1,s_2,$e_{1,1}$>, <s_2,s_2,$e_{2,0}$>, <s_2,s_2, $e_{2,1}$>, <s_2,s_1,$e_{2,2}$>, <s_1,s_2,$e_{1,2}$>,
<s_3,s_4,$e_{3,0}$>, <s_4,s_4,$e_{4,0}$>, <s_4,s_4,$e_{4,1}$>, <s_4,s_3,$e_{4,2}$>, <s_3,s_5,$e_{3,1}$>, <s_5,s_5,$e_{5,0}$>, <s_5,s_6,$e_{5,1}$>, <s_6,s_6,
$e_{6,0}$>, <s_6,s_6, $e_{6,1}$>, <s_6,s_6,$e_{6,2}$>, <s_6,s_5, $e_{6,3}$>, <s_5,s_7,$e_{5,2}$>,<s_7,s_7,$s_{7,0}$>, <s_7,s_7,$s_{7,1}$>, <s_7,s_7,$s_{7,2}$>,
<s_7,s_5,$e_{7,3}$>, <s_5,s_3,$e_{5,2}$>, <s_3,s_1,$e_{3,2}$>, <$\underline{s_1,s_e,e_{1,3}}$>, <$s_7$,$s_5$,$e_{7,4}$>, <$s_6$,$s_5$,$e_{6,4}$>,
<s_4,s_3,$e_{4,3}$>,<s_2,s_1,$e_{2,3}$>}

Algorithm Finishes.

The GUITAM is:
M={s_0,s_1,s_2,s_3, s_4,s_5,s_6, s_7},
T={<s_0,s_1,$e_{0,0}$>, <s_1,s_1,$e_{1,0}$>, <s_1,s_2,$e_{1,1}$>, <s_2,s_2,$e_{2,0}$>, <s_2,s_2, $e_{2,1}$>, <s_2,s_1,$e_{2,2}$>, <s_1,s_2,$e_{1,2}$>, <s_3,s_4,$e_{3,0}$>,
<s_4,s_4,$e_{4,0}$>, <s_4,s_4,$e_{4,1}$>, <s_4,s_3,$e_{4,2}$>, <s_3,s_5,$e_{3,1}$>, <s_5,s_5,$e_{5,0}$>, <s_5,s_6,$e_{5,1}$>, <s_6,s_6, $e_{6,0}$>, <s_6,s_6, $e_{6,1}$>,
<s_6,s_6,$e_{6,2}$>, <s_6,s_5, $e_{6,3}$>, <s_5,s_7,$e_{5,2}$>,<s_7,s_7,$s_{7,0}$>, <s_7,s_7,$s_{7,1}$>, <s_7,s_7,$s_{7,2}$>, <s_7,s_5,$e_{7,3}$>, <s_5,s_3,$e_{5,2}$>,
<s_3,s_1,$e_{3,2}$>, <$\underline{s_1,s_e,e_{1,3}}$>, <$s_7$,$s_5$,$e_{7,4}$>, <$s_6$,$s_5$,$e_{6,4}$>, <$s_4$,$s_3$,$e_{4,3}$>, <$s_2$,$s_1$,$e_{2,3}$>}
$\sum$={$e_{0,0}$, $e_{1,0}$, $e_{1,1}$, $e_{1,2}$,$e_{1,3}$, $e_{2,0}$, $e_{2,1}$,$e_{2,2}$, $e_{2,3}$, $e_{3,0}$, $e_{3,1}$, $e_{3,2}$, $e_{4,0}$,$e_{4,1}$,$e_{4,2}$,$e_{4,3}$, $e_{5,0}$, $e_{5,1}$, $e_{5,2}$,$e_{5,3}$, $e_{6,0}$,$e_{6,1}$,
$e_{6,2}$,$e_{6,3}$,$e_{6,4}$,$e_{7,0}$,$e_{7,1}$, $e_{7,2}$,$e_{7,3}$,$e_{7,4}$}

Figure 3.7 Illustration of AutoGenerateGUITAM algorithm for Simple Clinic Software

In Figure 3.7, the bold text in parentheses such as **(s_2,$e_{2,3}$,M)** signifies a call to the algorithm AutoGenerateGUITAM with given parameters. The algorithm starts from (**nil, nil, M**) where M = {} and start state and event are all nil. Each indent on the graph means a recursive call to AutoGenerateGUITAM. When the algorithm finishes, M contains the GUITAM generated.

3.5 Analysis of GUITAM

This section will analyze the GUITAM as compared to the EFG test automation. First the algorithm AutoGenerateGUITAM in Figure 3.4 will be proved to be complete. Then it will also be proved that for each EFG based test, there exists a GUITAM that can automate the test, with no further requirements on storage and computational power. This section will also illustrate that for the two scenarios of "non-fixed events set" and "expandable panel", GUITAM is able to automate tests, while EFG cannot.

3.5.1 Completeness of the Algorithm AutoGenerateGUITAM

Given certain criteria for comparing different states, AutoGenerateGUITAM is able to generate a GUITAM which models all runtime states and possible events of an AUT.

Theorem 3.1 AutoGenerateGUITAM can generate all possible states and transitions of an AUT.

Proof: Let us use finite induction to prove the completeness of the algorithm AutoGenerateGUITAM. Let *S* be the set of all states in GUITAM *M*.

1) s_0 is the state before the AUT is started. This state has only one transition called 'StartApplication' to the next state s_1. Line 14-21 ensures that$\forall e \in s_1.E$, e will be executed once and once only. Let N_{s1} be the collection of states that transited from s_1 after lines 14-21 finishe, then N_{s1} contains all possible states that can be from state s_1.

   $\forall s \in N_{s1}$, line 18 ensures that each s in N_{s1} will be treated the same as s_1 recursively.

2) Lines 4-9 ensure that all new states are added to *S*. Let s_k be any state already in *S*, let N_{sk} be the collection of states that transited from s_k after lines 14-21 finish, N_{sk} contains all possible states from S_k.

   Obviously, $\forall s_{k+1} \in N_{sk}$, $N_{s_{k+1}}$ contains all possible states that can be transited from s_{k+1}.

3) From 1) and 2), all possible states of the AUT are included after the algorithm finishes. □

From the proof, it can also be proved that all possible events are included as transitions in the GUITAM.

3.5.2 Inclusive Mapping between EFG and GUITAM.

In this section it will be proved that there exists an inclusive mapping between the EFG and the GUITAM, that is, for each EFG, there exists at least one GUITAM that is able to automate all the EFG automated tests.

Theorem 3.2: For each EFG, there exists at least one GUITAM, which can automate the tests of the EFG.

Proof: Let $C =<V, E, B, I>$ be an *EFG*. Let $\Sigma=V$ be the input domain which contains all the possible events. Because an event is always related to a GUI object, we can generate a state from any set of events. Suppose there is a function GenState(X) which can generate a state from a set of events X. S_0=GenState(B). $s_0.E=B$ is the initial state which contains the set of events of C that are available to the user when the component is first invoked; $T = S \times \Sigma \rightarrow S$ is a set of transitions $t=<s,s',v>$, where $s,s' \in S$, $v \in s.E$. s' is the state when $s'.E=\{v' \mid <v,v'> \in E\}$. Starting from s_0, all the transitions and their next states can be generated recursively. Thus, GUITAM $M=< \Sigma,S,s0,T >$ can automate all tests of the EFG. □

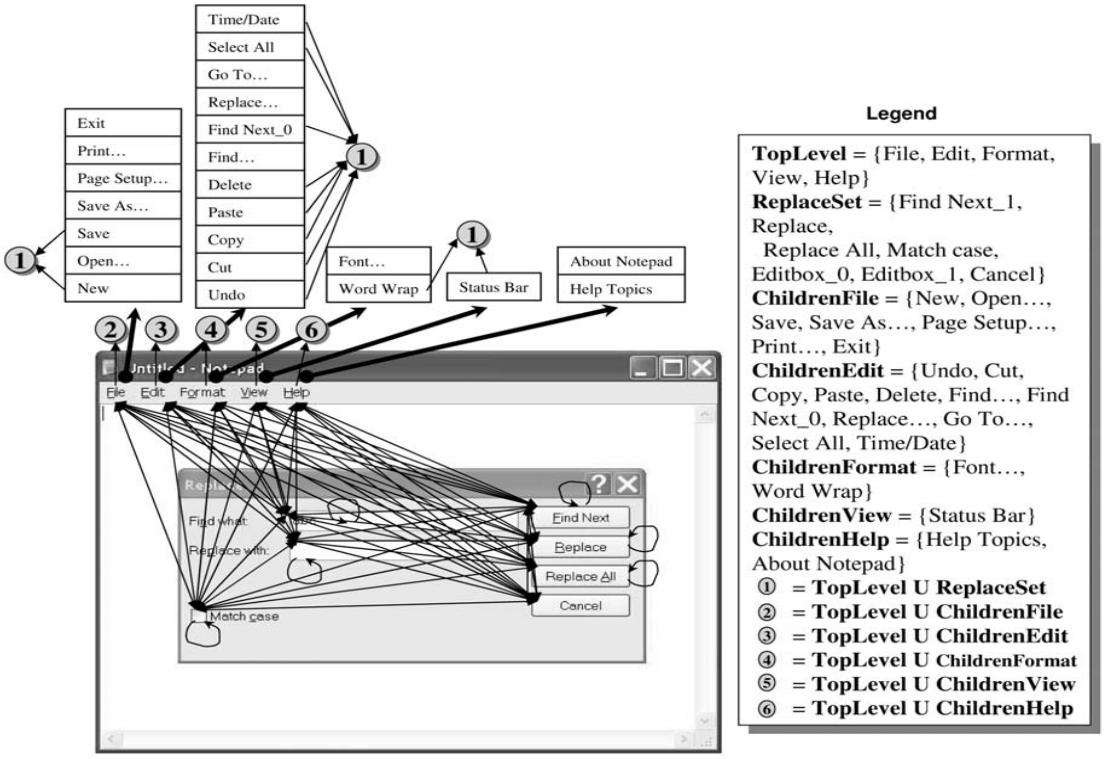
The proof of Theorem 3.2 also provides an approach for converting an EFG into a GUITAM. A drawback of this method is that there are often a number of unnecessary states in S, i.e., many states in S contain the same set of events. The number of states equals the number of events in Σ. The number of transitions in T equals the number of events in Σ as well. If we unite all the states, i.e., those states with the same set of events being considered as one state, the number of states will be greatly reduced. The

Theorem 3.2 also provides an algorithm to construct a more effective GUITAM of a given EFG. Figure 3.8 shows the details of the algorithm TransformEFGtoGUITAM.

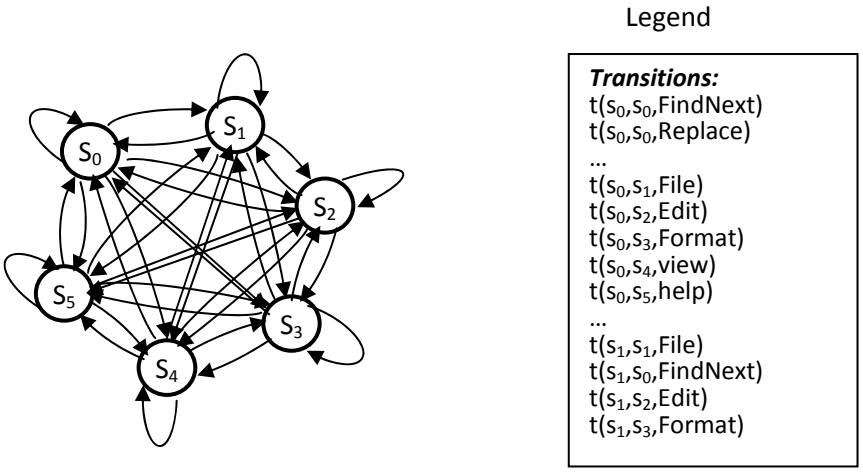
Algorithm TransformEFGtoGUITAM

```
1.    TransformEFGtoGUITAM (efg: EFG,M: GUITAM)
2.    {
3.      M.Init();
4.      M. Σ=efg.V;
5.      M. T = Ø;
6.      M.s0=GenState(B);
7.      M.S = M.S ∪ s0;
8.      Convert(efg, M, s0);
9.    }
10.   Convert(efg: EFG, M : GUITAM,s:State)
11.   {
12.     For each v in s{
13.       X={v' | <v,v'> ϵ efg.E} ;
14.       s'=GenState(X);
15.       s'.E=X;
16.       t=(s,s',v)
17.       M.S= M.S ∪ {s'};
18.       M. T = M. T ∪ {t};
19.       M. Σ = M. Σ ∪ X;
20.       Convert(efg, M, s');
21.     }
22.   }
```

Figure 3.8 Algorithm TransformEFGtoGUITAM



(a)



(b)

Figure 3.9 A GUITAM model converted from EFG in Figure 2.7

In Figure 3.9 (b), each state s_i is converted from the event set i of the EFG in Figure 3.9 (a). Figure 3.9 (a) is from Figure 2.8. All events in the EFG become transitions in this GUITAM.

3.5.3 Storage Analysis

To automate the GUI test which an EFG is able to perform, the GUITAM needs less storage than the EFG. The storage analysis is provided in Theorem 3.3.

Theorem 3.3: The space requirement of *M* created by algorithm 3.3 is not greater than the given EFG.

Proof:

In the given EFG (see section 2), vertices: = number of events. The order of storage requirement is $O(n)$.

Different types of events have a different number of edges (*B* is defined in the EFG):

– Menu-open: let *l*= number of events in *B* in EFG, m=number of menu-choices, the number of edges for these events is $C_l^2 + m = O(l^2)$.

– System-interaction: let *l*=number of events in *B*, then the number of edges for these events is $C_l^2 = O(l^2)$.

– Termination: let *k*=number of events of *B* of invoked components, then the number of edges for these events is k. The order is $O(k)$.

– Unrestricted-focus: let l = number of events in B, j = number of events in invoked modalless window, then the number of edges of the events is $C_{l+j}^2 = O((l+j)^2))$.

– Restricted-focus: let q=number of events in invoked modalless dialog, and then the edges of these events are q. The order is $O(q)$.

Because the number of events in B is much larger than the number of events of termination and invoking modal dialog, in each component, the number of events in B is very close to the total number n. On average, the number of edges in the EFG is $O(n^2)$.

In the constructed *M,* according to the definition of the *EFG*, every event has a fixed set of follow-up events, which means the values of properties are ignored. The number of states in *M* is approximately equal to the number of windows and possible popup menus, which is much less than the total number of events. In one component *C*, the state is about the number of top level menu items plus one.

Number of inputs in Σ equals the number of events, which is of *O(n)*.

Number of transitions *T* equals the number of events, which is of *O(n)*.

On average, the space complexity of *M* is of *O(n),* which is one order less than that of the given EFG. □

3.5.4 Computational Complexity Analysis

Both the EFG and GUITAM can be presented as directed graphs. Once the test cases are generated, the execution of the tests is the same with both models. Therefore, we only need to analyze the computational complexity for generating test cases from the models. The computational complexity of generating test cases depends on the requirement coverage of the test cases, the length of each test case and the number of test cases to be generated. A test case is an events sequence $(e_1, e_2, \ldots, e_k)$ where k is the length of the test case.

Theorem 3.4: Given the same length of each test, the computational complexity of generating test cases from the two models are of the same order.

Proof: Let n be the number of all events, and k be the test case length.

In the given EFG, except for a small number of events, most of the events have edges between each other, so the directed graph can be seen as a complete connected graph. Let q be the edge number of the graph, then $q = C_n^2$. To generate the test cases, traverse the graph and collect all the cases with length k, then the computational complexity is C_n^k.

In the GUITAM, each edge is related to an event, so the number of edges is n. To create a test case with length k, k edges need to be selected from the edge collection. The total number of possible combinations is C_n^k.

Thus, the computational complexity of generating test cases for both of the two models is of the same order. □

3.5.5 Dynamic GUI Interactions Modelling

According to the definition of the EFG and the algorithm GetFollows (Algorithm 2.1), we can see that EFGs are not able to model some dynamic situations when the underlying code changes the GUI object dynamically. For example:

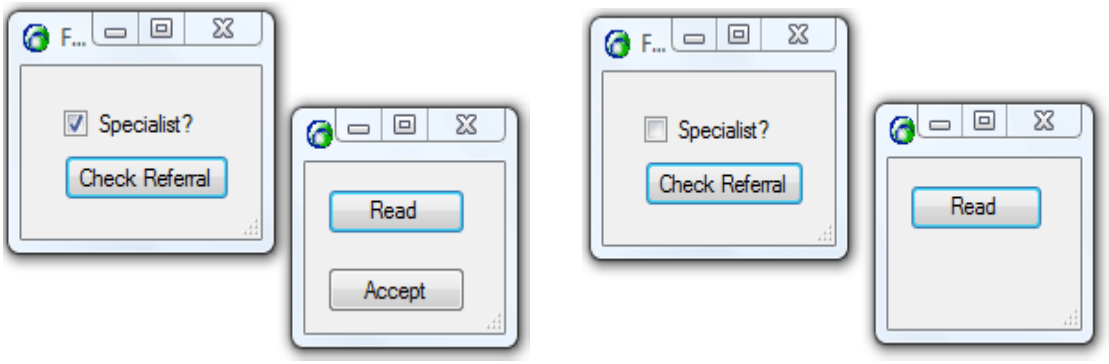
- Non-fixed events set. GUIs exist in many applications where the visibility of some objects is changed by the underlying code according to another object's state (e.g., the 'Checked' property value of a checkbox). In this case, the event leads to these GUIs being undefined or ill-defined in the following event set. They are dependable on the property of the checkbox.
- Expandable panel. Such GUIs also exist widely in many applications such as Microsoft Office 2007, where some panels (or modeless windows) are sometimes visible and sometimes invisible. Toggling the visibility-property value of a panel can cause some events to be exposed or hidden (controls in the newly-enabled panel or modeless window). According to the core algorithm of the EFG model, GetFollows, these events are not able to be modeled, and thus cannot be tested.

However, the GUITAM is able to model these situations. Two scenarios will be used to illustrate this.

- **Scenario 1:** Non-fixed events set.

Let's show an example code first (suppose CheckReferralClick is in Form f_1):

```
1.    CheckReferralClick ()
2.    {
3.       Form2 f2=new Form2();
4.       if(cbSpecialist.Checked)  f2.BAccept.Visible=true;
5.       else f2.BAccept.Visible=false;
6.       f2.ShowDialog();
7.    }
```



(a) (b)

Figure 3.10 Non-fixed follow-up event set of event 'CheckReferralClick'

In Figure 3.10 (a), when *cbSpecialist.Checked* is true, clicking on button 'Check Referral' opens a dialog with the 'Accept' button which means a specialist has the right to accept a referral. In Figure 3.10 (b), when *cbSpecialist.Checked* is false, clicking on the same button 'CheckReferralClick' opens a dialog without the 'Accept' button which means non-specialist doesn't have the right to accept a referral. Obviously, the same event 'CheckReferralClick' may have different sets of follow-up events.

According to the definition of the EFG, and the algorithm GetFollows, there will possibly be an edge from event CheckReferralClick to "Accept" in Form f_1. If there is

an edge, then when '*Checked*' is false, executing the edge leads to failure. If there is not an edge, when '*Checked*' is true, this case will never be executed. An EFG, therefore, cannot model this scenario.

With the GUITAM model, the value of the '*Checked*' property of *cbSpecialist* is used to differentiate states, so the form f_1 will have states which are called stateChecked and stateUnChecked respectively. f_2 has two states as well, named as stateWithAccept and StateNoAccept respectively. Then the state graph can be given as Figure 3.11.

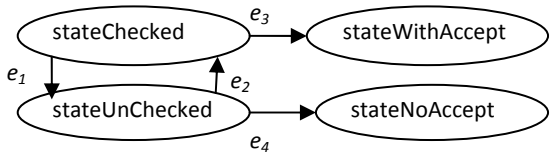


Figure 3.11 Non fixed events set in GUITAM

In Figure 3.11, two transitions, e_1 and e_2 are related to the *checkBox*$_1$Changed event.

e_1:<stateChecked, stateUnChecked, cbSpecialistChanged>

e_2:<stateUnChecked, stateChecked,cbSpecialistChanged>

e_3:<stateChecked, stateWithAccept, CheckReferralClick >

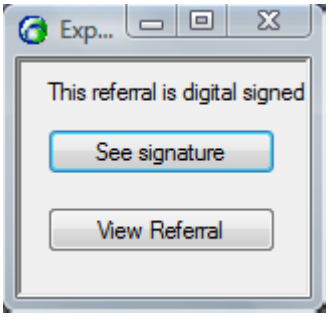
e_4:<stateUnChecked, stateNoAccept, CheckReferralClick >

It can be checked that all the execution flows of the application are modeled and this can be automated by GUITAM.

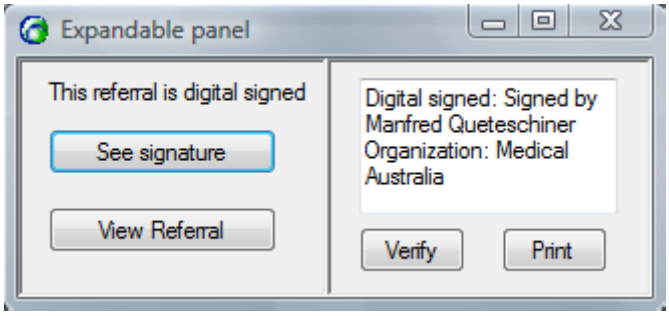
- **Scenario 2**: Expandable panel

Again, let me show a piece of code first. Suppose there are two panels. $Panel_1$ is always visible and, has two buttons: *See signature* and *View Referral*. $Panel_2$ can be either visible or not, affected by the event of SeeSignatureClick. $Penel_2$ has two buttons, *Verify* and *Print*. The two panels are in one form (Figure 3.12).

```
1.  SeeSignatureClick ()
2.  {
3.     panel2.visible=!panel2.visible;
4.  }
```



(a)



(b)

Figure 3.12 Expandable panel

In an EFG, event *SeeSignatureClick* has uncertain follow-ups. According to the algorithm, GetFollows can have different results and thus is undefined after

SeeSignatureClick, if $panel_2$ is visible, only *VerifyClick* and *PrintClick* are connected to *SeeSignatureClick*. However, a lot more needs to be done:

- *VerifyClick*, *PrintClick* should be connected to *ViewReferralClick*
- *SeeSignatureClick*, *ViewReferralClick* should be connected to *VerifyClick*
- *SeeSignatureClick*, *ViewReferralClick* should be connected to *PrintClick*

EFGs are not well defined in this situation.

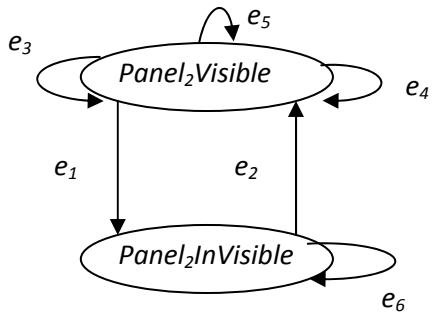


Figure 3.13 Panel visible changes state in the GUITAM

In the GUITAM, the two possible values of $panel_2$'s property 'Visible' are seen as two different states. Figure 3.13 illustrates the states and the transitions. The transitions are:

- e_1 <Panel$_2$Visible, panel$_2$Invisible, *SeeSignatureClick*>
- e_2 <Panel$_2$InVisible, panel$_2$Visible,*SeeSignatureClick*>
- e_3 <Panel$_2$Visible, panel$_2$Visible, *ViewReferralClick*>
- e_4 <Panel$_2$Visible, panel$_2$Visible, *VerifyClick*>
- e_5 <Panel$_2$Visible, panel$_2$Visible, *PrintClick*>

- e_6<Panel$_2$InVisible, panel$_2$InVisible, *ViewReferralClick*>

3.6 Representation of Test Cases

To test a GUI, test cases must be provided and executed. A test case for GUI testing is a series of actions. For convenience, some terminologies are introduced first.

In the GUITAM, $S = \{s_1, s_2, \ldots, s_n\}$ is the set of all states, $T = \{t_1, t_2, \ldots, t_m\}$ is the set of all transitions. We use the set $\mathbf{pred}(s_i) = \{s_p \mid (n_p, n_i) \in S\}$, $\mathbf{succ}(s_i) = \{s_p \mid (s_\rho, s_i) \in S\}$. $\mathbf{from}(t_i) = s_p$ iif $s_k = t_i(s_p)$, $\mathbf{to}(t_i) = s_p$ iif $s_p = t_i(s_k)$.

A sequence $\tau = < t_1, \ldots, t_k >$, $t_i = (s_i, s_{i+1}, e_i)$, $1 \leq i \leq k$ of consecutive transitions of M is called a ***walk*** and has an initial state, $\mathbf{init}(\tau) = s_1$, and a final state, $\mathbf{fin}(\tau) = s_{k+1}$. If $init(\tau) = fin(\tau)$, τ is ***closed***, a closed walk is a ***tour***. If $init(\tau) = s_0$, τ is ***initially rooted***. A tour is ***rooted*** if it is initially rooted. An empty sequence will be denoted by $\mathcal{E}$. If $\tau = \mathcal{E}$ or s_1 to s_k are distinct, τ is a ***path***. A closed path is a ***circuit***. A circuit with $k=1$ is a ***loop***. If $t_1, \ldots, t_k$ are distinct, τ is a ***route.*** Notice that in a path, $t_1, \ldots, t_k$ are also distinct, but in a route, $s_1, \ldots, s_{k+1}$ are not necessarily distinct.

A ***GUI test case*** is a rooted walk together with states information. A formal representation of a GUI test case is defined in Definition 3.4.

Definition 3.4: A ***GUI test case*** $\acute{\Gamma}$ is a triple $<s_0, \tau, S>$,

where s_0 is the initial state, $\tau = < t_1, \ldots, t_n >$ is a rooted walk, $S = < s_1; s_2; \ldots; s_n >$ is a set of expected states, where $s_i = t_i(s_{i-1})$ for $i = 1, \ldots, n$.

The expected state information is used for testing validation. To validate the test, the runtime GUI state is read and compared to the expected state after each event in the corresponding transaction t is executed. This comparison is taken by a mechanism called a test oracle. The expected state information is also called *oracle information*.

Definition 3.5: For a given test case $\acute{T} = <s_0, \tau, S>$, the ***test oracle information*** is a sequence $<S_1, S_2, \ldots, S_n>$, such that S_i is the (possibly partial) expected state of the GUI immediately after event $e_i \in \tau$ has been executed on it.

3.7 GUI Test Coverage Criteria

As in a rooted walk $\tau =< t_1, \ldots, t_n>$,as long as it is a legal sequence, there is no limit to the number of the occurrences of any t_i in the sequence. The number of possible test cases is actually infinite. Exhaustively executing all possible test cases during software testing is not therefore realistic, which means that we need to select test cases. Practically, only a small subset of all possible test cases is selected to form a test suite.

Definition 3.6: A ***GUI test suite*** $\check{T}$ is a set of test cases ($\acute{T}_1,\ldots, \acute{T}_e$); each $\acute{T}_1 \in \check{T}$ is a test case. All test cases have the same initial state s_0. e is the size of the suite.

To create test suites, coverage criteria of test cases are paramount for test cases selection. Different criteria serve different test purposes. Test coverage criteria are a set of rules which guide the generation of a test suite determining when to stop the generation, whether enough testing has been performed or further tests are needed, and provide an objective measure of the test suite quality. An ideal test criterion would be

capable of generating the smallest test suite possible for finding (if not all) the maximum number of errors of a software system. Common examples of coverage criteria for conventional software are structural, and include statement coverage, branch coverage, and path coverage, which require that every statement, branch and path in the program's code be executed by the test suite respectively. Existing coverage criteria developed for traditional software do not address the adequacy of GUI test cases. GUIs are typically developed using instances of precompiled elements stored in a library. The source code of these elements may not always be available to be used for coverage evaluation based on code. Moreover, GUI test cases are composed of events which may be performed in a random order. Event-based logic is hard to obtain from the code. Code based coverage is not suitable for testing GUI events.

In developing GUI testing coverage criteria, one needs to take account of different aspects to those in traditional software testing. Firstly, since GUIs consists of functional components, coverage criteria must be developed for covering the components. Secondly, since all operations to a GUI are through events, coverage criteria must be developed for covering the events routes. Thirdly, since GUI states contain information about an AUT, coverage criteria must be developed for covering the GUI states. Finally, the test designer should recognize whether a coverage criterion can be fully satisfied [73].

A GUITAM *M* model contains all possible GUI states and events in light of the given criteria of differentiating states. Because each event is related to a transition in the

GUITAM, the transition is equivalent to an event or a small set of events. Some coverage criteria are defined for test case generation and adequacy verification of an GUITAM.

Definition 3.7: For a GUITAM *M*, a test suite $\check{T}$ satisfies the ***state coverage criterion*** if and only if $\forall\, s \in M.S, \exists\, \acute{T} \in \check{T} \wedge\ \exists\, t \in \acute{T}.\tau \wedge (t.s1 = s \vee t.s2 = s)$.

To minimize the number of test cases, the paths which cover all the states can be selected. Each path constitutes a test case. All the test cases constitute the test suite $\check{T}$.

Definition 3.8: For a GUITAM *M,* a test suite Ť satisfies the ***event coverage criterion*** if and only if $\forall\, e \in M.\Sigma, \exists\, \acute{T} \in \check{T} \wedge\ \exists\, t \in \acute{T}.\tau \wedge\ t.e = e$.

Event coverage criterion is also called transition criterion in GUITAM testing. To reduce the suite size and meet this criterion, routes which cover all the transitions are selected. Each route constitutes a test case. All the test cases constitute the test suite Ť that satisfies the event coverage criterion.

Definition 3.9: For a GUITAM *M,* a test suite Ť satisfies the ***component coverage criterion*** if and only if $\forall\, o \in O, \exists\, \acute{T} \in \check{T} \wedge\ \exists\, t \in \acute{T}.\tau \wedge\ comp(t.e) = o$,

where *O* is the set of all components in the GUIs, and *comp(e)* denotes the component which contains the event *e*.

Definition 3.10: For a GUITAM *M,* a test suite Ť satisfies the ***length-n coverage criterion*** if and only if $\forall \acute{T} \in \check{T},\ len(\acute{T}.\tau) = n$, where $len(\acute{T}.\tau)$ denotes the number of transitions in $\acute{T}.\tau$.

3.8 Test Case Generation

With a GUITAM model, say *M*, test cases can be automatically generated by traversing the states in *M*. Any walk in M is an executable sequence of events. In principle, an infinite number of event sequences may be performed on a GUI. Depending on the resources available, a manageable number of these event sequences should be generated as test cases and tested on the GUI. Note that, unlike a traditional directed graph, there may exist more than one transition from one state to another (including to itself) in a GUITAM, which means more than one directed arcs from one state to another. There are various possible approaches to automatically generate test cases for GUIs, including the following:

1) Randomly select from walks in GUITAM

Since any walk in a GUITAM is an executable sequence, test cases can be generated by selecting a given number of walks randomly from the full set of the walks. Although straightforward to implement, this approach may yield a test suite without any control over choice and thus the test coverage is not predictable.

2) Criteria-based test case generation

According to given criteria, test cases can be generated by traversing the states and transitions in a GUITAM. For example, to meet the ***state coverage criterion,*** an algorithm can select the paths from the GUITAM until all states are covered. The paths can be as long as possible so that the minimum number of test cases can be generated which satisfy the criterion. Figure 3.14 shows a general algorithm of criteria-based test cases generating.

Algorithm GeneralGeneratingTestCase

```
1)  GeneralGeneratingTestCase (M :GUITAM, C :Criterion, Ť :TestSuite)
2)  {
3)      Ť=Ø;
4)      For each t ∈ M.T ∧ from(t) = s0
5)          GenerateTestCase(M, C, Ť,w, t, 0);
6)  }
7)  GenerateTestCase(M :GUITAM, C :Criterion, Ť :TestSuite, w :walk, t :transition,
    len:int)
8)  {
9)      w = w ∪ {t};
10)     if(w is proper){
11)         Testcase Ť = createtestcase(w);
12)         Ť = Ť ∪ {Ť} ;
13)     }
14)     if( meetcriterion(Ť, C)) return;
15)     for each t ∈ M.T ∧ from(t) = to(t)
16)     {
17)         generateTestCase(M, C, Ť, w, t, len+1);
18)     }
19)     w=w-{t};
20) }
```

Figure 3.14 General algorithm of criteria-based test cases generating

3) Defect classification orientated test case generation

Defects in GUI applications have many different types. Different types of defects are related to different kinds of events. The size of a test suite is often very large if it tries to cover all kinds of defects. The suite size is related to combinations of involved events. It may increase exponentially when the number of involved events increases. Normally, a certain type of defects involves a certain type of events which are only a part of the whole event set. The smaller number of events involved, defect classification directed test suite sizes will also decrease exponentially. This thesis systematically establishes a GUI defect classification, which includes criteria of classifying defects, distribution of defects and defect classification directed test case generation. The classification of defects will be presented in Chapter 4.

4) Use case oriented test case generation

An application was built for providing functions for users to perform tasks. In most cases, a user uses the application through typical scenarios. These typical scenarios are usually designed before the implementation of the application and treated as part of the application specifications. These typical scenarios are called use cases. Although use cases cannot cover all the possible actions on an AUT, they serve the most important functions of the AUT. Due to the huge permutations of events sequences, many existing methods try to reduce the number of test cases by limiting the length of each test case to certain steps, normally three to finish the tests in a practical time. Three steps are normally far from sufficient to cover a task. To efficiently test tasks, long test cases are inevitably needed. By using a use case as the backbone, an envelope model was

developed to encapsulate all possible branches of states and events related to a task. Corresponding to the backbone within the envelope, task-oriented test cases can be generated automatically for effective and efficient testing within a practical time frame. Use case oriented test case generation will be discussed in Chapter 5.

3.8 Test Oracles

Once test cases have been generated, they can be executed on the GUI automatically. Obviously, if the verification of test results is done manually, the process cannot be called 'automatic'. Checking whether the GUI behaves correctly is usually done by a mechanism called a test oracle.

In GUI testing, the inputs are events, but there are no special outputs. The state of the GUI can be seen as an output. Each event in the test case affects the state of the GUI. To check the expected states, state information needs to be compared after each event is performed. Otherwise, incorrect GUI behavior for one event may result in a state in which future events in the sequence cannot be executed at all. Automated oracles need to be developed to answer the question of whether a GUI execution under a test case behaves as expected. Both the derivation of the expected state information and the comparison between the expected and actual states are supposed to be automated.

Generating the expected state after each event is critical for automated oracle development. Expected state can be either generated from the AUT's specifications or

gathered from a base version of the software for regression testing. Figure 3.15 and Figure 3.16 show the two mechanisms of generating expected states respectively.

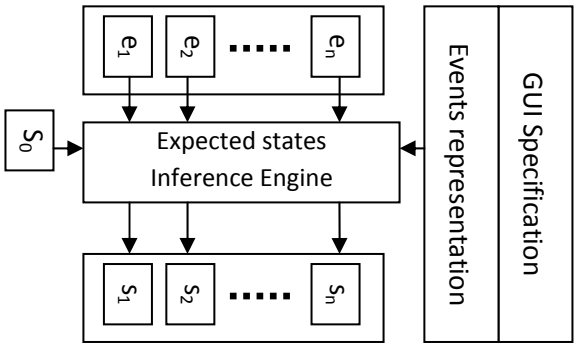


Figure 3.15 Mechanism of generating expected states from specifications

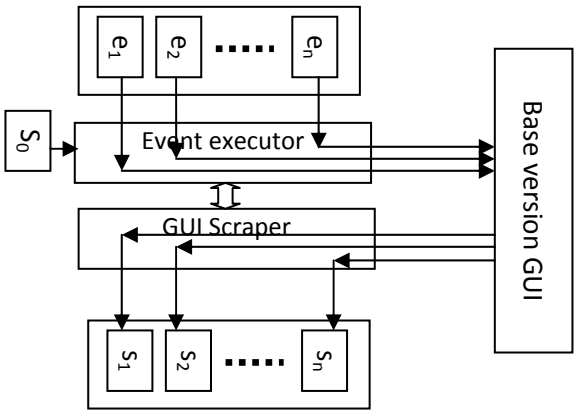


Figure 3.16 Mechanism of generating expected states from base version of AUT

3.8.1 Expected states generation from AUT's specifications

Events in the GUITAM are modeled as state transitions. Each transition $t = <s, s', \hat{e}>$ is related to one event or a small set of events which indicates an operator. For example, a transition is related to an operator called "Move an Icon", it may involve a set of elementary events including 'press mouse left button on the icon', 'move mouse',

'release mouse button'. An event is usually linked to an object, which is a widget or component in the GUI. To enable the automatic expected states generation, each possible operator needs to be well represented in the specifications of the AUT.

Definition 3.10: An ***operator*** is a 5-tuple <$\varkappa$, ρ, $\hat{e}$, *pre*, *post* > where:

- $\varkappa$ is the name of the operator. It can be an empty string because an operator will be identified by $\hat{e}$.
- ρ is a set of parameters. For example, "Move an Icon" may need the coordinates and distance.
- $\hat{e}$ is one or a set of events.
- *pre* is a set of literals. Each literal is an assertion <*o, p, v, true*|*false*>. Where *o* is an object, *p* is the property name of the *o*, v is the value of the *p*, *true*|*false* indicates the assertion must be *true* or *false*. For example, a literal *l*=<*Lable1, Text, "Name", true*> means the value of property 'Text' of object "*Label1*' needs to be "*Name*". *pre* represents the set of preconditions. Before an operator is performed, the literals in *pre* must be satisfied in the state *s*.
- *post* is also a set of literals. Each literal is an assertion <*o, p, v, true*|*false*>. After the operator is performed, the state *s'* needs to satisfy all the literals in post. If there are any literals which are not satisfied, there is a potential defects.

From Definition 3.10, the expected states *s'* can be derived from *s*. Algorithm GenerateExpectedStateFromSpecifications in Figure 3.17 shows the procedure of deriving *s'* from *s*.

Algorithm GenerateExpectedStateFromSpecifications

```
1)  GenerateExpectedState(s:GUIState, op:Operator, s':GUIState)
2)  {
3)      foreach literal l in op.pre
4)          if( s doesn't satisfy l) return error.
5)      s' =DeriveFrom( s);
6)      foreach literal l in op.post
7)          if l satisfies <_,_,_,true>
8)              if(s' contains l.o)
9)                  Set (s', l.o, l.p, l.v);
10)             else
11)                 AddTo(s', l.o, l.p, l.v);
12)         if l satisfies <_,_,_,false>
13)             if(s' contains l.o)
14)                 SetIgnore (s', l.o);
15) }
```

Figure 3.17 Algorithm of generating expected states from specification

In Figure 3.17, lines 3-4 check the pre-conditions to see if the state *s* is the state that *op* needs. Line 5 derives basic information of *s'* from *s*. Lines 7-11 read all the positive literals and make modifications to existing objects in or add new objects to *s'*. Lines 12-14 set ignorance of comparing the corresponding property values.

3.8.2 Expected states generation from the base version of the AUT

Regressive GUI testing is used for testing an AUT over a series of the AUT's versions. When an AUT is modified, to ensure that the modifications don't affect other parts of the AUT which are not modified, a regressive test is needed to perform test cases on both the previous version (base version) and the later version. In regressive GUI testing, the expected states can be easily generated from the base version by performing the test case on it and retrieving the expected state information. Figure 3.18 shows the algorithm of generating expected states from a base version of an AUT.

Algorithm GenerateExpectedStateFromBaseVersion

```
1)  GenerateExpectedState(Ť :TestCase )
2)  {
3)      NavigateTo(Ť.s0);
4)      for i=1 to n
5)      {
6)        Perform (ti in Ť.τ);
7)        si=readState();
8)      }
9)  }
```

Figure 3.18 Algorithm of generating expected states from base version of AUT

3.9 Implementation and Experiment

To evaluate the effectiveness and efficiency of the GUITAM model for GUI testing automation, a system called GUITAM Runner was implemented. GUITAM Runner is implemented in Microsoft C# in the platform of Microsoft Visual Studio .Net 2008. This system contains several modules:

1) The GUI Scraper is a module that automatically scrapes information from an application's GUI and gathers all information of the widgets (objects), including the set of open windows, the hierarchical organized object trees and the corresponding values of properties of each object. This module provides the basic techniques for reading information of a GUI, creating a GUITAM model for an AUT, generating regressive test oracles etc.

2) The Event Performer is a module that can perform events on behalf of a real user automatically. Given an event, e.g., Click on Button 'B1' in Window 'Form1', the Event Performer will automatically find out the button captured 'B1', calculate the clickable area and perform the mouse click on the object. This is the elementary module for traversing the GUIs of an AUT, executing test cases or whatever.

3) The GUITAM Generator is a module that automatically generates a GUITAM model for an AUT.

4) The Test Case Generator uses the GUITAM model and certain coverage criteria to create test cases

5) The Test-Oracle Generator automatically executes the generated test cases on the latest GUI version and stores the captured states related to the test cases for regressive testing. Test-oracle information can also be generated by deriving the expected states from specifications.

6) The Test Executor executes an entire test suite automatically on the AUT. It performs all the events recorded in the transitions of the test cases and compares the captured live state information with the corresponding oracle information. The difference between the states will be reported as potential defects.

Most current GUI testing tools are based on Capture/Replay(C/R) techniques, with the support of test scripts. The C/R technique is based on the GUI object level rather than just mouse coordinates, which increases the stability of the testing. However, without a GUI model, these C/R based tools cannot support real GUI testing automation. Highly skilled professionals are needed to prepare the test scripts, and manual work is needed for recording the test cases. The generation of test cases and test oracles is still laborious in these C/R based tools. After many years research, the team led by Professor Atif M. Memon at the University of Maryland developed GUITAR, which is now an open source project on source forge [129]. GUITAR is an EFG model based tool which is developed in Java environment. As EFG is claimed to be the first practical GUI testing model, GUITAR is also seen as the first practical model based GUI testing tool. GUITAM is the core of this research and also provides a comprehensive method for modeling GUIs. GUITAM Runner is a GUITAM based GUI testing platform which provides a unified solution to GUI testing automation. GUITAM Runner was developed in the Microsoft Visual .Net environment in C#. It is currently still in the laboratory stage. To the best of my knowledge, there are no mature commercial products in the software market yet which support a comprehensive GUI testing model.

To illustrate the difference between C/R based tool and GUITAM Runner, a typical C/R based tool, Automation Anywhere [130], is selected for the comparison. Table 3.3 compares the differences between GUITAM Runner and Automation Anywhere.

Table 3.3 Comparison between GUITAM Runner and Automation Anywhere

Testing stage	**GUITAM Runner**	**Automation Anywhere**
GUI Model Creation	GUITAM model of an given AUT is generated automatically	Doesn't have this stage
Test case Generation	1) Automatic generation 2) One test case is a route of transitions in the model. 3) Coverage criteria are used	1) Manual generation by recording user operations 2) Manual editing 3) No coverage criteria
Test oracle Generation	Automatically generated from the base version of the given AUT.	Manually edited or assigned.
Test case Execution	Automatic	Automatic
Test Reporting	Automatic	Automatic

3.9.1 Subject Applications

Four subject applications are used in this study. Two of them are from an online code source named CodeProject. Calculator [103] is implemented in C++ and EasyWriter [104] is implemented in C#. The other two are from my previously built applications. EnglishStudy is implemented in C# and ScreenDrawer is implemented in C++. Table 3.4 shows the information for the subject applications.

Table 3.4 Subject applications

Subject application	Windows	Objects	LOC
Calculator	2	73	1580
EasyWriter	9	804	988
EnglishStudy	3	217	3729
ScreenDrawer	2	127	3423

In Table 3.4, LOC means Lines of Codes. A brief introduction is given to each of the subject applications.

1) Calculator is an application which provides calculations and mathematic functions. In the main interface, besides one textbox for showing the calculation result, other components are mainly buttons. These buttons can be divided into groups: parameter buttons which include '0' ~ '9', 'A'~'F', operators which include '+','-','*', '/' and the like, mathematical functions including 'sin', 'cos' etc. and some statistics functions which include 'stat', 'avg', 'sum' and so on. Figure 3.19 shows the main GUI of Calculator.

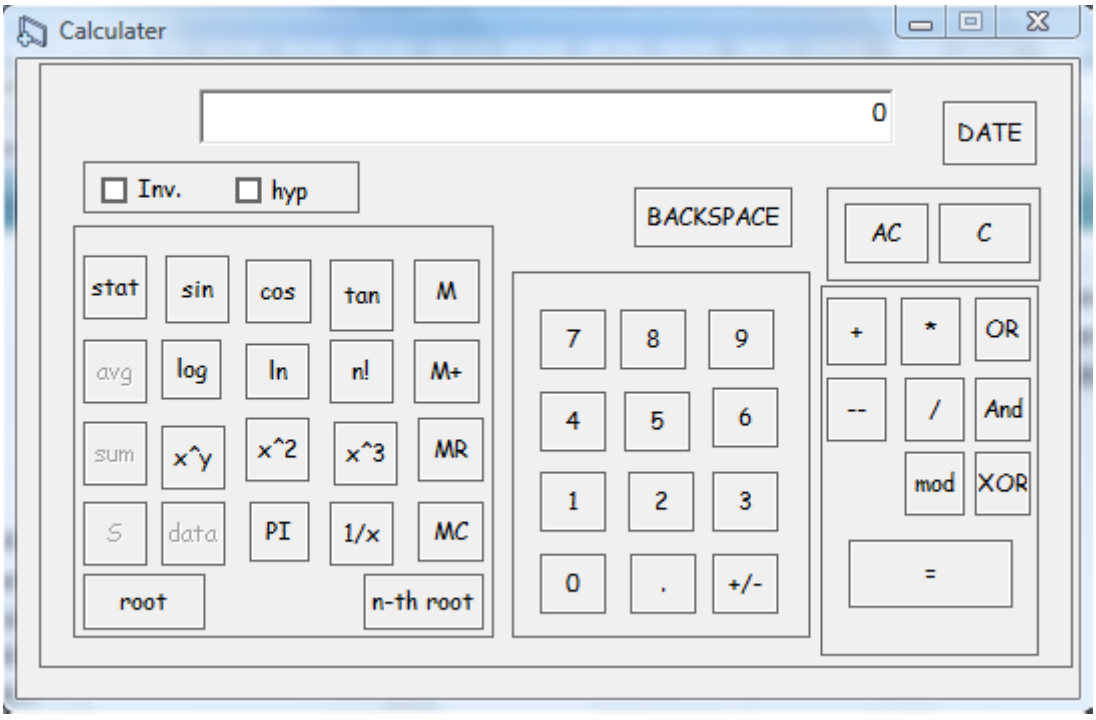


Figure 3.19 Main interface of Calculator

2) EasyWriter is a text editor which is much more powerful than Microsoft Notepad. Besides basic text editing functions, it also provides many other useful functions for easy writing. These functions include: auto indent, auto colored key words for certain programming languages, zooming, and text encoding selection. Figure 3.20 shows the main interface of EasyWriter.

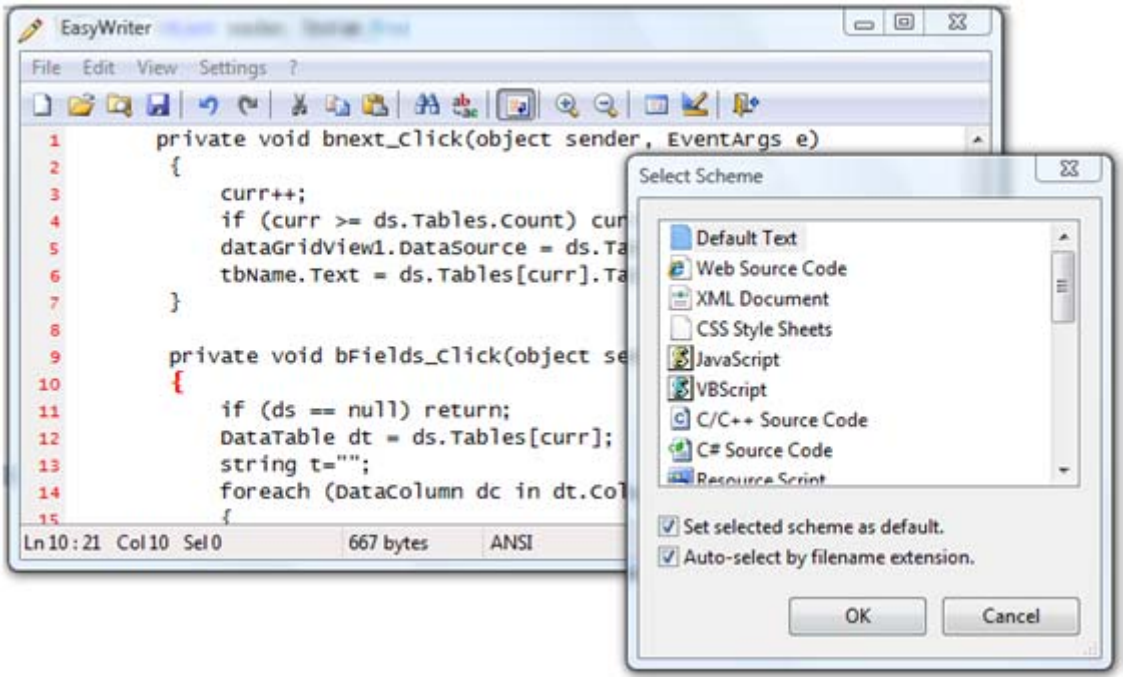


Figure 3.20 Main interface of EasyWriter

3) EnglishStudy is an application to help Chinese pupils study English, especially for studying words and sentences. Functions provided by this application include creating new books, management words and sentences for each book, auto importing from text files, hiding or showing certain language while studying, and

speaking the sentences using TTS (Text to Text Speech). Figure 3.21 shows the main interface of EnglishStudy.

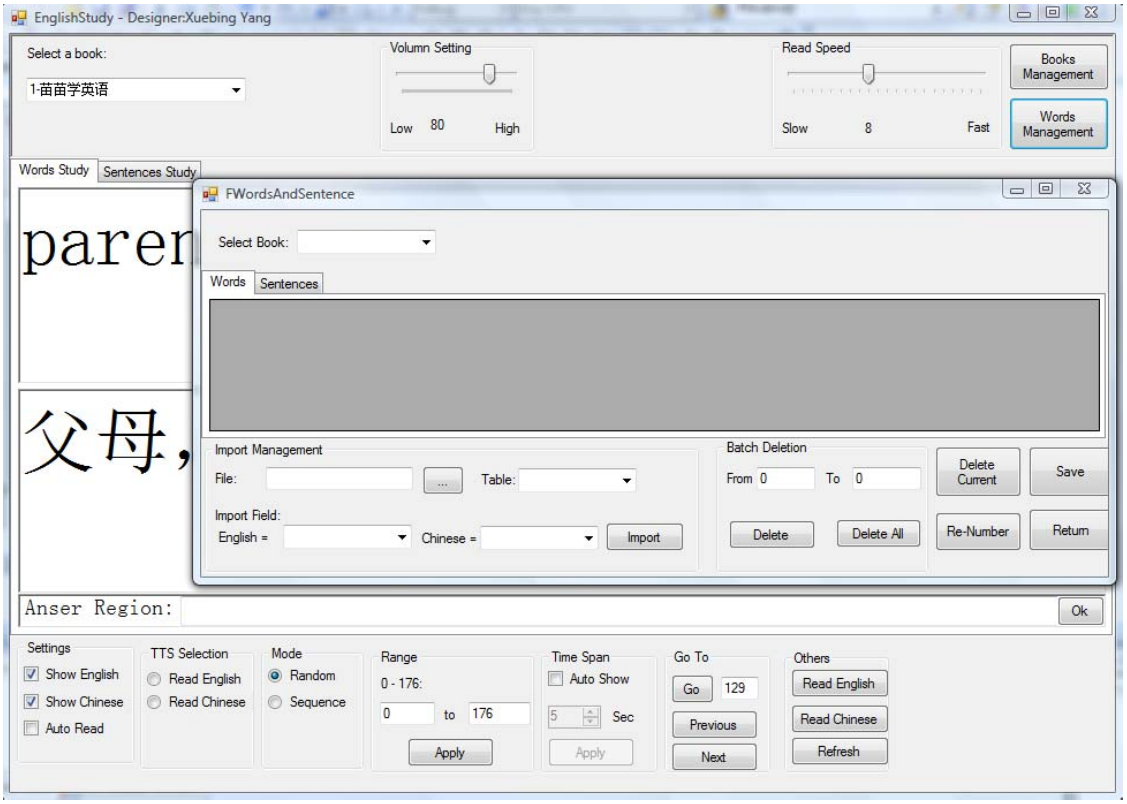


Figure 3.21 Main interface of EnglishStudy

4) ScreenDrawer is an application which was developed for presenting information on computer screens to large audiences. For example, it is used by teachers to introduce particular part on the screen of running software, such as a statistics tool, a programming platform, a specific part of a video and so on. It appears as an icon as shown in Figure 3.22. You can use your computer in any way you want, and when you need to present a certain part of the screen, you just need to click on the

icon or press hot key ‘Ctrl-F1’ to start free drawing mode on the screen. The main functions of this application include free drawing, selecting any part, moving the selected part or enlarging the part, typing words on the place you last clicked the mouse button, drawing lines, rectangles, circles, diamonds and so on. You can also use it as a screen capture tool to copy a selected part or the whole screen to the clipboard. By pressing ‘Ctrl-F1’ again, the computer will go back to normal mode for performing common tasks. Figure 3.22 shows the drawing mode of ScreenDrawer.

Figure 3.22 the Icon of ScreenDrawer

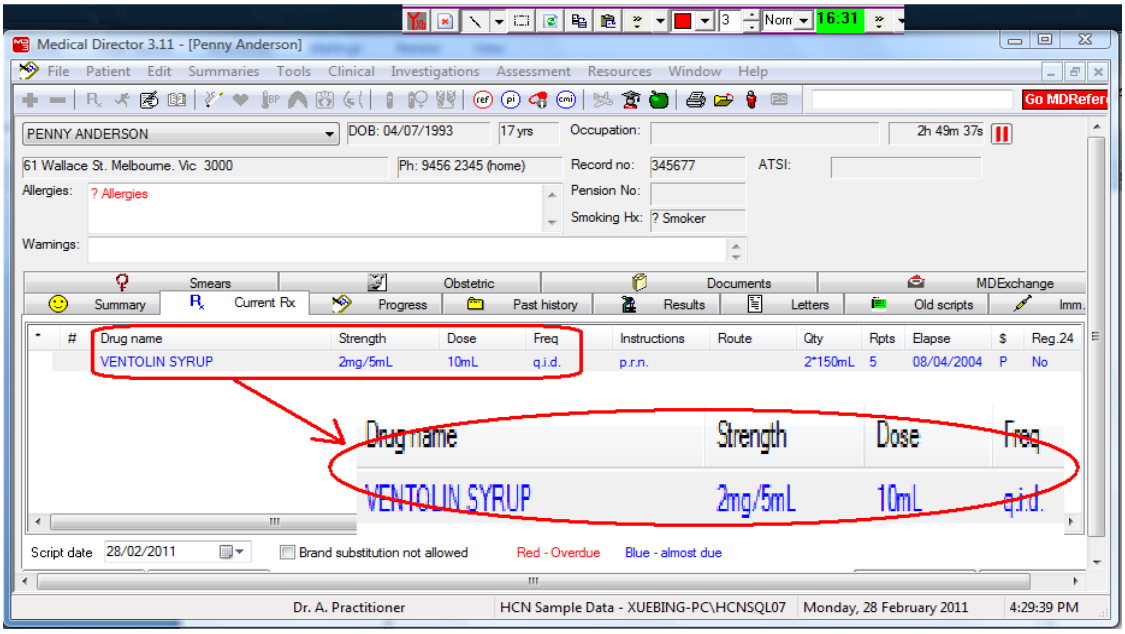


Figure 3.23 The drawing mode interface of ScreenDrawer

3.9.2 Automatic GUITAM generation

GUITAM Generator is used to automatically create GUITAM models for each subject application. Table 3.5 shows the information for GUITAM for each subject application.

Table 3.5 GUITAM information of subject applications

Subject application	States	Transitions	Size (MB)
Calculator	42	586	1.2
EasyWriter	32	222	0.5
EnglishStudy	12	362	0.4
ScreenDrawer	13	136	0.26

3.9.3 Test case generation

The test case generator uses a given GUITAM model of a certain subject application and coverage criteria to generate test cases for each subject application. Table 3.6 shows the number of test cases generated from the GUITAM models of the subject applications. These test cases are generated using the length-n coverage criterion which is defined in Definition 3.10.

Table 3.6 Test cases generated for each subject application

Subject application	Length								Total Selected
	1		2		3		4		
	Full	Sel	Full	Sel	Full	Sel	Full	Sel	
Calculator	74	74	4989	1022	327648	2356	21484904	4554	8006
EasyWriter	10	10	84	84	752	752	6580	1125	1971
EnglishStudy	68	68	2850	853	543332	2682	23657452	4248	7851
ScreenDrawer	7	7	382	382	4852	1016	974258	3762	5167

In Table 3.6, the column ‘Full’ indicates the number of all possible test cases with the given length (in one test case, no repeated events) and the column ‘Sel’ indicates the

actual number of test cases generated. Due to the large space of all possible test cases, only a practical number of test cases are selected. Note that so many events are used for editing, changing font, changing color, etc., when selecting the test cases, only one or two typical events of the type are kept for test case generation.

3.9.4 Oracle information

Because all the subject applications have source code, the original version of the application is seen as the base version and oracle information is generated from the base version automatically. Test-Oracle Generator executes all test cases in a given suite, and reads and records the GUI real time state information after each event in the test case is performed. The recorded state information is linked to certain test cases as the test oracle information for comparison with the state when the same test case is executed on the modified version of the application. Table 3.7 shows the oracle information of each subject application for the test suite shown in table 3.6.

Table 3.7 Oracle information for each subject application

Subject application	Test cases		Oracles size (MB)
Calculator	1	74	3.5
	2	1022	65
	3	2356	162
	4	4554	412
	Total	8006	642.5
EasyWriter	1	10	0.12
	2	84	1.75

	3	752	19
	4	1125	42
	Total	1971	62.87
EnglishStudy	1	68	1.8
	2	853	17
	3	2682	134
	4	4248	378
	Total	7851	530.8
ScreenDrawer	1	7	0.1
	2	382	7.1
	3	1016	31
	4	3762	126
	Total	5167	164.2

3.9.5 Test Executor

Test executor is used to execute all test cases in a given test suite for an AUT. Since the oracles of test cases have been generated, when test cases are performed on a modified version of the AUT, real time state information will be captured by the GUI Scraper and compared to the corresponding state information described in the oracles. Fault seeding technique is used to seed faults into the source codes for evaluating the effectiveness and efficiency of our approach. Figure 3.24 shows the number of faults seeded for each subject application.

To compare with the faults detection effectiveness of EFG presented by Memon and Xie [37], we seeded the faults using the same classes and criteria. The classes for fault

seeding are shown in Table 3.8 and the numbers of faults seeded for each class listed above are shown in Figure 3.25.

Table 3.8 Types of seeded faults

Type No.	Type Description	Related Operators
1	Modify relational operator	>, <, >=, <=, ==, !=
2	Invert the condition statement	
3	Modify arithmetic operator	+, -, *, /, =, ++, -, +=, -=, *=, /=
4	Modify logical operator	&&, ||
5	Set/return different Boolean value (true, false)	
6	Invoke different (syntactically similar) method	
7	Set/return different attributes	
8	Modify bit operator	&, |, ^, &=, !=, ^=
9	Set/return different variable name	
10	Set/return different integer value	
11	Exchange two parameters in a method	
12	Set/return different string value	

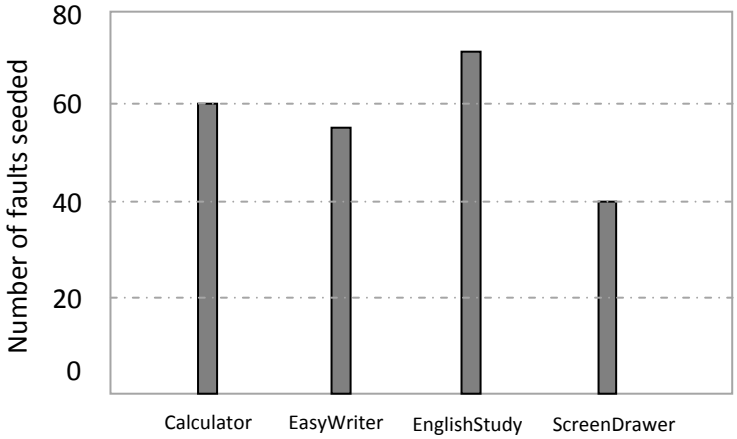


Figure 3.24 Number of faults seeded to the subject applications

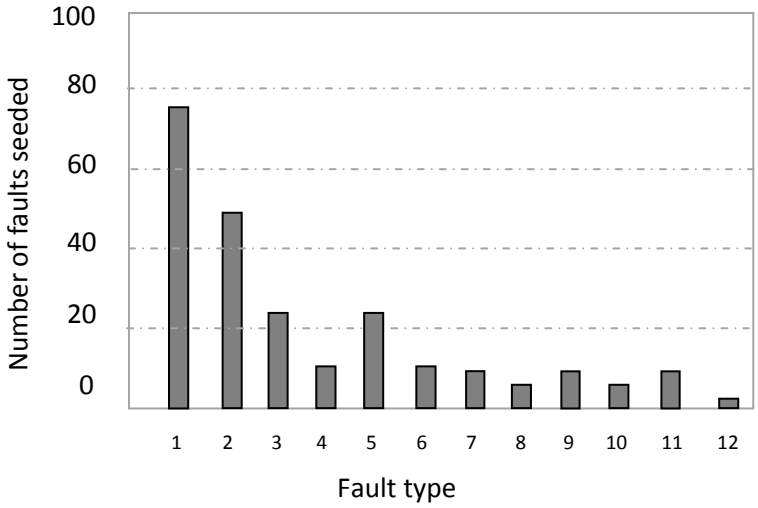


Figure 3.25 Number of faults seeded to each type

When comparing between oracle state information and runtime state, any mismatches will be reported. Some property values are ignored due to their dynamic characteristics. For example, an object such as a textbox for showing a timer changes overtime. A window position changes every time when it is opened. In an online media player, or share prices from the Internet, and the like, contents change over time. If all these dynamic contents are compared, mismatches are inevitable. All these kinds of objects are ignored for comparing before the test cases are executed.

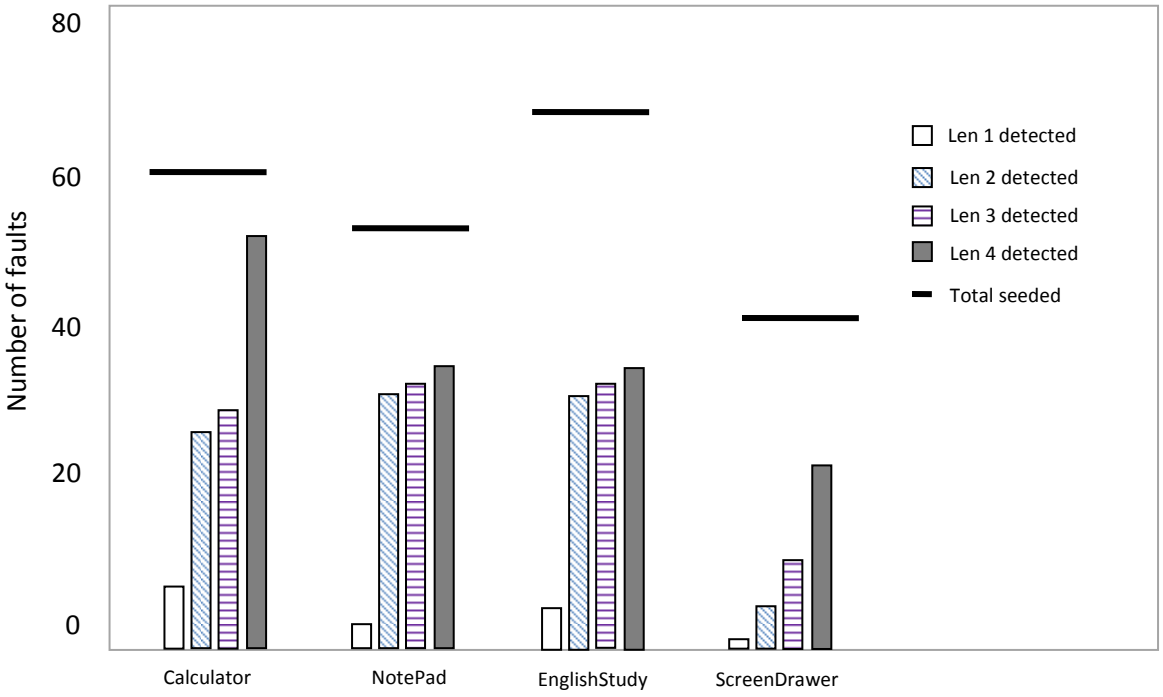


Figure 3.26 Faults detected by different length of cases

Figure 3.26 shows the fault detection results for each subject application with different collection of test cases. We found that the test cases with length 1 had the least ability to detect faults. The reason for this is not only the steps, but also the small number of test cases. Because every test case starts from the initial state, many other states cannot be reached by only one step. From the test case information in Table 3.5, we can see that more test cases were generated for longer test cases. Longer test cases can reach more states so that more defects are detected. In the Calculator testing, test cases with lengths 2 and 3 had similar fault detecting ability, while the test cases with length 4 had much better fault detection effect. This is because most of the functions provided in Calculator need at least 4 steps, for example, 3 + 5 = 8, which needs at least 4 clicks to

get the result 8. EasyWriter and EnglishStudy are MDI based applications in which most of the states can be reached by fewer steps from the initial state. In these two applications, test cases with lengths 2, 3 and 4 had similar effectiveness for detecting faults. Screen drawer is a dialog-based application, which means that the states are in a hierarchical manner (dialogs are opened by other dialogs and monopolize the focus), and longer test cases can reach more states than shorter test cases. From the results shown in figure 3.27, longer test cases have significantly more effect in detecting defects. Figure 3.28 shows the fault detection effects for different lengths of test case for each subject application.

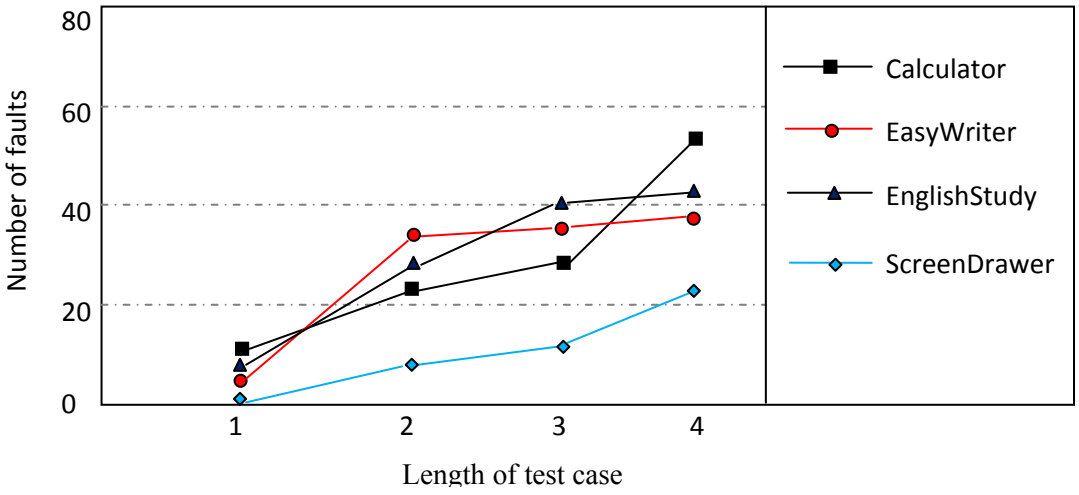


Figure 3.27 Faults detect effects for different length of cases

The detected faults may also be classified by fault type. Figure 3.28 summarizes the results. The yellow part of each column shows the number of faults detected, and the gray part of each column shows the number of faults not detected. The result shows that

GUITAM-based testing is able to detect all types of the faults as long as the defects lead to property value changes of GUI objects. Because not all seeded faults reflect their changes to the GUI object property values, some of them cannot be detected. This was not an exhaustive test. Not all paths of the codes were tested, and some faults were missed due to coverage reasons.

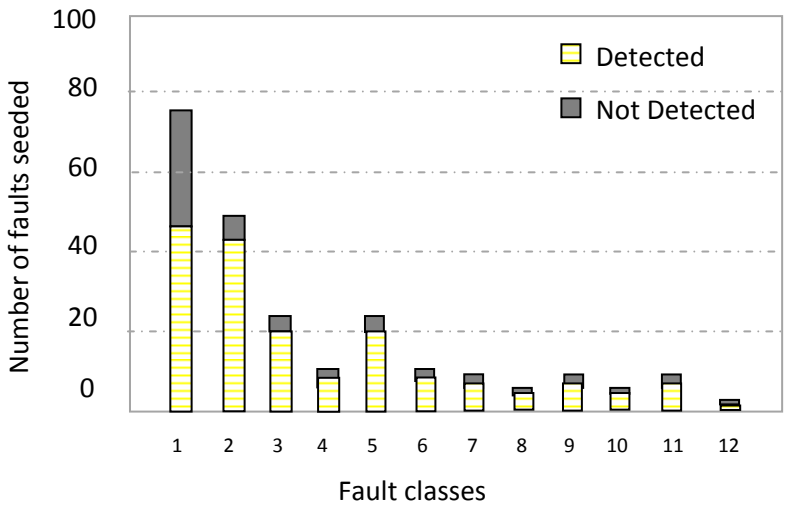


Figure 3.28 Faults detected for each fault type

In contrast with test the results presented by Memon and Xie in [37], GUITAM is able to find all types of defects. For the first three types, the proportion of detected faults to all the seeded faults in this experiment is about 15% better than the proportion of detected faults shown by Momon and Xie [37]. Figure 3.29 shows the results of comparison between percentages of each fault type detected by GUITAM and EFG. The data of faults detected by EFG is from Memon and Xie [37].

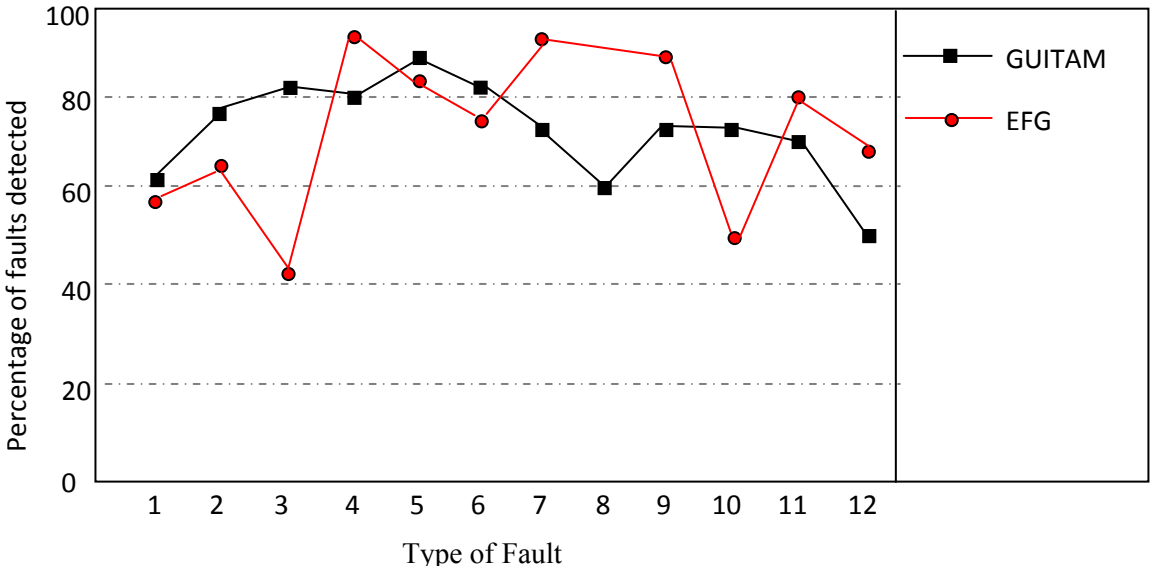


Figure 3.29 Percentage of detected faults for each type of faults

3.10 Conclusion

This chapter presented a GUI Testing Automation Model (GUITAM). In this model, both GUI states and events are considered. It has been proved that GUITAM is not only able to automate all testing that an EFG can automate, but it is also able to model a series of important scenarios which an EFG cannot. The efficiency of GUITAM, in terms of storage and computational complexity, has also been proved to be at least as good as that of the EFG model.

This chapter also presented the GUI testing automation procedure with GUITAM. This procedure includes GUI test coverage criteria, GUI test case generation, test oracle creation and test execution automation. The corresponding principles and algorithms were also provided for GUITAM based testing automation.

Empirical study also shows the ability for and efficiency of automatically testing GUI-based applications with the GUITAM model. GUITAM models were generated automatically for four subject applications. Test cases with different lengths were also generated automatically from the generated GUITAM model. Test oracle information was generated by executing the test cases on the original applications and capturing the runtime state information. To test the effectiveness of detecting faults, faults were seeded to the source codes of the applications and test cases were executed on the modified applications automatically. The test results show that the automatic process of GUI testing is not only practical but also effective. From the results, we also found that the efficiency and effectiveness of GUI testing are greatly dependent on the set of test cases. The following chapters will present new methods of generating more efficient and effective test cases.

Chapter 4

Defect Classification

Today's GUIs are becoming more and more complex and contain a large number of objects and events. The number of permutations of all possible event sequences increases exponentially as the number of events increases. Defect classification can allow the GUI testing to focus on certain types of defect, and greatly reduce the number of test cases needed. Some researchers have carried out software defect classification for different reasons [105-109], but few of them have linked defect classification to GUI testing automation. Defects can be classified into different groups and each group is usually linked to a certain set of events and objects. By classifying the defects, test cases can be generated specifically for finding certain types of defect. The total number of test cases needed will decrease exponentially as number of the relevant objects and events become smaller.

4.1 Introduction to Defect Classification

GUI defects can be roughly classified into three groups: directly detectable defects, undetectable defects and comparably detectable defects.

Directly detectable defects are those which can be detected without involving an AUT's specifications and base version information. These defects usually include crashes, and recognizable error messages. Crashes cause the AUT to stop responding to inputs.

Recognizable error messages are usually the system error dialogs or specifically customized message boxes which pop up while the AUT is running. This type of defect can be detected by random testing tools. When these types of defect occur, they can be easily recognized, and no specifications or regression information are needed. Random input testing is also referred to as “stochastic” testing or “monkey” testing [93]. The latter designation is used to impart the idea of someone without a brain, or without knowing what he's doing, seated in front of a computer and interacting randomly with the keyboard or mouse. Microsoft has reported that 10-20% of the bugs in their software projects are found by monkey test tools [93].

Undetectable defects are those defects that GUI testing automation is not able to detect. Because GUI defects are manifested as failures observed through the GUI, and only the effects on the GUI can be extracted, the defects whose results are not reflected in the GUI are not detectable. Examples of this type of defect include wrong files being saved with a successful process, emails being sent successfully but failing to reach their destinations, wrong data being saved into background databases with successful responses, and background thread execution without outputs to the foreground GUI. These types of defects require testing via direct methods and are primarily conducted manually.

Comparably detectable defects are the type which GUI testing automation principally deals with. This kind of defect can be detected by comparing the actual state affected by the defects with the expected state. The expected state can be a state from a base

version in regressive testing, or derived from the specifications of an AUT. When a test case is executed, the actual states will be compared with the expected results and the differences will be reported.

Some researchers have worked on defect classification with particular focuses. L. Briand and Y. Labiche, and I. Krsul [111, 112] have addressed defect classes at different phases of the development life cycle, e.g., defects in the requirements specification document. K. Weidhenhaupt et al. [113] have focused on defects in e-commerce applications. M. L. Lough and G. J. Meyers [114, 115] have focused on taxonomies for security issues. In general, typical defect classes of an AUT are listed as follows [105]:

- Completeness defects subsume all defects related to an incomplete implementation of the specified functionality. This usually includes missing functionality defects and undesired functionality.
- Input/Output (I/O) defects subsume all defects related to wrong input respectively to wrong output data of the AUT. Boundary defects, defects concerning wrong size, shape or format of the data or combination defects are typical I/O defects.
- Calculation defects subsume all defects resulting from wrong formula or algorithms in the AUT.
- Data handling defects subsume all defects related to the lifecycle and the order of operations performed on data. This usually includes duplicated data or dataflow

defects (defects related to the sequence of accessing a data object (e.g., a data update before the data has been created).

- Control flow and sequencing defects subsume all defects related to the control flow or the order of actions. Typical control flow defects concern wrong sequencing of the actions performed or iteration and loop defects, which subsume all defects related to the control flow of iterations and loops.
- Concurrency defects subsume all defects related to the concurrent execution of parts or of multiple instances of the AUT.
- Display and navigation defects subsume all defects related to the user interface, which are not usability defects. Typical defects of this class are display defects and navigation defects.
- NFR (non-functional requirement) defects subsume all defects related to the quality of the AUT. This usually includes defects concerning reliability, usability, efficiency, maintainability and portability.
- Inter-Software defects subsume all defects concerning the interface of the AUT with other software systems. Typical defects of this class are input/output defects, concurrency defects or completeness defects such as missing third-party software.
- Other defects subsume all other defects including hardware defects, test design defects, OS and compiler defects etc.

The defect classes are not orthogonal, i.e., a defect can be categorized into more than one class. Additionally, a defect can also be associated to a combination of defect

classes. Different types of defects accounts for different percentages of the total defects in software systems. By analyzing and re-organizing the results of Brooks, Robinson and Memon [53, 108], the distribution of different types of defects is shown in Table 4.1. Not all classes of defects mentioned above are detectable during GUI testing automation.

Table 4.1 Distribution of defect types

Fault Type	**Defects (%)**
Completeness	13.53
I/O	6.5
Calculation	11.69
Data handling	17.34
Control flow and sequencing	17.86
Concurrency	5.18
Display and Navigation	8.52
NFR	4.26
Inter-software	4.85
Other	10.27

Among these classifications, Completeness defects, Calculation defects, Input/Output (I/O) defects, Display and navigation defects are detectable using GUI testing automation. Inter-Software defects are not, unless they manifest as crashes or typical error messages. Data handling defects, control flow and sequencing defects, concurrency defects and, NFR defects are usually detected by traditional software testing methods which involve large amounts of manual analysis and observation. My taxonomy will be restricted to defects which can be detected during GUI testing automation, about 40% of all the software defects listed in Table 4.1. Other types of non-GUI relevant defects are supposed to be dealt with by other testing techniques. For

example, traditional variable boundary value testing shall be tested with conventional white-box testing techniques rather than with GUI testing techniques.

4.2 Types of Objects

GUI defects may be detected by executing GUI test cases. Each GUI test case is composed of a series of events. Each event is linked to an object. Take, for example, a test case τ = < "(*Edit*$_1$, Focus), (*Edit*$_1$, Type String), (*Edit*$_1$, SelectAllText), (*CopyButton*, Click), (*Edit*$_2$, Focus), (*PasteButton*, Click) >. In this test case, each event is linked to a GUI object. After the test case is executed, object properties will be examined to ensure the correctness of the effects. Objects are the basic and important elements of a GUI. Different types of object are usually linked to different kinds of function, and thus different kinds of defect. Obviously, it is straightforward to classify GUI defects according to GUI objects.

Although the number of object types keeps changing, as new types of objects being introduced into developing environments, the set of basic types of objects is relatively stable. By analyzing popular software and development tools, a summary of objects used nowadays in GUI-based applications is presented. Table 4.2 lists the most commonly used types of object and their corresponding events.

Table 4.2 Object types and their related events

Object Type	Main possible events	Class	Possible test actions
Form (Window)	① ② ③ ④ ⑤ ⑥ ⑦ ⑧	Top container	① ②

Dialog	① ② ③ ④ ⑤ ⑥ ⑦ ⑧	Top container	① ②
Open File Dialog	① ② ③ ④ ⑤ ⑥ ⑦ ⑧ ⑭ ㊲	Common Dialog	① ②
Save File Dialog	① ② ③ ④ ⑤ ⑥ ⑦ ⑧ ⑭ ㊲	Common Dialog	① ②
Color Dialog	① ② ③ ④ ⑤ ⑥ ⑦ ⑧	Common Dialog	① ②
Font Dialog	① ② ③ ④ ⑤ ⑥ ⑦ ⑧	Common Dialog	① ②
ReBar	⑮ ㊳	Static	
Button	⑦ ⑧ ⑯	Functional	⑦
Tool Button	⑦ ⑧ ⑯	Functional	⑦
Edit	⑦ ⑧ ⑭ ⑯ ⑱ ⑲ ⑳ ㊲	Interactive	⑭ ⑱ ⑲ ⑳
Pane	③ ⑦ ⑧ ⑨ ⑮	Static	⑧
ComboBox	⑭ ㉕ ⑬	Composite interactive	⑭ ㉕ ⑬
List	⑦ ⑧ ⑨ ⑬ ㉕	Functional container	⑬
ListItem	⑦ ⑧ ⑨ ⑬	Functional responsive	⑦ ⑧
Text (Label)	⑦ ⑧	Static	⑧
DataGrid	⑦ ⑧ ⑬ ㉒ ㉓ ㉔ ㉖ ㉕ ㉗	Functional responsive	⑦ ⑧ ⑬ ㉒ ㉗
Check Box	⑦ ⑧ ㉘ ㉙	Interactive	⑦
Radio Button	⑦ ⑧ ㉘ ㉙	Interactive	⑦
Tab Control	⑦ ⑧ ⑬	Container	⑬
TabItem	⑦ ⑬	Container	⑦

Scroll Bar	⑦ ㉚	GUI adjustment	㉚
Group Box	⑦ ⑧	Static	
Title Bar	⑦ ⑧ ㊳	GUI adjustment	
MenuBar	⑦ ⑯	Functional container	⑦
Context Menu	⑦ ⑯	Functional container	⑦
Menu Item	⑦	Functional	⑦
Status Bar	⑦ ⑧	Responsive	
Document(multi-line Edit, Rich box)	⑦ ⑧ ⑨ ⑭ ⑱ ⑲ ⑳ ㊲	Composite interactive	⑦ ⑧ ⑨ ⑭ ⑱ ⑲ ⑳ ㊲
PictureBox	⑦ ⑧ ⑩ ⑪ ⑫	Responsive	⑦ ⑧
TreeView	⑦ ⑧ ⑨ ⑬ ㉛ ㉜	Functional container	⑦ ⑧ ⑨
TreeNode	⑦ ⑧ ⑨	Functional responsive	
Split, Split Container	㊳	GUI adjustment	
TabLayout	③	GUI adjustment	
Track Bar	㉚ ㉟	Composite interactive	㉚
ProgressBar	⑦ ⑧	Responsive	
Spinner	⑭ ⑦	Composite interactive	
DateTime Picker	⑦ ⑧ ⑭	Composite interactive	⑭

In the columns entitled "Main possible events "and "Possible test actions, circled numbers are used to list the events. Each number represents a type of event:

① Open ② Close ③ Resize ④ Maximize ⑤ Minimize ⑥ Restore ⑦ Mouse Click ⑧ Mouse Double Click ⑨ Mouse Right Click ⑩ Mouse Press ⑪ Mouse

up ⑫ Mouse Move ⑬ Select item ⑭ type string ⑮ Dock ⑯ Hover ⑰ Focus ⑱ copy ⑲ cut ⑳ paste ㉑ Scroll ㉒ Header Click (Row, column) ㉓ Row Resize ㉔ Column Resize ㉕ Scroll ㉖ Row Click ㉗ Row DB Click ㉘ Check ㉙ UnCheck ㉚ Change Position ㉛ Expand ㉜ collapse ㉝ Move the Splitter ㉞ Move the track ㉟ Drag ㊱ Drop ㊲ Select ㊳ Move

The objects listed in Table 4.2 are divided into several classes. Different classes of objects play different roles and are related to different kinds of GUI defects. Some objects play functional roles which interact with users by providing immediate function. Most functional objects such as buttons and menu items are related to underlying codes which provide important functions with complex enterprise logic. Defects within these codes are much more significant than other defects. On the contrary, most static objects such as labels and panes are not related to any functional codes at all. They are normally dragged from the object repository to the forms on the design stage without the need of writing any codes. In GUI testing, these kinds of objects are less important for detecting GUI defects. Table 4.3 show the classification of objects and their importance in GUI testing. Numbers from 0 to10 are used to show the importance, where 0 signifies the least importance and 10 the most.

Table 4.3 Classification of objects

Class of objects	Type of objects	Importance
Functional	Button, Tool Button, Menu Item	10
Interactive	Edit, Check Box, Radio Button	9
Composite interactive	ComboBox, Document, Track Bar, Spinner, Datetime Picker	9
Functional responsive	ListItem, DataGrid, TreeNode	6
Top container	Form , Dialog	3
Common Dialog	Open File, Save File, Color, Font	3
Responsive	Status Bar, PictureBox, ProgressBar	1
Container	Tab Control, TabItem,	2
Functional container	Functional container, Context Menu, TreeView	2
GUI adjustment	Scroll Bar, Title Bar, Split, Split Container, TabLayout	1
Static	Text (Label), Group Box, Pane	0

The functional, interactive, composite interactive, and functional responsive objects are given more importance because most codes are linked to these objects. Forms and dialogs are also important, but less so, because they are just objects containers. On the one hand they are usually opened by a button, menu item or a double click on a list item. On the other, after they are opened, their functions are provided by the objects contained in them. Functions such as 'Close window or dialog' are usually triggered by buttons or menu items within them. Functional objects are always provided with underlying codes to perform certain tasks. Unless disabled, functional objects will be useless without underlying codes. Interactive and composite interactive objects provide ways for users to exchange information with the underlying codes. They expose the software data to users visually, and accept inputs from users which will become

parameters of the functional underlying codes. Usually some of them are provided with validation codes and many others work with the default functionality without any extra codes at all. Different from responsive objects, functional responsive objects usually play multiple roles. Although they are used for displaying data in organized groups such as lists or tables (grids), they are also provided with functional codes. For example, when selecting a list item, the underlying codes change the other objects' properties accordingly. When double clicking on a list item, it may open a dialog for further information. To analyze the objects, information for objects contained in software from Microsoft Office and clinic software Medical Director 2 (MD2) and 3 (MD3) from Health Communication Network [110] was collected. From the results, we found that functional and interactive objects account for about 50% of all the objects in this software. Table 4.4 and Table 4.5 show the details of the objects information in the subject applications.

Table 4.4 Object statistics for Microsoft Offices software

	Word	Excel	PowerPoint	Total	Class	Percentage
BUTTON	81	63	73	217	Functional	43%
TOOL Button	14	10	9	33		
MENU ITEM	14	14	2	30		
EDIT	2	1	0	3	Interactive	1%
CHECKBOX	0	3	0	3		
Document	1	1	1	3	Composite interactive	2%
COMBO BOX	2	4	2	8		
LIST ITEM	19	0	20	39	Functional responsive	7%
LIST	1	1	2	4		

WINDOW	1	1	2	4	Top container	1%
PICTURE	1	3	1	5	Responsive	1%
TAB	1	1	1	3	Container	4%
TAB ITEM	7	9	8	24		
MENU BAR	2	2	2	2		
SPLIT	43	52	28	123	GUI adjustment	21%
TITLE BAR	3	3	3	9		
SCROLL BAR	2	2	1	5		
PANE	22	21	18	61	Static	11%
LABEL	5	1	5	11		
Total	260	219	178	657		

Table 4.5 Object statistics for MD2 and MD3

	MD3	MD2	Total	Class	Percentage
BUTTON	196	247	443	Functional	22%
TOOL Button	0	0	0		
MENU ITEM	126	123	249		
EDIT	80	73	153	Interactive	12%
Spinner	0	1	1		
Radio Button	29	31	60		
CHECKBOX	86	96	182		
Document	15	12	27	Composite interactive	2%
COMBO BOX	26	19	45		
LIST ITEM	192	181	373	Functional responsive	14%
Data Grid	15	6	21		
LIST	27	22	49		
WINDOW	2	2	4	Top container	1%
Dialog	7	9	16		
PICTURE	6	10	16	Responsive	1%
Status Bar	2	1	3		

TAB	3	3	6	Container	2%
TAB ITEM	18	27	45		
MENU BAR	10	8	18		
SPLIT	19	9	28	GUI adjustment	4%
TITLE BAR	9	11	20		
Thrumb	13	21	34		
SCROLL BAR	21	34	55		
PANE	24	30	54	Static	42%
GroupBox	14	34	48		
LABEL	761	490	1251		
Total	1701	1500	3201		

4.3 GUI object based Defect Classification

According to Tables 4.4 and 4.5, about half of the objects in a GUI application are static objects, GUI adjustment objects and simply responsive objects. This kind of object is usually not related to any underlying codes. To reduce the test case space for more efficient GUI testing, the test cases should mainly focus on the functional and interactive objects. Suppose there are *n* functional and interactive objects, then the number of total objects is about *2n*. Using the *component coverage criterion* together with the *length-k criterion*, and supposing only one event will be selected from an object, then the number of all the possible test cases is $N_{all} = C_{2n}^{k} = \frac{(2n)!}{k!(2n-k)!} = \frac{(2n)(2n-1)(2n-2)\ldots(2n-k+1)}{k!}$.

Because inputs such as clicks and double clicks on the static and responsive objects normally don't lead to any change to the state, testing these objects won't help find any

defects. If only the functional and interactive objects are considered, then the number of total possible test cases is

$$N_f = C_n^k = \frac{n!}{k!(n-k)!} = \frac{n(n-1)(n-2)\ldots(n-k+1)}{k!}.$$

To show the difference, let's analyze an example. Suppose there are 10 objects in one form, say 5 buttons and 5 labels, and k=3. If bout 5 of them are functional and interactive objects, then

$$\frac{N_f}{N_{all}} = \frac{n(n-1)(n-2)\ldots(n-k+1)}{(2n)(2n-1)(2n-2)\ldots(2n-k+1)} = \frac{5\times4\times3}{10\times9\times8\times7\times6\times5\times4\times3} = \frac{1}{30240}$$

The number of all possible test cases for buttons is only 1 in 30240 of the number of all possible test cases for both buttons and labels. Obviously, the bigger the number *n* is, the bigger the difference between N_f and N_{all}. Focusing on a certain type of defect will exponentially reduce the domain space of test cases.

Because in a GUI application, functions are normally triggered by events from functional objects, completeness defects and calculation defects can be detected by test cases with coverage of functional GUI objects. In a GUI application, I/Os are performed through interactive GUI objects such as textboxes, radio buttons etc. Input domain and boundary information can be obtained from the specifications of the AUT. Test cases with coverage of all interactive objects will ensure the efficiency of detecting I/O defects. Display and navigation defects involve responsive, interactive and functional objects. Most display defects reside in responsive and interactive objects.

Navigations are usually led by buttons and menu items. Test cases that cover all functional objects are able to detect most navigation defects.

To simplify the classification of defects, this chapter focuses on the GUI testing automation detectable defects. Considering the characteristics of GUI testing automation, this research focused mainly on three major defect groups based on the types of GUI object: functional defects, interactive defects, and GUI adjustment defects. Other types of defects are ignored in this research because they account for a minor proportion of all defects.

Functional defects subsume all defects that reside in the underlying codes which can be invoked by events from functional objects and have their results reflected in GUI objects and their properties.

Functional defects include defects which are related to functional objects. These objects include Buttons, Tool Buttons, Menu Items, List Items, Data Grid Rows and Cells, and Tree Nodes. Most objects are implemented to perform certain functions and are connected with underlying codes.

Interactive defects subsume all defects which occur during the interaction between a user and the AUT. These include data editing, data automatic validation, etc.

Interactive defects include defects which are related to interactive objects. These objects include EditBoxes, ComboBoxes, CheckBoxes, RichEdits, TrackBars, Spinners, and DateTime Pickers. These objects are usually used for displaying internal variable values

and receive inputs from users. Most of the time, these objects are provided with stereotyped functions without the need for any coding. The values of the properties of these objects are also seen as parameters of the functions related to the functional objects.

GUI adjustment defects subsume all defects which occur during the GUI adjustments such as resizing, re-grouping, scrolling, and changing tab etc.

GUI adjustment defects include defects which relate to GUI adjustment objects. These objects include Scroll Bars, Title Bars, Splits and Split Containers, and Tab Layouts. These objects are normally used to resize the windows or components, scroll the content, or re-arrange the area of the windows. Because the functions provided by these objects are usually provided by the system as standard functions which are well tested before they are deployed, very few defects are related to these objects.

Besides the defects listed above, few defects are related to static objects such as Panes and Labels, and some responsive objects such as ProgressBars. Because this kind of defect accounts for a very minor proportion of the total, we ignored them in this thesis to reduce the test case space. The defects related objects are listed in Table 4.6.

Table 4.6 Defect Classes and their related object types

Defects Class	Related Object Type	Percentage of Total
Functional defects	Button, Tool Button, Menu Item, List Item, Data Grid, TreeView	63%
Interactive defects	EditBox, ComboBox, CheckBox, RichEdit, TrackBar, Spinner, DateTime Picker	29%
GUI adjustment	Scroll Bar, Title Bar, Split, Split Container, Tab Layouts	8%

defects		

Having re-grouped the defect types, the distribution of these three types among all GUI testing automation detectable defects is shown in Figure 4.1.

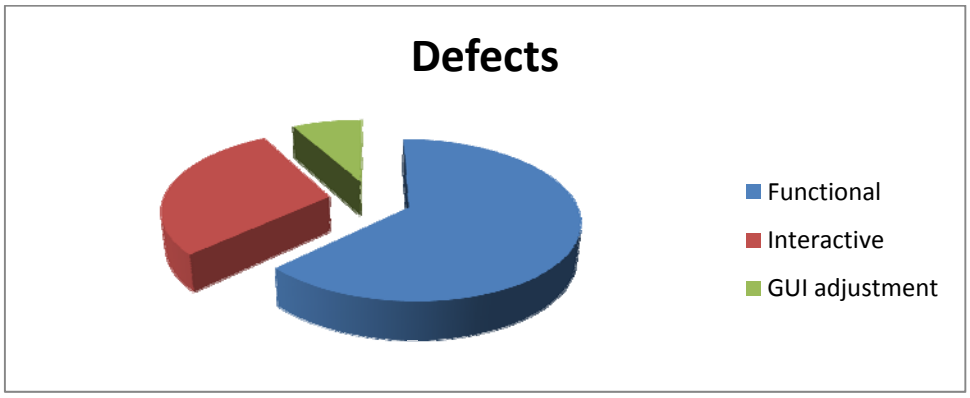


Figure 4.1 Distribution of GUI detectable defects

4.4 Classification directed test case generation

GUI detectable defects are those that occur during the use of GUI-based AUTs and appear in observable GUI behaviours such as error messages, wrong windows or dialogs, and wrong object property values. These uses of an AUT include starting, configuring, performing task on, and closing the AUT. All the interactions taken by users through the GUI, such as starting an AUT by clicking on a system menu item or double clicking on an icon, opening a window by clicking on a button or a menu item, inputting information by typing on a focused text box, resizing a window by pressing the left mouse button and dragging the mouse, will eventually get down to events on certain GUI objects. In contrast to traditional software testing methods such as unit testing, GUI defects are usually found by performing actions on the GUI and observing

the behaviours and manifests of the GUI. By focusing on a certain class of defects, as the related objects, properties and events are confined within certain groups, the number of all possible test cases can be exponentially reduced without losing the ability to detect defects.

4.3.1 Functional defects directed test cases generation

Functional defects principally reside in codes which can be invoked from functional objects and functional responsive objects. To generate functional defects directed test cases, a coverage criterion which covers all functional objects and functional responsive objects is described in Definition 4.1.

Definition 4.1: A test suite $\check{T}$ satisfies the ***functional object coverage criterion*** if and only if $\forall\, o \in O_f, \exists\, \acute{T} \in \check{T} \wedge\ \exists\, t \in \acute{T}.\tau \wedge comp(t.e) = o$,

where O_f is the set of all functional and functional responsive objects in the GUIs, and *comp(e)* denotes the component which contains the event *e*.

Figure 4.2 shows the algorithm for generating functional defects directed test cases. Functional object coverage criterion is parameter of Criterion.

Algorithm FunctionDefectsTestCaseGenerating

1) FunctionDefectsTestCaseGenerating (*M* :GUITAM, *C*:Fun_Obj_Criterion, $\check{T}$:TestSuite)
2) {
3) $\check{T}=\emptyset$;

```
4)      For each t ∈ M.T ∧ from(t) = s0
5)      {
6)          GenerateFunctionalTestCase(M, C, Ť,w, t, 0);
7)      }
8)  }
9)  GenerateFunctionalTestCase(M: GUITAM, C: Fun_Obj_Criterion, Ť :TestSuite, w:
    walk, t :transition, len:int)
10) {
11)     if( ∀τ ∈ w ∧ τ ≠ t) w = w ∪ {t};
12)     else return;
13)     If(∃t ∈ w ∧comp(t.e) ∈ Of ∧∀ω ∈ Ť∧ω ≠ w)
14)     {
15)         Testcase Ť = createtestcase(w);
16)         Ť = Ť ∪ {Ť} ;
17)     }
18)     if( meetcriterion(Ť, C)) return;
19)     for each t ∈ M.T ∧ from(t) = to(t)
20)         GenerateFunctionalTestCase (M, C, Ť, w, t, len+1);
21)     w=w-{t};
22) }
```

Figure 4.2 Algorithm for generating functional defects directed test cases

The procedure GenerateFunctionalTestCase in Figure 4.2 is a recursive function. It depth-firstly traverses all the routes in a GUITAM model and collects all the routes that contain at least one event whose related object is a functional or a functional responsive object. Once the test suite covers all functional and functional responsive objects, the procedure exits and the output parameter $\check{T}$ contains the resulting test suite which satisfies the functional object coverage criterion.

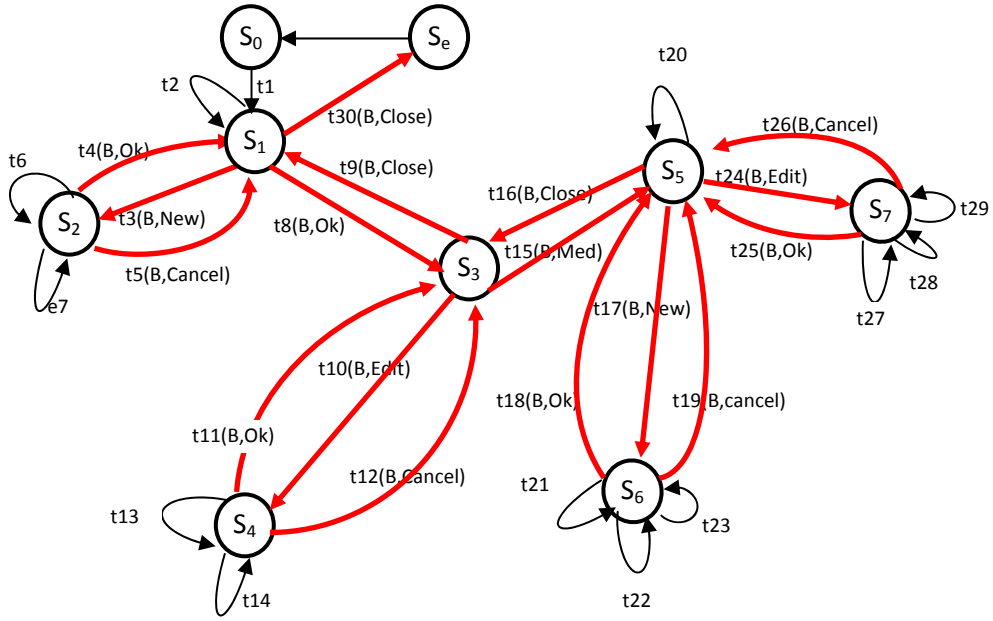


Figure 4.3 Example of Functional Object Coverage

To illustrate the functional object coverage criterion and functional defects directed test case generation, we have used the Simple Clinic Software GUITAM to explain. Figure 4.3 is the graphic model of the GUITAM of Simple Clinic Software. In Figure 4.3, the thick red transitions are related to functional objects. The test suite generated by the FunctionDefectsTestCaseGenerating algorithm in Figure 4.2 needs to cover all these transitions.

4.3.2 Interactive defects directed test case generation

Interactive defects are mainly related to interactive objects and composite interactive objects. To generate interactive defects directed test cases, a coverage criterion which

covers all interactive objects and composite interactive objects is described in Definition 4.2.

Definition 4.2: A test suite $\check{T}$ satisfies the ***interactive object coverage criterion*** if and only if $\forall\, o \in O_i, \exists\, \acute{T} \in \check{T} \wedge\ \exists\, t \in \acute{T}.\tau \wedge comp(t.e) = o$,

where O_i is the set of all interactive and composite interactive objects in the GUIs, and *comp(e)* denotes the component which contains the event *e*.

In Figure 4.4, the doted blue transitions are related to interactive objects. Figure 4.5 shows the algorithm for generating interactive defects directed test cases. The test case suite generated needs to cover all these transitions.

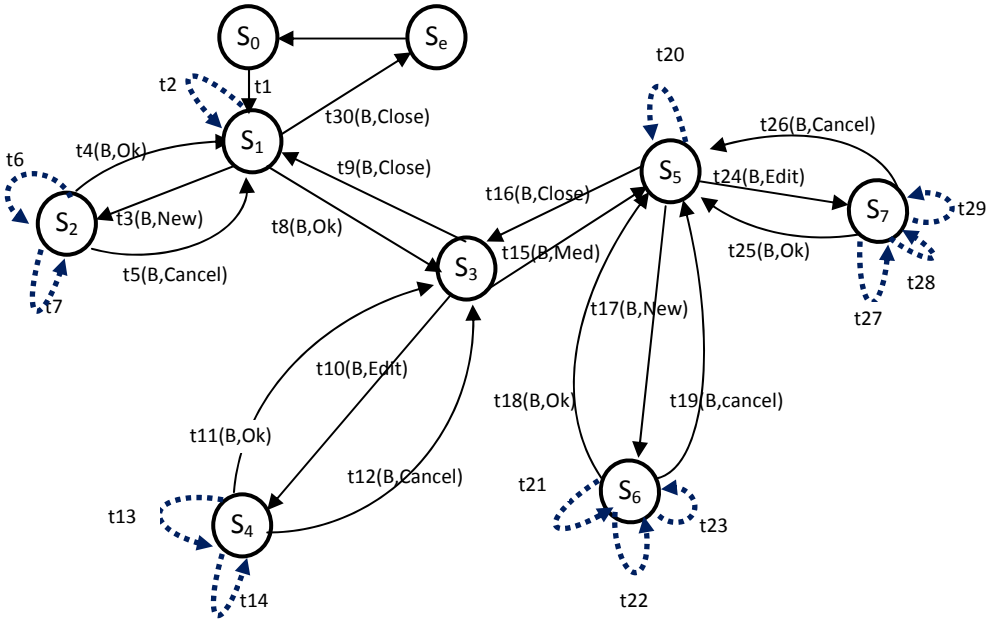


Figure 4.4 Example of Interactive Object Coverage

Procedure GenerateInteractiveTestCase in Figure 4.5 is a recursive function. It also depth-firstly traverses all the routes in a GUITAM model and collects all routes that contain at least one event whose related object is an interactive or a composite interactive object. Once the test suite covers all interactive and composite interactive objects, the procedure exits and the output parameter $\check{T}$ contains the resulting test suite which satisfies the interactive object coverage criterion.

Algorithm InteractiveDefectsTestCaseGenerating

1) InteractiveDefectsTestCaseGenerating (*M* :GUITAM, *C*:Fun_Obj_Criterion, $\check{T}$:TestSuite)
2) {
3) $\check{T}=\emptyset$;
4) For each $t \in M.T \wedge from(t) = s_0$
5) {
6) GenerateFunctionalTestCase(*M, C,* $\check{T}$*,w, t, 0*);
7) }
8) }
9) GenerateInteractiveTestCase(*M* :GUITAM, *C*:Interactive_Obj_Criterion, $\check{T}$:TestSuite, *w:* walk, *t*:transition, *len:* int)
10) {
11) if($\forall \tau \in w \wedge \tau \neq t$) $w = w \cup \{t\}$;
12) else return;
13) *if(*$\exists t \in w \wedge comp(t.e) \in O_i \wedge \forall \omega \in \check{T} \wedge \omega \neq w$*)*
14) {
15) Testcase $\acute{T}$ = createtestcase(*w*);
16) $\check{T} = \check{T} \cup \{\acute{T}\}$;
17) }
18) if(meetcriterion($\check{T}$, *C*)) return;
19) for each $t \in M.T \wedge from(t) = to(t)$
20) GenerateInteractiveTestCase (*M, C,* $\check{T}$*, w, t, len+1*);
21) *w=w-{t};*
22) }

Figure 4.5 Algorithm of generating interactive defects directed test cases

4.3.3 GUI adjustment defect directed test case generation

GUI adjustment defects are related mainly to GUI adjustment objects. To generate GUI adjustment defects directed test cases, a coverage criterion which covers all GUI adjustment objects is described in Definition 4.3.

Definition 4.3: A test suite $\check{T}$ satisfies the ***GUI adjustment object coverage criterion*** if and only if $\forall\, o \in O_a, \exists\, \acute{T} \in \check{T} \wedge\ \exists\, t \in \acute{T}.\tau \wedge comp(t.e) = o$,

where O_a is the set of all GUI adjustment objects in the GUIs, and *comp(e)* denotes the component which contains the event *e*.

Figure 4.6 shows the algorithm for generating GUI adjustment defects directed test cases. GUI adjustment object coverage criterion is the criterion parameter.

The algorithms in Figures 4.2, 4.5 and 4.6 are similar. The main difference between them is the criterion and the subject object collection. Each algorithm selects test cases which cover the given criterion. The generated suite size does not necessarily have the smallest number of test cases. Actually there are many shorter test cases which are subsets of longer test cases. Removing these can greatly reduce the suite size without violating the coverage criterion. Figure 4.6 shows the algorithm for refining the test suite.

The algorithm TestSuiteRefining (in Figure 4.7) is straight-forward. Line 3 sorts the test cases in ascending mode by transition sequence. Lines 5-12 check each test case and its following test case. If a test case is a subset of another test case, it must be a subset of

the test case which is just next to it. If a test case is a subset of another test case, this test case shall be removed from the suite.

Algorithm *GUIAdjustment*DefectsTestCaseGenerating

1) GUIAdjustment DefectsTestCaseGenerating (*M* :GUITAM, *C*:Fun_Obj_Criterion, $\check{T}$:TestSuite)
2) {
3) $\check{T}=\emptyset$;
4) For each $t \in M.T \wedge from(t) = s_0$
5) GenerateGUIAdjustmentTestCase(*M, C,* $\check{T}$*,w, t, 0*);
6) }
7) GenerateGUIAdjustmentTestCase (*M* :GUITAM, *C* :GUIAdjustment_Obj_Criterion, $\check{T}$:TestSuite*, w:* walk, *t*:transition, *len:* int)
8) {
9) if($\forall \tau \in w \wedge \tau \neq t$) $w = w \cup \{t\}$;
10) else return;
11) if($\exists t \in w \wedge comp(t.e) \in O_a \wedge \forall \omega \in \check{T} \wedge \omega \neq w$)
12) {
13) Testcase $\acute{T}$ = createtestcase(*w*);
14) $\check{T} = \check{T} \cup \{\acute{T}\}$;
15) }
16) if(meetcriterion($\check{T}$, *C*)) return;
17) for each $t \in M.T \wedge from(t) = to(t)$
18) GenerateGUIAdjustmentTestCase (*M, C,* $\check{T}$*, w, t, len+1*);
19) *w=w-{t};*
20) }

Figure 4.6 Algorithm for generating GUI adjustment defects directed test cases

Algorithm TestSuiteRefining

1) TestSuiteRefining ($\check{T}$:TestSuit)

```
2)  {
3)      SortByTransitionSequenceAscending(Ť)
4)      i=0;
5)      while( i< Ť.size-1)
6)      {
7)          Ť1= Ť[i];
8)          Ť2= Ť[i+1];
9)          if(Ť1 is subset of Ť2)
10)             Remove(Ť ,Ť1);
11)         else i=i+1;
12)     }
13) }
```

Figure 4.7 Algorithm for refining test suite

4.3.4 Functional-interactive defects directed test case generation

Sole defect classification directed test cases suite size is much smaller and also useful for finding related common defects, especially when the functions related to the functional objects are independent. For example, an "open file" menu item opens an open file dialog; a 'C' button clears the contents of the textbox in a calculator. If there are some malfunctions under these buttons, a mouse click on the button will find the faults. Unfortunately, this is not always the case. Many functional objects function differently when the contents of interactive objects are different. For example, when a 'File name' text box is empty and you click 'Open' button, it behaves differently from when the 'File name' text box is filled with a file name. To find these kinds of defects, compound coverage of both interactive and functional objects is needed to generate test

cases which take account of different valued interactive objects. It is possible to test all value combinations for bi-valued objects whereas it is not possible to test all value for other objects such as text boxes, spinners etc. For input fields with large or unlimited domain sizes, the specifications of the AUT must be used for boundary information. With the boundary information for each field, the test cases can be generated within all these combinations. The number of combinations will increase exponentially as the number of interactive objects increases. Suppose that there are *m* bi-valued objects, *n* large-domain objects and *k* functional objects, and for each large-domain object three values are used: lower boundary, upper boundary and mid-value, then the total number of possible test cases is$2^m \times 3^n \times k$. Obviously, it is such a prodigious number that testing all the possibilities is impractical.

To generate functional-interactive test cases, a functional-interactive coverage criterion is given in Definition 4.4. Since it is impractical to test all the possible combinations, this coverage criterion requires covering all possible states for each object at least once and each test case's last transition is an event of one of the functional object.

Definition 4.4: A test suite $\check{T}$ satisfies the ***functional-interactive coverage criterion*** if and only if $\forall\, o \in O_i, \forall v \in V_o, \forall o' \in O_f, \exists\, \acute{I} \in \check{T} \wedge\ \exists\, t \in \acute{I}.\tau \wedge comp(t.e) = o \wedge e = setvalue(v) \wedge \exists\, \acute{I}' \in \check{T} \wedge\ \exists\, t' \in \acute{I}'.\tau \wedge comp(t'.e) = o'$,

where O_i is the set of all interactive or composite interactive objects in the GUIs, and *comp(e)* denotes the component which contains the event *e,* O_f is the set of all

functional objects in the GUIs, V_o is the set of all possible statuses of object *o*, and $\acute{T}''$ is the last transition of $\check{T}$.

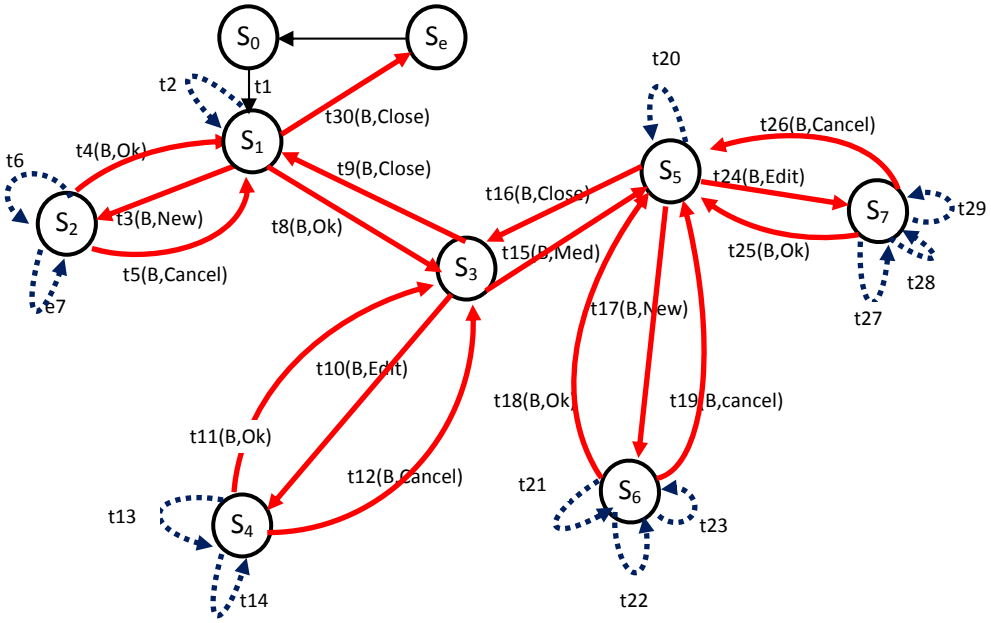


Figure 4.8 Example of Functional-Interactive Object Coverage

In Figure 4.8, the solid red lines are the functional object related transitions and the blue dotted lines are the interactive object related transitions. Functional-Interactive object coverage needs the test suite to cover all the solid and dotted transitions. Each test case needs to include both functional transitions and interactive transitions. Figure 4.9 shows the algorithm for generating functional-interactive compound test cases.

Algorithm Functional-InteractiveTestCaseGenerating

1) Functional-InteractiveTestCaseGenerating (*M* :GUITAM, *C* :Functional_interactive_Criterion, $\check{T}$:TestSuite)
2) {

3) $\check{T}$=Ø;
4) For each $t \in M.T \wedge from(t) = s_0$
5) Generate Functional-Interactive TestCase(*M, C, Ť,w, t, 0*);
6) }
7) Generate Functional-InteractiveTestCase (*M* :GUITAM, *C* :Functional_interactive_Criterion, $\check{T}$:TestSuite, *w* :walk, *t* :transition, *len:* int)
8) {
9) if($\forall\tau \in w \wedge \tau \neq t$) $w = w \cup \{t\}$;
10) else return;
11) if($\exists t \in w \wedge o = comp(t.e) \in O_i \wedge \exists v \in V_o \wedge t.e = setvalue(v) \wedge \forall\omega \in \check{T} \wedge \omega \neq w \wedge comp(t.e) \in O_f$)
12) {
13) Testcase $\acute{T}$ = createtestcase(*w*);
14) $\check{T} = \check{T} \cup \{\acute{T}\}$;
15) }
16) else return;
17) if(meetcriterion($\check{T}$, *C*)) return;
18) for each $t' \in M.T \wedge from(t') = to(t)$
19) Generate Functional-InteractiveTestCase (*M, C, Ť, w, t', len+1*);
20) *w=w-{t};*
21) }

Figure 4.9 Algorithm for generating functional-interactive compound test cases

The algorithm Functional-InteractiveTestCaseGenerating in Figure 4.9 traverses the graph in a given GUITAM model and selects the routes which contain events that set the interactive objects' main property values. For bi-valued objects such as check boxes, it can be either setvalue(true) or setvalue(false). For a large-domain object such as a spinner, it can be either setvalue(minimum), setvalue(maximum) or

setvalue((maximum+minimum)/2). Functional-interactive test cases see the value set of the interactive and composite interactive objects as the parameters of the functions which can be triggered by functional objects. Each test case generated by this algorithm ends with a functional event.

4.4 Experiment

This experiment was based on the experiment in Chapter 3. The same four subject applications and the generated GUITAM models were used. For each subject application, functional defects directed test cases, interactive defects directed test cases and functional-interactive defects directed test cases were generated. Test oracle information was generated as described in Chapter 3. Because the four subject applications are different kinds of applications, the distributions of each kind of object are different. In Calculator, all inputs are made by buttons, and functional objects account for the majority of the objects. In ScreenDrawer, because functions are mainly brought out by buttons and menu items, functional objects also account for the majority of the objects. In the other two applications, functional objects still account for significant proportions of the objects, but there are comparatively less of them. Table 4.7 shows the distributions of different kinds of objects in the subject applications.

Table 4.7 Distribution of different kinds of objects in subject applications

Subject application	Object number

	Functional		Interactive		Component Interactive		Others	
Calculator	54	75%	4	5%	0	0%	15	20%
EasyWriter	54	36%	22	14%	8	5%	68	45%
EnglishStudy	45	21%	23	11%	13	7%	129	61%
ScreenDrawer	84	60%	8	6%	4	3%	44	31%

Defect classification directed test cases for certain applications were generated according to a given type of defects and coverage criterion. Because the number of any given type of object is much smaller than the total number of objects, the combinations of events related to the given type of objects were exponentially decreased and therefore the required number of defect classification directed test cases was much smaller than the general generated test cases in Chapter 3. Table 4.8 shows the number of different kinds of test cases generated for each of four subject applications respectively.

Table 4.8 Number of defect classification directed test cases

Subject application	Test case number			
	Functional	Interactive	Functional-Interactive	Total
Calculator	265	26	662	953
EasyWriter	224	165	326	715
EnglishStudy	162	172	586	920
ScreenDrawer	326	56	462	844

Test oracle information was also generated automatically from the base versions of the subject applications. By executing the selected test cases on the base version of each application, the state information was retrieved and saved after each event in each test

case was performed. Table 4.9 shows the oracle information for each subject application.

Table 4.9 Oracle information for each subject application

Subject application	Test cases		Oracles size (MB)
Calculator	Functional	265	14.5
	Interactive	26	1.8
	Functional-Interactive	662	62.3
	Total	953	78.6
EasyWriter	Functional	224	3.6
	Interactive	165	3.4
	Functional-Interactive	326	6.9
	Total	715	13.9
EnglishStudy	Functional	162	4.6
	Interactive	172	4.9
	Functional-Interactive	586	20.3
	Total	920	29.8
ScreenDrawer	Functional	326	8.2
	Interactive	56	1.6
	Functional-Interactive	462	12.5
	Total	844	22.3

Because the total number of test cases generated by the defect classification directed method was much smaller than that in Chapter 3, the oracle size of the test suite for each application was consequently much smaller. Figure 4.10 shows the comparison

of oracle sizes used between the Length-n method used in Chapter 3 and the Classification Directed method used in this chapter.

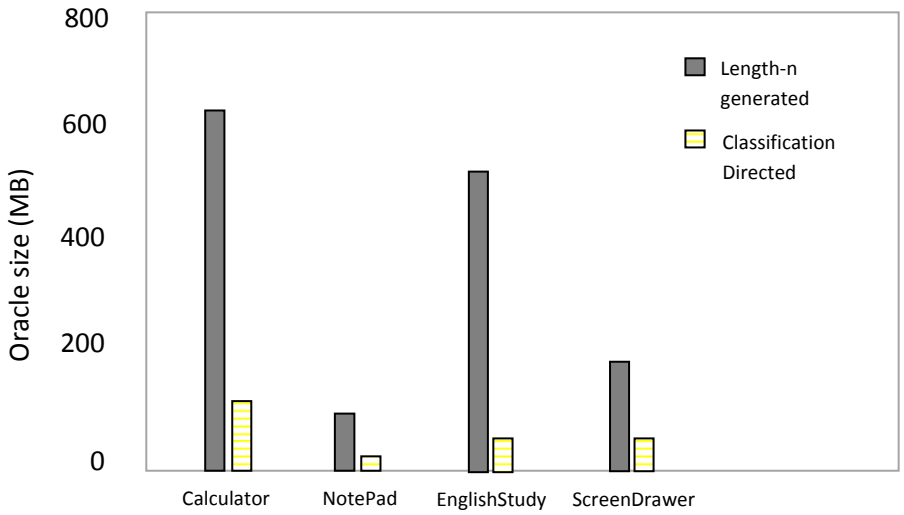


Figure 4.10 Comparison of oracle sizes generated by different methods

Different kinds of test cases were executed separately on the subject applications to check the effectiveness of the detecting faults. Figure 4.11 shows the numbers of faults detected by the different kinds of test case. The solid bar stands for the total number of faults detected by the three kinds of test case.

From the results shown in Figure 4.11, we can conclude that functional directed test cases can find more faults than interactive-only directed test cases. The joint functional-interactive directed test cases have the best ability to detect faults. From Table 4.8, we know that the number of functional-interactive test cases is larger than

the number of other kinds of test case, which is one of the reasons why it is able to detect more faults.

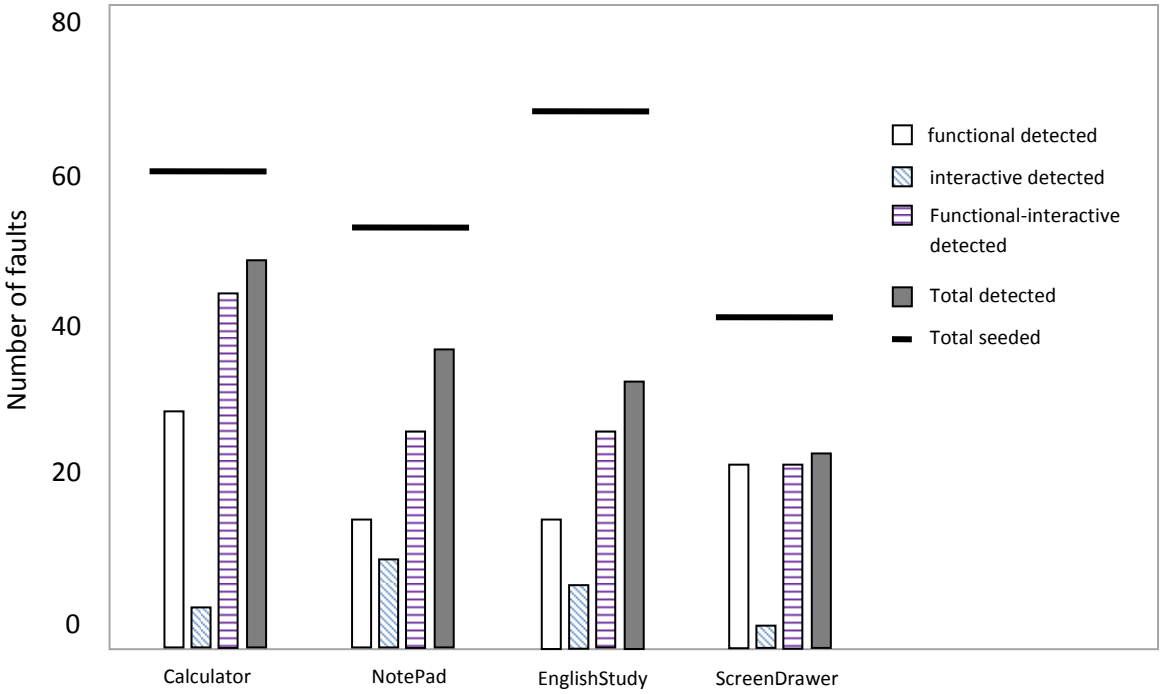


Figure 4.11 Number of faults detected by defect classification directed test cases

In comparison with the results shown in Figure 3.26, even though the ability of a specific type of cases to detect faults varies, the number of total faults detected by all the types of test case is similar to the number of faults detected by the general generated test case. The results show that defect classification directed test cases have a similar ability to detect faults as do general generated test cases, but with much fewer test cases. Figure 4.12 shows the numbers of test cases used for the testing, where the grey bar shows the number of test cases generated with the length-n coverage criterion

in Chapter 3 and the yellow bar shows the number of test cases generated with the defect classification directed method described in this chapter.

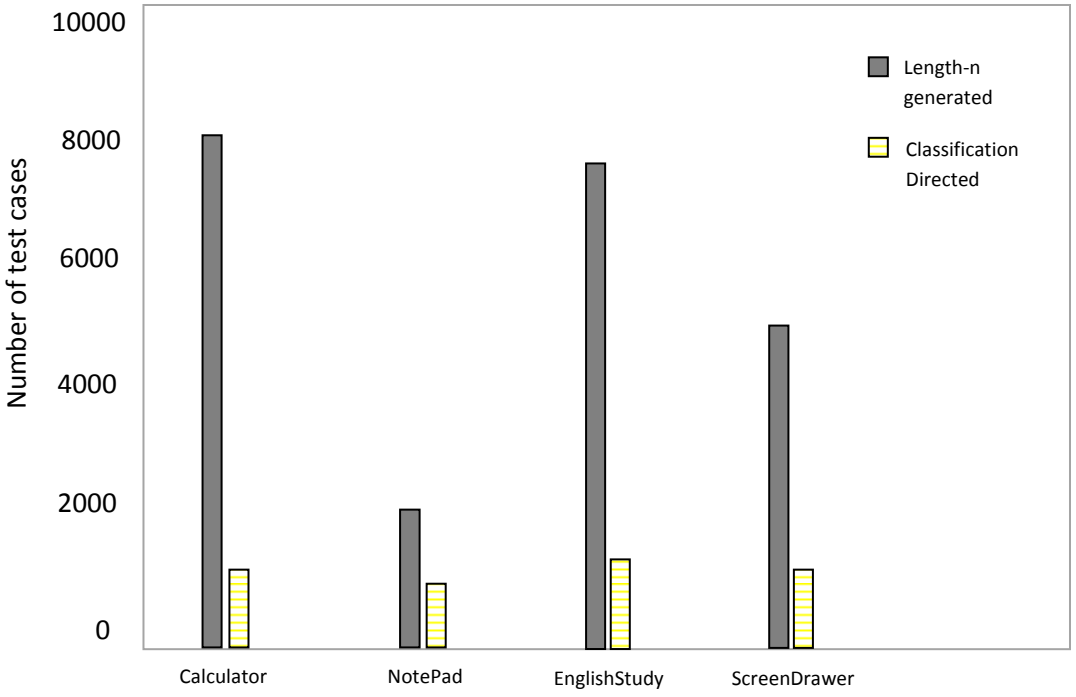


Figure 4.12 Comparison of numbers of test cases generated by different methods

From Figures 4.12 and 4.13, we can see that even the numbers of test cases used in defect classification directed generation method are fewer - 10% of the length-n method used in Chapter 3. However, the numbers of faults found with the smaller number of test cases was almost the same. Apparently, the defect classification directed test cases generating method can generate more efficient test cases.

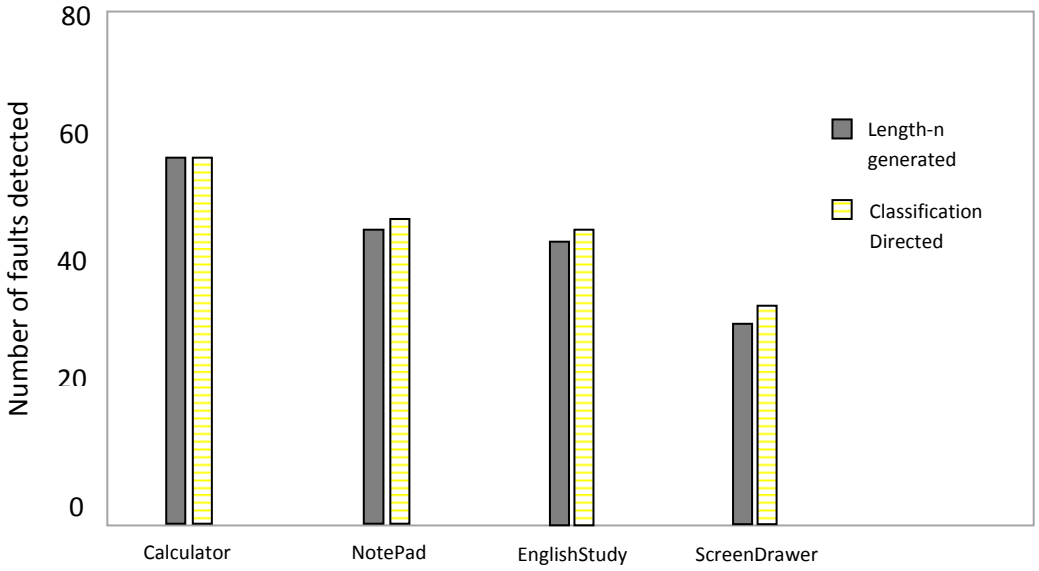


Figure 4.13 Comparison of numbers of faults detected by different methods

4.5 Conclusion

This chapter analyzed the classification of defects. Defect classification is based on the classification of objects. According to the characteristics of object functionality, defects are divided into three main classes: functional defects, interactive defects, and GUI adjustment defects. The distribution of different classes of defects in popular software was also presented. To reduce the number of test cases without losing the quality of detecting defects, defect classification directed test case coverage criteria and test case generation algorithms were presented as well. The test case coverage criteria included functional object coverage criterion, interactive object coverage criterion, GUI adjustment object coverage criterion, and functional-interactive coverage criterion. Corresponding test case generating algorithms included Functional Defects Directed

Test Case Generating, Interactive Defects Directed Test Case Generating, GUI Adjustment Defects Directed Test Case Generating, and Functional-interactive defects directed test case generating. By focusing on certain classes of defects, the corresponding algorithms can generate efficient test cases with a very small test suite size.

An experiment was also carried out for evaluating the effectiveness of defect classification directed test cases. In the experiment, functional directed, interactive directed and functional-interactive directed test cases were generated respectively for all four subject applications. The results showed that, with a much smaller number of test cases, defect classification directed test cases can effectively and efficiently detect faults.

Chapter 5

Long Use Case Closure Envelope Model

In principle, an infinite number of event sequences may be performed on a GUI. As discussed in previous chapters, exhaustive testing on GUIs is impossible. It is very important to generate a manageable number of effective test cases in the light of the resources available. Various approaches may be used to automatically generate test cases for GUIs such as random test case generation and model-based structural test case generation. Due to the prodigious number of possible test cases, conventional methods usually try to avoid test cases explosion by limiting the length of each test case to certain steps, normally three to finish the tests in a practical time. Three steps are normally far from enough to cover a task. Chapter 4 presented defect classification and defect classification directed test cases generating algorithms which greatly reduce the number of test cases without losing the ability to detect GUI faults. However, lack of human knowledge about business logic of applications limits the defect classification directed methods from efficiently detecting defects that reside in the logic of long tasks. Tasks are typical scenarios which are most often used by software users. Errors in these typical tasks may lead to fatal interruption or even the disaster of losing important data. Some research has been done on software testing by using use cases [116, 117, 118, 119, 11], but few of them integrate use cases into GUI testing automation. In this chapter, by making use of the use cases which are either from

AUTs' specifications or from the records of user actions of performing typical tasks, a Long Use Case Closure Envelope Model is proposed in order to generate highly task-oriented test cases.

5.1 Use cases representation

Use cases are used to describe the behaviour of a system. System functionalities are identified and described with a set of use cases during the analysis phase of a project. Actors are used to represent parties outside the system which interact with the system. Actors can be either humans or any other systems such as computers, hardware etc. Actors must be external to the use cases of the system and supply stimulus to the use cases. Use cases capture *who* (actor) does *what* (interaction) with the system, for what *purpose* (goal), without dealing with system internals. A complete set of use cases specifies all the different ways to use the system, and therefore defines all behaviour required of the system, bounding the scope of the system.

Usually use cases are described in use case diagrams with Unified Modelling Language (UML) [122]. UML (1999) provides three relationships that can be used to structure use cases. These are *generalization*, *include* and *extends*. An *include* relationship between two use cases means that the sequence of behaviour described in the included (or sub) use case is included in the sequence of the base (including) use case. Including a use case is thus analogous to the notion of calling a subroutine [121].

The *extends* relationship provides a way of capturing a variant to a use case. Extensions are not true use cases but changes to steps in an existing use case. Typically extensions are used to specify the changes in steps that occur in order to accommodate an assumption that is false [121]. The *extends* relationship includes the condition that must be satisfied if the extension is to take place, and references to the extension points which define the locations in the base (extended) use case where the additions are to be made.

A *generalization* relationship between use cases "implies that the child use case contains all the attributes, sequences of behaviour, and extension points defined in the parent use case, and participates in all relationships of the parent use case." The child use case may define new behaviour sequences, as well as add behaviour into and specialized existing behaviour of the parent [122]. Figure 5.1 is an example of a use case diagram for an online shopping system.

Formally, a Use Case Diagram can be defined as Definition 5.1.

Definition 5.1 (***Use Case Diagram***) A use case diagram $UCD = (n, ACT, UC, \rightarrow\!\triangleright, \text{—}, \text{and } \overset{\ll i \gg}{\dashrightarrow})$ consists of a diagram name n; a finite set ACT of actor's names which can be users and external systems; a finite set UC of use cases; and three relations $\rightarrow\!\triangleright$, —, and $\overset{\ll i \gg}{\dashrightarrow}$, where $\rightarrow\!\triangleright \subseteq (ACT \times ACT) \cup (UC \times UC)$; — $\subseteq ACT \times UC$; and $\overset{\ll i \gg}{\dashrightarrow} \subseteq$

UC × *UC;* as usual we write $p \rightarrow q$, rather than $(p, q) \in \rightarrow$, and analogously for — and <<i>> --> [120].

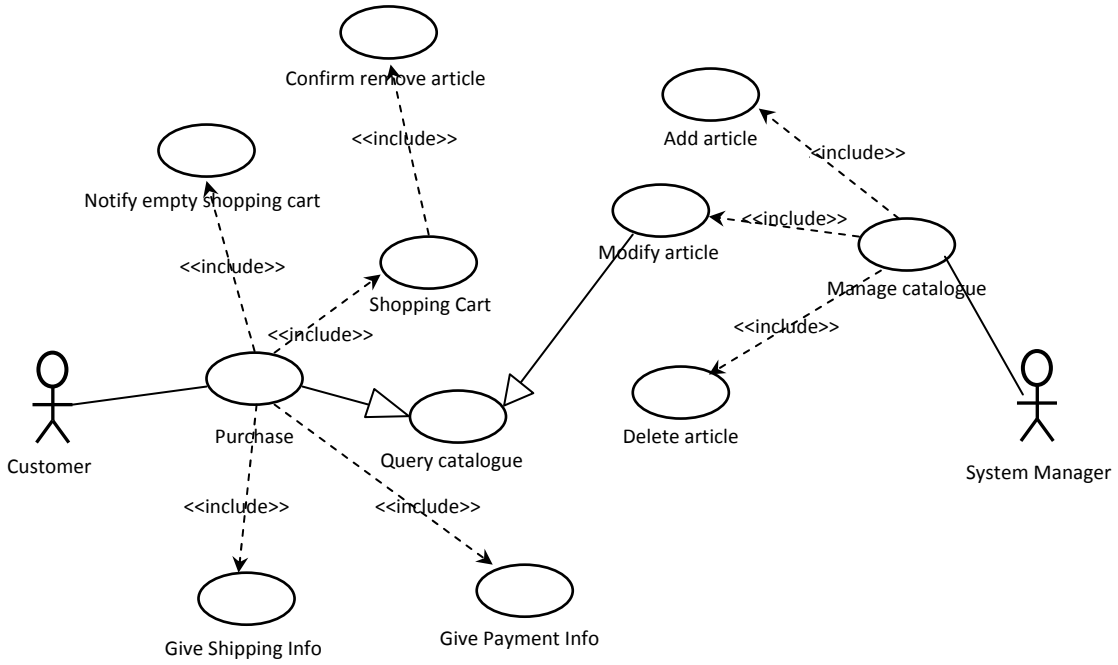


Figure 5.1 Online shopping use case diagrams

Each use case should have a unique name suggesting its purpose. The name should express what happens when the use case is performed. It is convenient to include a reference number to indicate how it relates to other use cases. The name field should also contain the creation and modification history of the use case preceded by the keyword history [121]. Each use case can be detailed with an activity diagram.

Activity diagrams are graphical representations of workflows of stepwise activities and actions with support for choice, iteration and concurrency. This chapter uses activity

diagrams to describe the operational step-by-step workflows of components in a use case. Activity diagrams are constructed from a limited repertoire of shapes, connected with arrows. The most important shape types are [120]:

● : a black circle, representing the start (initial state) of the use case;

◉ : an encircled black circle, representing the end (final state) of the use case.

▭ : rounded rectangles, representing stereotyped states or other activity diagrams. If another activity diagram is represented, this diagram can either represent the behaviour of another use case or simply a way of allowing a hierarchical decomposition of the original activity diagram.

◇ : diamonds, representing conditions. Conditions can represent user choices or business / data logic. User choices are a condition of the user's interaction with a graphical component. Business / data logic is an internal checking condition.

▬ :bars, representing the start (split) or end (join) of concurrent activities;

⟶ :arrows running from the start towards the end and representing the order in which activities happen. Each arrow is also called a transition. A transition can be a stereotype, condition or both together.

When describing a use case with an activity diagram, the states and transitions are mainly considered. In an activity diagram, ●,◉,▭ represent initial state, final state and intermediate state respectively. An intermediate state can be either a stereotyped

state or not. Stereotyped states represent atomic states which can be labelled and directly mapped to a set of GUI components. A non-stereotyped state means that the state is described in another activity diagram which can either represent the behaviour of another use case or simply a way of allowing a hierarchical decomposition of the original activity diagram. Transitions (→) can be labelled by means of stereotypes, conditions or both together. A stereotyped transition can usually be directly connected to an event on the GUI such as a button click. Conditions can represent user choices or internal business / data logic. A user choice condition is a condition of the user's interaction with a GUI component such as selecting a radio button. A business /data logic condition is an internal checking condition in an internal process such as whether a shopping cart is empty. Figure 5.2 is an example activity diagram describing a purchase use case. Definition 5.2 formally defines a use case activity diagram.

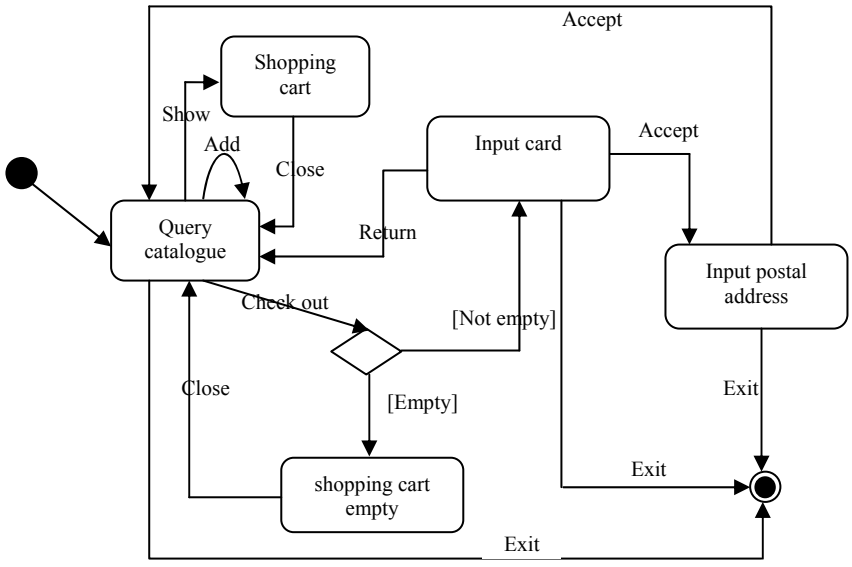


Figure 5.2 Activity diagram of a purchase use case

Definition 5.2 (***Use Case activity diagram***) A use case activity diagram (UCAD) $u = (n, S, SI, IN, OUT, COND, \rightarrow)$ consists of: a use case name n; a finite set S of states which consist of: (1) a finite set UC of use cases activity diagram; (2) a finite set SS of stereotyped states of the form (sn, p) where sn is a state name and $p \in OUT$; (3) three special states SP, the initial, end and branching states; a finite set SI of stereotyped interactions of the form $[C]/(in, i)$ where $C \in COND$, in is an interaction name, and $i \in IN$. The condition $[C]$ is optional; a finite set IN of input stereotypes; a finite set OUT of output stereotypes; a finite set $COND$ of conditions; a transition relation $\rightarrow \subseteq S \times (SI \cup COND) \times S$. As usual we write $A \xrightarrow{\lambda} B$ rather than $(A, \lambda, B) \in \rightarrow$, where λ can be $[C]$ or $[C]/(in, i)$.[120]

A *scenario* is an instance of a use case, and represents a single path through the use case. Thus, one may construct a scenario for the main flow through the use case, and other scenarios for each possible variation of flow through the use case (e.g., triggered by options, error conditions, security breaches, etc.). Scenarios may be depicted using sequence diagrams.

A *Sequence Diagram* shows interactions among a set of objects in temporal order. It depicts the objects by their lifelines and shows the messages they exchange in time sequence. For GUI-based software, the message sequence can be converted into an events sequence on the GUI which represents a task execution procedure. A Sequence Diagram has two dimensions: the vertical dimension represents time, and the

horizontal dimension represents the objects. Messages are shown as horizontal solid arrows from the lifeline of the object sender to the lifeline of the object receiver. A message may be guarded by a condition, annotated by iteration or concurrency information, and/or constrained by an expression. Each message can be labelled by a sequence number representing the nested procedural calling sequence throughout the scenario, and the message signature. A use case sequence diagram is formally defined in Definition 5.3. Figure 5.3 is a sequence diagram of a use case "Submit a cart".

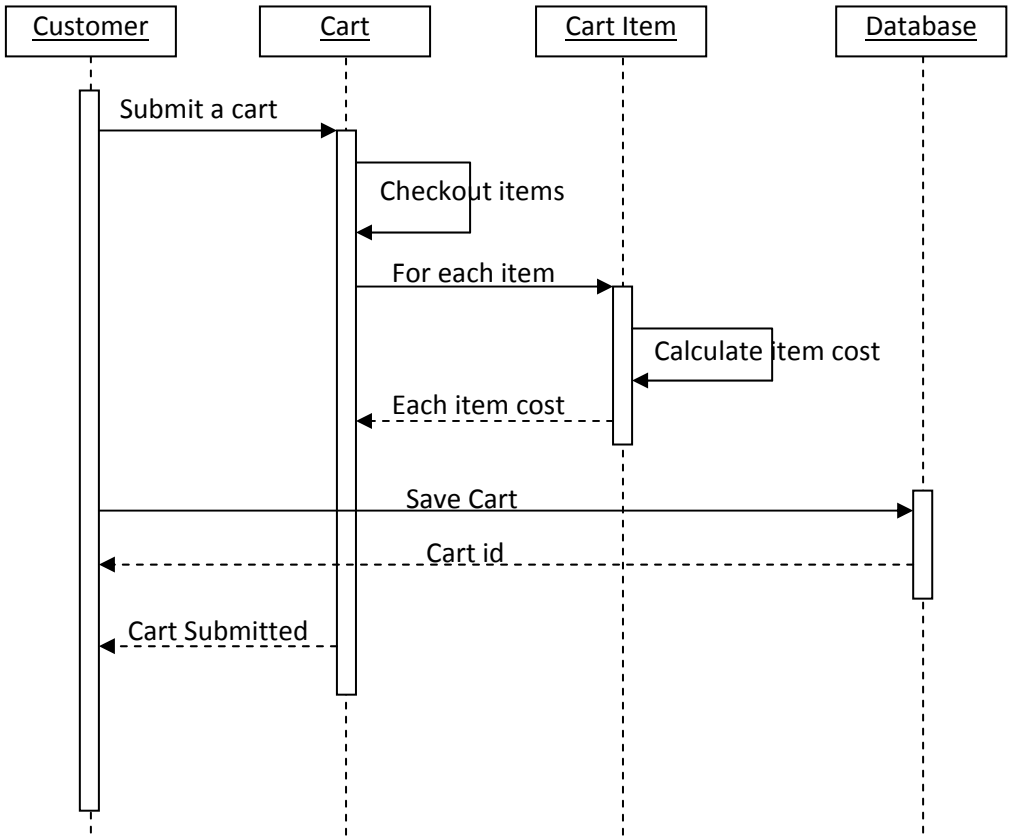


Figure 5.3 "Submit a cart" use case sequence diagram

Definition 5.3 (***Use Case Sequence Diagram***) A use case diagram *UCSD* = (O, M, E) where:

- $O = (o_1, o_2, ..., o_m)$, is a collection of objects. Each object can be also an actor.
- $M = (m_1, m_2, ..., m_k)$, is a collection of messages. Each message is a tuple: $m = (o_i : C_i; o_j : C_j ; action; order)$ where o_i is the source object of the message with class type C_i. o_j is the target object of the message with class type C_j. *action* is a guarded method call. *order* is the number of the message in the corresponding sequence.
- $E = M (s, r)$, is the event set. Event is to send and receive messages. [120]

A sequence diagram describes a series of ordered messages which indicate the operations and data flow between objects. Apparently, a sequence diagram is more for humans rather than for computers to understand. The lack of detailed information limits the capacity for automation. Both object and message are abstract concepts which can be better understood by a human being than by machines. For testing purpose, the series of messages need to be linked to concrete events and parameters, and the objects need to be linked to GUI states and widgets. Each object can be either an actor or a concrete set of GUI objects (widgets). For examples, a customer in Figure 5.3 is an actor. A cart may be a set of GUI objects: a label named 'Cart', a listbox showing the list of items, and a textbox showing the amount of total price. When a non-actor object is linked to GUI objects, it is easy to find the corresponding states which contain the GUI objects. Each message in a sequence diagram must be

generated under certain context, which is called a state. Each message can be embodied with a series of events on a GUI. To add detailed information into a sequence diagram, a new model called Detailed Use Case Sequence Diagram (DUCSD) was created, which can be easily used for automatic task oriented test case generating. DUCSD is formally defined in Definition 5.4.

Definition 5.4 (***Detailed Use Case Sequence Diagram***) A use case diagram *DUCSD = (O, M, E, S)* where:

- $O = (o_1, o_2, ..., o_m)$, is a collection of objects. Each object can be either an actor or not. If an object is not an actor, it can be lined with a set GUI objects. $\forall o \in O$, $o = (t, W)$ where t is the type of an object. The value of t can be either "actor" or "non-actor". W is a set of GUI objects. $W = \emptyset$ if t="actor".
- $M = (m_1, m_2, ..., m_k)$, is a collection of messages. Each message is a tuple: $m = (o_i : C_i; o_j : C_j ; action; order; s, E;)$ where o_i is the source object of the message with class type C_i. o_j is the target object of the message with class type C_j. *action* is a guarded method call. *order* is the number of the message in the corresponding sequence. s is the initial state which describes the initial context of starting the message. E is a set of events which depict the detailed process of the generating and sending the message.
- E is the union of all event sets in messages in M

- $S=(s_1,s_2,...,s_n)$ is all GUI states that are involved in events in E. $\forall s \in S$, s is basically a GUI state defined in GUITAM with an additional Boolean mark: *IsStateLess*. If the *IsStateLess* = *true*, the state is stateless which means no matter how it comes to this state, the results of any other sequences that start from this state won't be affected.

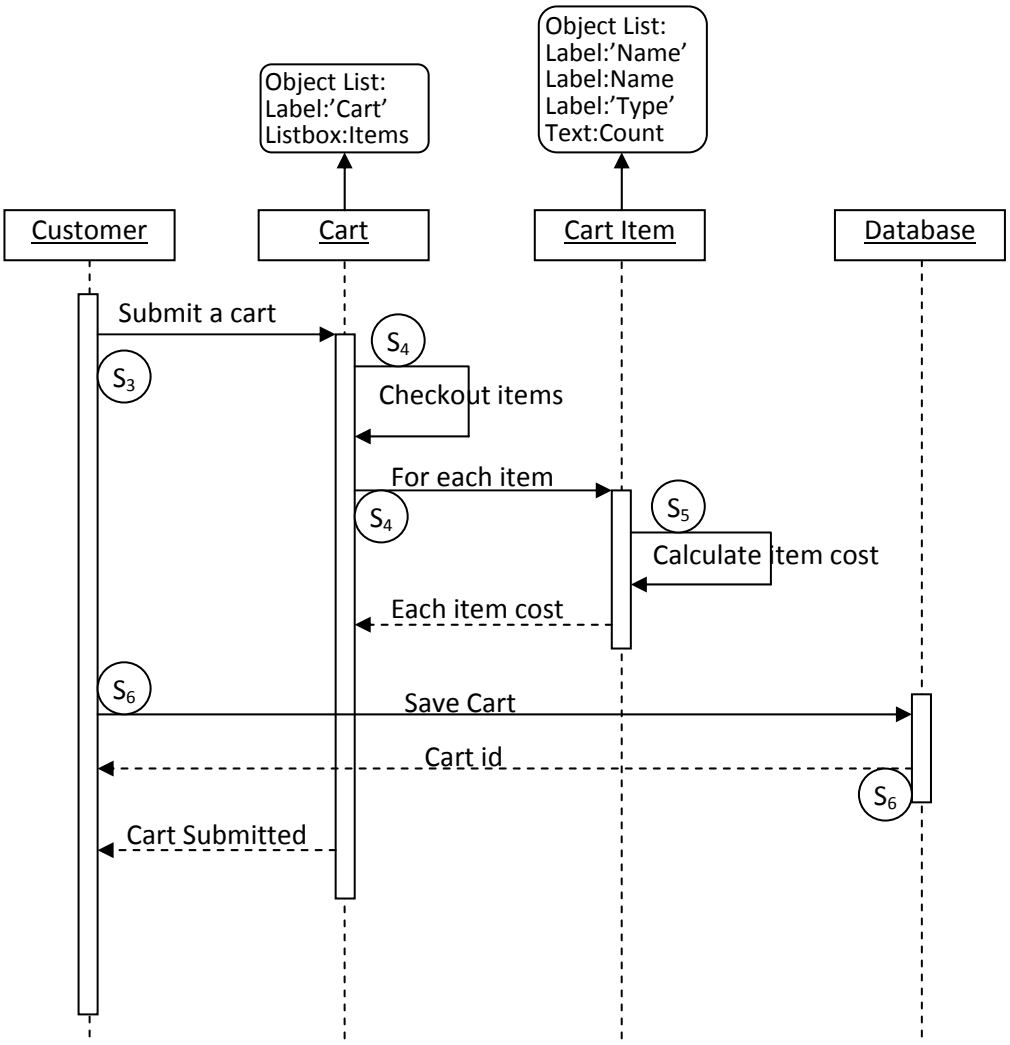


Figure 5.4 "Submit a cart" detailed use case sequence diagram

The detailed information for a DUCSD cannot be retrieved from the specifications of an AUT because conventional UML doesn't support this new model. Thus manual work is needed to put the detailed information into an existing use case. Figure 5.4 shows the detailed use case sequence diagram of use case "Submit a cart". In this diagram, two shapes are added to the existing UML model, which are a round rectangle and circle. A round rectangle is used to list the detailed GUI objects related to an abstract object, e.g., a Cart. A circle is used to mark the real GUI state to a message. A rectangle can be edited with the GUITAM tool by pointing the mouse to certain GUI objects, and adding them to a related abstract object. A circle can be edited by navigating the AUT to certain state and connecting the state to a related abstract message. With the GUITAM edit tool, a detailed use case sequence diagram can be easily generated from an existing use case sequence diagram.

5.2 Backbone of a use case

Use cases cannot be directly used in GUITAM automation because the objects and messages described in a use case or a use case sequence diagram are not explicitly linked to states, transitions and objects in GUITAM created from the run time AUT. Conversion algorithms are needed to translate a use case or a use case sequence diagram into a sub-set of GUITAM. A well defined use case is supposed to use the same terminologies as those used in the real GUI implementation which enables the automatic translation from a use case or a use case sequence diagram into a subset of GUITAM. Because each use case has distinct final states, a new model named Use

Case Backbone (UCBB) was specifically defined to describe use cases as a subset of GUITAM. Use Case Backbone is formally defined in Definition 5.5. The conversion from a Use Case Activity Diagram (UCAD), a Use Case Sequence Diagram (UCSD) and a Detailed Use Case Sequence Diagram (DUCSD) to a UCBB will be discussed in the following paragraphs.

Definition 5.5 A Use Case Backbone (UCBB) M_s is 5-tuple (Σ,S,s_0,T, G), is basically a subset of another GUITAM M where

$M_s.s_0 \in M.S$ is the initial state of the use case,

$M_s.\ \Sigma \subseteq M.\Sigma$ is the set of all inputs (events),

$M_s.S \subseteq M.S$ with each s in $M_s.S$, s has an additional Boolean mark named *IsStateLess*,

$M_s.T \subseteq M.T$ is the set of transitions,

$M_s.G \subseteq M.S$ is the set of goal states of the use case, and

$\forall s \in M_s.S$, there exists at least one path from $M_s.s_0$ to s. G.

In a use case activity diagram, a stereotyped state is usually reflected in a state in GUITAM. This state should include one or more active windows or dialogs which have either the title with the name of the stereotyped state or have labels with the name. With the stereotyped state name and its description, a fuzzy search can be used to get the corresponding state in GUITAM. For example, a stereotyped state called 'Open File' can be interpreted as an 'Open file' dialog which is supposed to be one of the

states in GUITAM. If this mapping fails, manual work must be involved to select the corresponding state. Sometimes, one name may be found in more than one state in GUITAM. In this circumstance the related transitions and conditions in the use case can be used to differentiate the specified state from others. We used 'SearchCorrespondingState' to denote the process of looking for the corresponding state in GUITAM of a given name in a use case state. Likewise, we used 'SearchCorrespondingTransition' to denote the process of looking for the corresponding transition in GUITAM of a given message in a use case. Figure 5.5 shows the algorithm ConvertUseCaseActivityDiagramToUCBB.

Algorithm ConvertUseCaseActivityDiagramToUCBB initializes the subset M_s (line 3), and then looks for the use case's initial state's corresponding state in M. This state is set as the initial state s_0 in M_s (lines 4-5). Line 6 starts the recursive ConvertActivityDiagramToUCBB procedure. Lines 7-8 adds the goal state in the use case to $M_s.G$.

The ConvertActivityDiagramToUCBB procedure depth-firstly visits each node through the transitions. Each transition will be visited once and only once (lines 11 – 25). Each use case transition's corresponding GUITAM transition will be found and added to $M_s.T$. Each node can be either a stereotyped state or not. If it is a stereotyped state, the corresponding state in the given GUITAM will be found and added to $M_s.S$. (lines 17-19). If it is a use case, a recursive call to ConvertUseCaseActivityDiagramToUCBB

will be invoked to generate a sub UCBB M_s' for this use case and then the generated sub UCBB M_s' will be merged into M_s (lines 20 – 23).

Algorithm ConvertUseCaseActivityDiagramToUCBB

```
1.  ConvertUseCaseActivityDiagramToUCBB ( u: UseCase, M: GUITAM, Ms: UCBB )
2.  {
3.      Ms. ∑= Ø; Ms.S = Ø; Ms.T = Ø; visited= Ø;
4.      su=GetInitialState(u);
5.      s= SearchCorrespondingState(M, su); Ms.s0= s; Ms.S = Ms.S ∪ {s}
6.      ConvertUseCaseToGUITAMSub(u, M, Ms, su);
7.      g=GetUseCaseGoalState(u);
8.      Ms.G = Ms.G ∪ {SearchCorrespondingState(g)};
9.  }
10. ConvertActivityDiagramToUCBB(u: UseCase, M: GUITAM, Ms: UCBB, s:
    UseCaseState , visited:UseCaseTransition)
11. {
12.     for each t ∈u.SI from(t)=s  t ∉ visited
13.     {
14.         τ = SearchCorrespondingTransition(t, M);
15.         Ms.T=Ms.T ∪ { τ}
16.         s'=to(t);  visited = visited ∪ {t}
17.         if(s' is stereotyped) {
18.             s= SearchCorrespondingState(M, su); Ms.S = Ms.S ∪ {s}
19.         }
20.         else{ //s is another use case
21.             ConvertUseCaseActivityDiagramUCBB ( s', M,  Ms');
22.             Ms = Ms ∪ Ms'
23.         }
24.         ConvertActivityDiagramToUCBB (u,M,Ms,s',visited);
25.     }
26. }
```

Figure 5.5 Algorithm Convert Use Case to UCBB

Use case activity diagrams detail the logic of a use case and therefore completely describe the flow of the use case. Fully detailed use cases are often not available due to

the uncertainty of user requirements. Many designers use sequence diagram to describe the use cases instead. From a sequence diagram, we can also create a UCBB which is very practical and efficient for producing test cases. The objects in a sequence diagram are not so straightforward to be mappable to a state in GUITAM. The core information which is important for the conversion is the messages. Because each message is encoded with the name, objects involved and function description, this information can be used for looking for the corresponding transitions in a given GUITAM. We also use 'SearchCorrespondingTransition' to search a corresponding transition in the GUITAM by the information in a message. Figure 5.6 shows the algorithm for converting a use case sequence diagram to a subset of a GUITAM.

Algorithm ConvertUCSDToUCBB is straightforward. It uses the first message to find the transition in a GUITAM and uses the source state of the transition as the initial state s_0. It also adds transition to $M_s.T$ and both the source and destination states of the transition to $M_s.S$. For each of the rest of the messages, it searches for the corresponding transition in the GUITAM and adds the transition to $M_s.T$ and both the source and destination states to $M_s.S$. Line 41 adds the last state in the sequence diagram to the goal state set $M_s.G$.

Algorithm ConvertUCSDToUCBB

1. ConvertUCSDToUCBB (*d*: SequenceDiagram, *M:* GUITAM, M_s: UCBB)
2. {
3. $M_s.\Sigma = \emptyset$; $M_s.S = \emptyset$; $M_s.T = \emptyset$;
4. SortMessagesByMessageOrderAscending(*d*);
5. *msg=d.M[0];*

```
6.      τ = SearchCorrespondingTransition(msg, M);
7.      Ms.T= Ms.T ∪ { τ };
8.      ssrc= from(τ ); sdst=to(τ ); Ms.s0= ssrc; Ms.S = Ms.S ∪ {ssrc,sdst};
9.      for i=1 to d.M.length-1
10.      {
11.          msg=d.M[i];
12.         τ = SearchCorrespondingTransition(msg, M);
13.         Ms.T= Ms.T ∪ { τ };
14.         ssrc= from(τ ); sdst=to(τ ); Ms.S = Ms.S ∪ {ssrc,sdst};
15.         if(i=d.M.length-1) Ms.G= Ms.G ∪ { sdst };
16.      }
17. }
```

Figure 5.6 Algorithm of converting use case sequence diagram to UCBB

Because of the lack of detailed information in both raw activity diagrams and raw sequence diagrams, it is in practice very hard to convert them automatically into UCBB without any human effort. The new detailed use case sequence diagram DUCSD has detailed information which helps automate the process of the conversion to an UCBB. Figure 5.7 shows the algorithm for converting a DUCSD to a UCBB.

Algorithm ConvertDUCSDToUCBB

```
1.  ConvertDUCSDToUCBB ( d: DUCSD, M: GUITAM, Ms: UCBB )
2.  {
3.      Ms. ∑= Ø; Ms.S = Ø; Ms.T = Ø;
4.      SortMessagesByMessageOrderAscending(d);
5.      for i=0 to d.M.length-1
6.      {
7.          msg=d.M[i];
8.          for j=0 to msg.E.length-1
9.          {
10.             τ =SearchTransition(msg.E[j]);
11.             if(i=0 and j=0)  Ms.s0= from(τ );
12.             Ms.T= Ms.T ∪ { τ };
13.             ssrc= from(τ ); sdst=to(τ );
```

```
14.              tssrc=from(msg.E[j]); tsdst=to(msg.E[j]);
15.              ssrc.IsStateLess=tssrc.IsStateLess; sdst.IsStateLess=tsdst.IsStateLess
16.              Ms.S = Ms.S ∪ {ssrc,sdst};
17.              if(i=d.M.length-1 and j=msg.E.length-1) Ms.G= Ms.G ∪ { sdst };
18.          }
19.      }
20. }
```

Figure 5.7 Algorithm for converting DUCSD diagram to UCBB

Tha algorithm ConvertDUCSDToUCBB in Figure 5.7 makes full use of the detailed information in a DUCSD and generates a corresponding UBCC. Line 3 initializes the UCBB M_s, and then sorts the messages in *d.M* (line 4). The algorithm traverses all the events in each message and looks for all corresponding transitions and states in the given GUITAM *M*. The first event of the first message is used to find the first transition of the use case sequence diagram and the initial state is retrieved from the transition (line 11). The value of 'IsStateLess' property of each state recorded in the DUCSD will be copied to the corresponding destination state. (line 15). The last event of the last message will be used to find the last transition and this transition will be used to retrieve the goal state of the UCBB (line 17). The quality of the conversion relies on the detailed information given in the DUCSD. This algorithm will fail to convert if the messages are not encoded with detailed event sequences.

5.3 Encapsulating the UCBB with an envelope

Both activity diagram and sequence diagram describe the expected procedures for terminal users to use the systems. These procedures are supposed to be the processes most often used by users. Defects that reside in these procedures are often fatal to the functionality of the software. But users don't always use the software in exactly the same ways as described in use cases. Very commonly, software users perform a task

proximately to the given procedure. Most defects reside in the by-side routes which are close to the main route of the given procedure. Generating test cases for testing task related defects needs to take into account the by-side routes which are close to the main expected route. The backbone of a use case in a UCBB, which is generated from either an activity diagram, sequence diagram or detailed use case sequence diagram contains only the main route of a given use case which doesn't include the by-side routes. To generate effective use case orientated test cases, the UCBB needs to be extended to include the related by-side routes. A method called '***Closure***' was used to extend the use case to a larger set of events and states. We call this extended UCBB an envelope, encapsulating all possible branches of states and events possibly related to a given task.

States in a UCBB are called ***base nodes***. The event sequence and corresponding states in the UCBB form the ***backbone*** of the use case. To extend a UCBB to a broader set, events and states in the AUT's GUITAM which are ***close*** to the backbone will be added to the UCBB. A ***by-side*** is a path that starts from one of the base nodes and ends with one of the base nodes. Any other nodes in the by-side path don't belong to the base nodes. To distinguish different types of events, we give a weight λ to each type of events. We used ***length*** to measure the by-sides. The length of a by-side $l = \sum \lambda_e$, e belongs to the events that make up the path. We used ***'closure set'*** to denote all the expanded notes and transitions.

Definition 5.6 A ***closure set*** M_c of UCBB M_s is a 3-tuple *(Σ,S,T),* where *Σ, S, T* are composed of all the events, states and transitions that are contained in all *by-sides* of M_s with length less or equal to a given threshold *d* respectively. The length of each *by-side* $l = \sum \lambda_{e_i}, i = 0\ to\ k$, e_i is the i^{th} event from the backbone; λ_{ei} is the weight of e_i. $from(e_0) \in M_s.S \wedge to(e_k) \in M_s.S \wedge \forall i | 0 < i \leq k, from(e_i) \notin M_s.S \wedge \forall i | 0 \leq i < k, to(e_i) \notin Ms.S.$

Figure 5.8 shows a GUITAM for the popular clinic software Medical Director 3 (MD3) [110] and illustrates the backbone (UCBB) and closure set of sending a referral letter. Figure 5.9 shows the typical interfaces of MD3. Figure 5.9 (a) is the first GUI showing the user agreement, (b) is the login GUI, (c) is the GUI for selecting a patient, and (d) is the main GUI for treating a patient. (d) shows the state of 'Current Medication'. Many other main states such as 'Progress', 'Results', 'Letters', and 'Old Scripts' are organized in different tabs which can be easily changed by clicking on the related tab. (e) is the GUI for editing patient details and (f) is a GUI for writing a new referral letter. In Figure 5.8, the red states and transitions represent the UBCC of the use case, which includes opening MD3, selecting a patient, doing medication, creating a new letter, sending the letter, closing the patient and closing MD3.

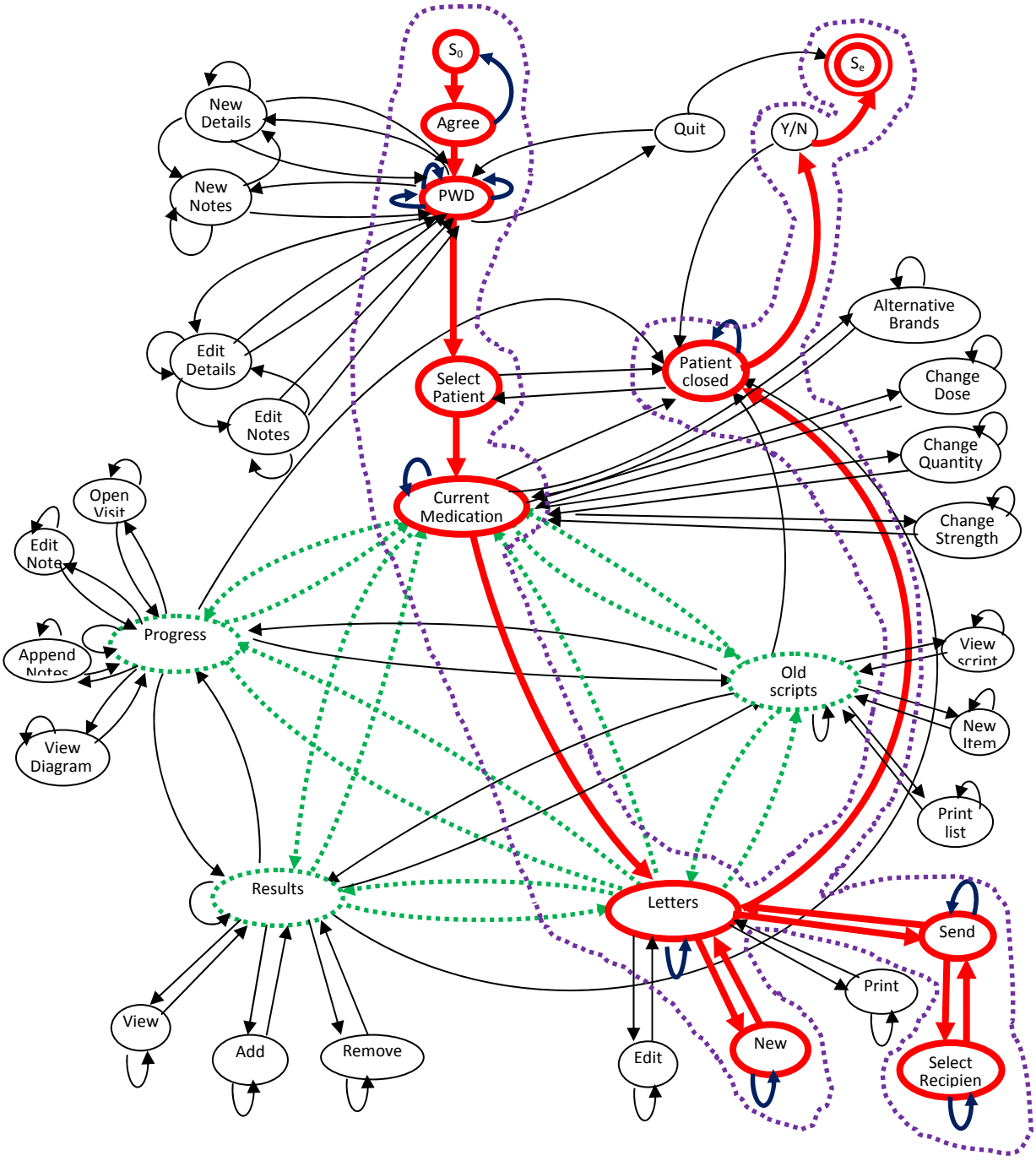
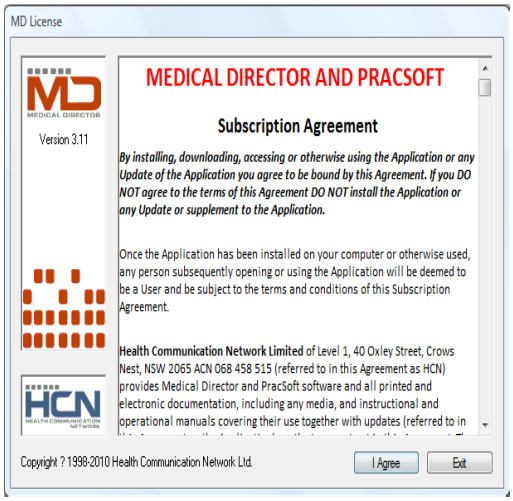
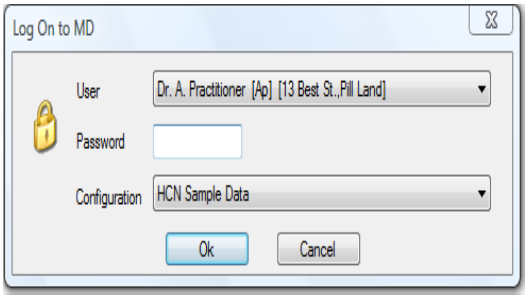


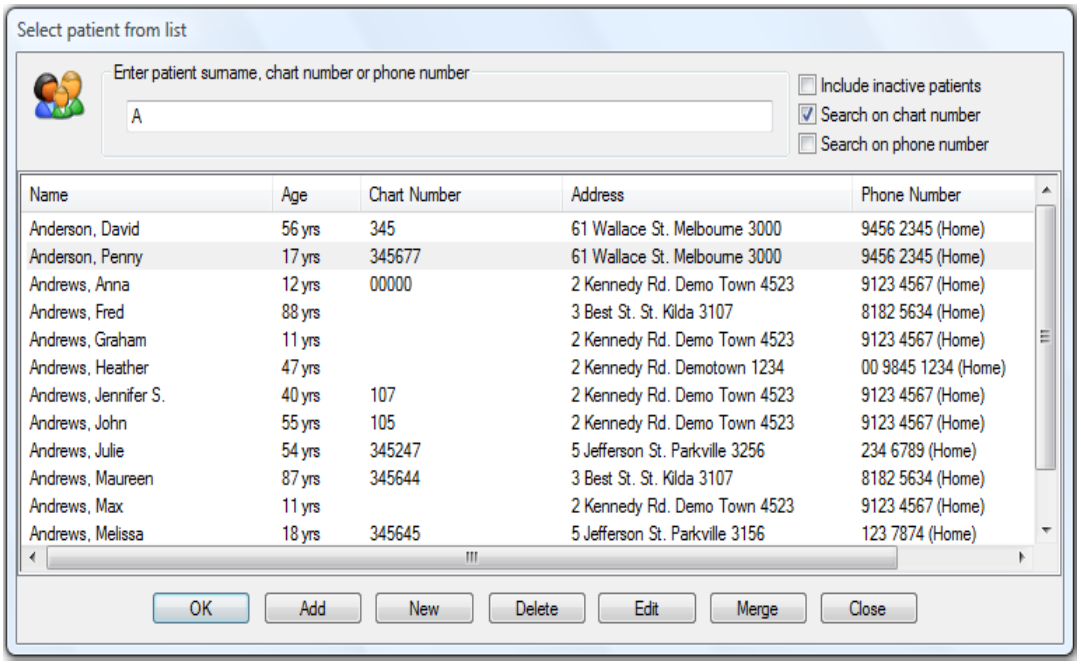
Figure 5.8 UCBB and Closure Set of Sending a Referral in MD


MD License
MEDICAL DIRECTOR AND PRACSOFT
Subscription Agreement
Version 3.11
By installing, downloading, accessing or otherwise using the Application or any Update of the Application you agree to be bound by this Agreement. If you DO NOT agree to the terms of this Agreement DO NOT install the Application or any Update or supplement to the Application.
Once the Application has been installed on your computer or otherwise used, any person subsequently opening or using the Application will be deemed to be a User and be subject to the terms and conditions of this Subscription Agreement.
Health Communication Network Limited of Level 1, 40 Oxley Street, Crows Nest, NSW 2065 ACN 068 458 515 (referred to in this Agreement as HCN) provides Medical Director and PracSoft software and all printed and electronic documentation, including any media, and instructional and operational manuals covering their use together with updates (referred to in
Copyright ? 1998-2010 Health Communication Network Ltd.
I Agree
Exit

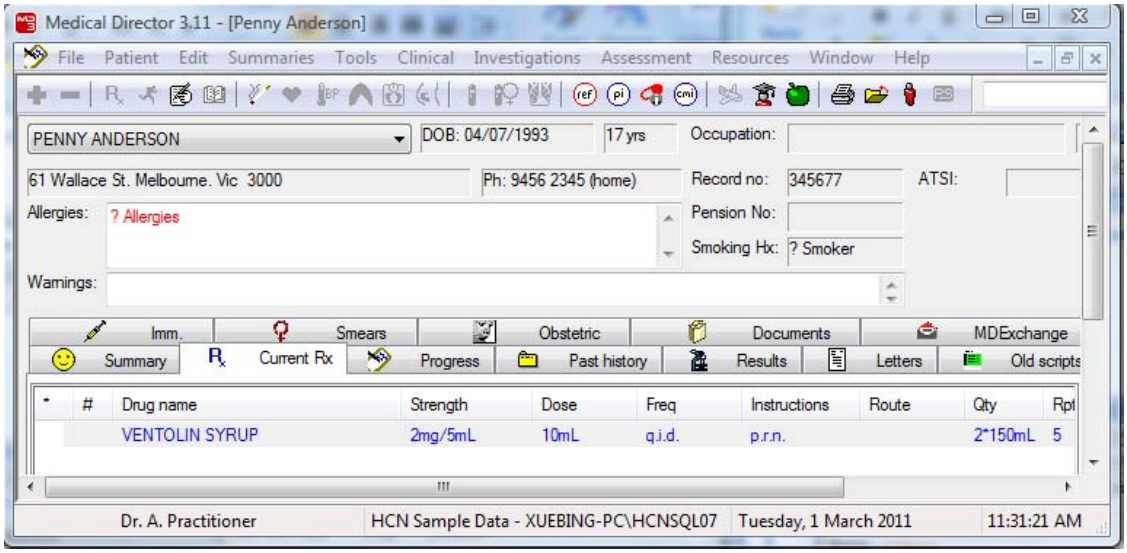
(a)


Log On to MD
User
Dr. A. Practitioner [Ap] [13 Best St.,Pill Land]
Password
Configuration
HCN Sample Data
Ok
Cancel

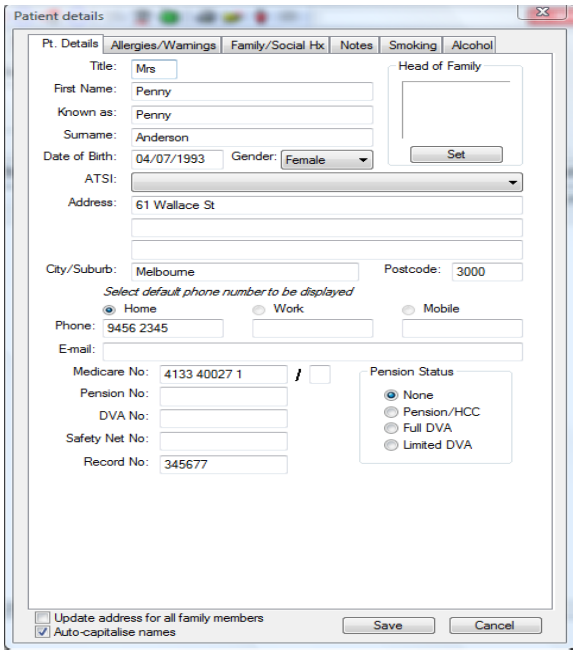
(b)

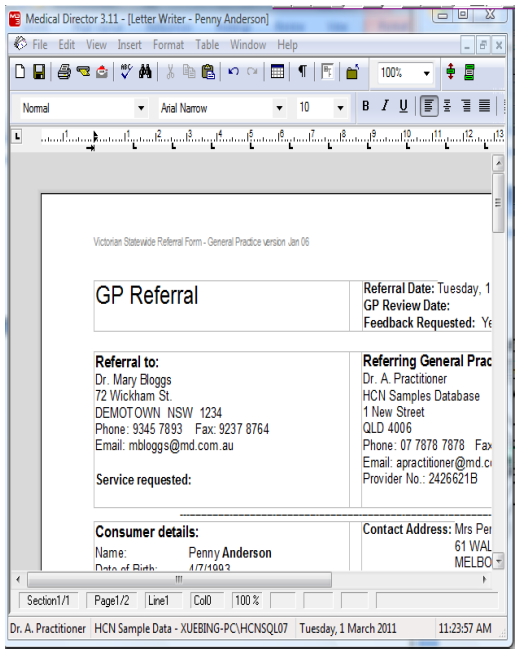

Select patient from list
Enter patient surname, chart number or phone number
A
Include inactive patients
Search on chart number
Search on phone number
Name | Age | Chart Number | Address | Phone Number
Anderson, David | 56 yrs | 345 | 61 Wallace St. Melbourne 3000 | 9456 2345 (Home)
Anderson, Penny | 17 yrs | 345677 | 61 Wallace St. Melbourne 3000 | 9456 2345 (Home)
Andrews, Anna | 12 yrs | 00000 | 2 Kennedy Rd. Demo Town 4523 | 9123 4567 (Home)
Andrews, Fred | 88 yrs | | 3 Best St. St. Kilda 3107 | 8182 5634 (Home)
Andrews, Graham | 11 yrs | | 2 Kennedy Rd. Demo Town 4523 | 9123 4567 (Home)
Andrews, Heather | 47 yrs | | 2 Kennedy Rd. Demotown 1234 | 00 9845 1234 (Home)
Andrews, Jennifer S. | 40 yrs | 107 | 2 Kennedy Rd. Demo Town 4523 | 9123 4567 (Home)
Andrews, John | 55 yrs | 105 | 2 Kennedy Rd. Demo Town 4523 | 9123 4567 (Home)
Andrews, Julie | 54 yrs | 345247 | 5 Jefferson St. Parkville 3256 | 234 6789 (Home)
Andrews, Maureen | 87 yrs | 345644 | 3 Best St. St. Kilda 3107 | 8182 5634 (Home)
Andrews, Max | 11 yrs | | 2 Kennedy Rd. Demo Town 4523 | 9123 4567 (Home)
Andrews, Melissa | 18 yrs | 345645 | 5 Jefferson St. Parkville 3156 | 123 7874 (Home)
OK
Add
New
Delete
Edit
Merge
Close

(c)



(d)





(e) (f)

Figure 5.9 Typical GUIs of Medical Director 3

Depending on the weight assigned to the transitions and the threshold length, the closure set of the UCBB may be different. Some transitions lead to opening a new window or dialog, some transitions lead to a tab change, and some other transitions just do editing without changing the state they belong to. In Figure 5.8, the five states, Current Medication, Progress, Results, Letters and Old Scripts are separated by tabs (Figure 5.9 (d)) and can be changed by selecting the corresponding tab. To explain the procedure of generating the closure set for the UCBB in Figure 5.8, the weight λ is assigned for each group of transitions (events) as follows:

- $\lambda_e = 3$ if e opens a new window or dialog or closes a windows or dialog;
- $\lambda_e = 2$ if e is a selection of a tab;
- $\lambda_e = 1$ if e is other events.

According to Definition 5.6, if we set the threshold length as 3, then only the blue transitions in Figure 5.8 will be included in the closure set. If we set the threshold length as 4, then the blue transitions, the dotted green transitions and the dotted green states will be included in the closure set.

With the closure set and UCBB of a use case, we can form a new set which contains highly relevant states and events to the use case. This set is called an envelope of the use case.

Definition 5.7 An ***envelope*** M_e of a use case is a 5-tuple (Σ,S,s_0,T, G), which is a combination of a UCBB M_s and a Closure Set M_c, *where* $M_e.s_0 = M_s.s_0$, $M_e.G=M_s.G$, $M_e.S = M_s.S \cup M_c.S$, $M_e.T = M_s.T \cup M_c.T$ *and* $M_e.\Sigma = M_s.\Sigma \cup M_c.\Sigma$.

Definitions 5.6 and 5.7 also present the method of generating an envelope from a GUITAM model and a UCBB of a use case. Figure 5.10 shows the algorithm for generating the closure set of a use case from a UCBB and a GUITAM model. Figure 5.11 shows the algorithm for generating the envelope of a use case.

Algorithm GenerateClosureSetOfUsecase

```
1.  GenerateClosureSetOfUsecase (Ms: SubSet, M: GUITAM, d: double, Mc: ClosureSet)
2.  {
3.      Mc.Σ= Ø; Mc.S = Ø; Mc.T = Ø;
4.      for each s ∈Ms.S{
5.          s'=GetRelevantState(s,M);
6.          GenClosureSet(Ms, M, d, s', Mc');
7.          Mc.Σ=Mc.Σ∪Mc'.Σ; Mc.S= Mc.S∪ Mc'.S; Mc.T = Mc.T ∪Mc'T
8.      }
9.  }
10. GenClosureSet(Ms:Subset, M:GUITAM, d:double, s:GUIState, Mc:ClosureSet, w: Walk)
11. {
12.     for each t ∈ M.T ∧ from(t) = s
13.     {
14.         w'= w ∪ {t} ;
15.         if (∑ λt' ≤ d ∧t' ∈ w')
16.             if(to(t) ∈ Ms.S) AddWalkToEnvelope(Mc, w')
```

17. else GenClosureSet(M_s, M, d, $to(t)$, w', d, M_c);
18. }
19. }

Figure 5.10 Algorithm for Generating Closure Set of a use case

In Figure 5.10, GenClosureSet is a recursive procedure which starts from a GUIState *s*, then depth-firstly traverses all possible paths which start from this state. A path with length less or equal to a given threshold *d* and with the last node is one of the states in the backbone will be considered as a by-side of the backbone. All corresponding states and transitions in the by-side are added to the ClosureSet M_c by calling the function AddWalkToEnvelope. When the GenClosureSet procedure finishes, the parameter M_c will contain all the by-sides that meet the requirement which are from the given state *s*. Function GenerateClosureSetOfUsecase calls the procedure GenClosureSet for each GUI state in the base set and unions all the ClosureSets together to form the full collection of by-sides which are close to the backbone.

Algorithm GenerateEnvelopeOfUsecase

1. GenerateEnvelopeOfUsecase (M_s: SubSet, M_c: ClosureSet, M_e: Envelope)
2. {
3. $M_e.\Sigma = \emptyset$; $M_e.S = \emptyset$; $M_e.T = \emptyset$; $M_e.G = \emptyset$;
4. $M_e.s_0 = M_s.s_0$,
5. $M_e.G = M_s.G$,
6. $M_e.S = M_s.S \cup M_c.S$,
7. $M_e.T = M_s.T \cup M_c.T$
8. $M_e.\Sigma = M_s.\Sigma \cup M_c.\Sigma$.
9. }

Figure 5.11 Algorithm for generating envelope of a use case

The algorithm GenerateEnvelopeOfUsecase in Figure 5.11 is used to generate an envelope for a use case. It takes a base set M_s as the backbone, a ClosureSet M_c and simply unions the corresponding collections to form the envelope of a use case M_e.

In Figure 5.8, the envelope of the use case for sending a referral letter in Medical Director 3 depends on the weight λ and the threshold length. If we use the assignment of the weight λ and threshold lengths, the envelopes of the use case are as follows.

- Length=3, the envelope includes all red transitions, red states, green transitions. See the part enclosed by the purple dashed line.
- Length=4, the envelope includes all red transitions, red states, blue transitions, blue transitions and blue states.

With the envelope of a use case, we can generate task oriented test cases. Each test case starts from the initial state s_0 in the envelope and ends at one of the goal states in G.

5.4 Use case envelope based test case generation

Apart from the goal set of states, other parts of an envelope of a use case constitute a subset of GUITAM. The purpose of creating envelopes of use cases is to generate task oriented long test cases. In an envelope, there is an initial state s_0 and a goal state set G. Any route from s_0 to a state in G can be seen as a test case. In comparison with the full GUITAM model of an AUT, the numbers of both transitions and states within an envelope is much smaller. Many typical scenarios contains only less than 10 steps with

which the generated envelope contains only very small numbers of transitions and states. The extremely small numbers of transitions and states even make it possible to test all paths which start from s_0 and end in one of the goal states in G within the envelope. All test cases generated from the envelope are related to the use case and are very effective for detecting defects which relate to the corresponding task.

For small envelopes, test cases can be all the different paths that start from s_0 and end in one of the goal states. The algorithm in Figure 5.12 depth-firstly traverses all paths which start from s_0 in the given envelope and takes all the different paths which end in one of the states in the goal state set G. These full paths form the set of task-oriented test cases.

Algorithm EnvelopeFullPathsTestCaseGenerating

```
1.  EnvelopeFullPathTestCaseGenerating(Me:Envelope, Ť :TestSuite)
2.  {
3.        Ť=Ø; w=Ø;
4.        EnvelopeFullPathTraverse(Me, Me.s0, w, Ť)
5.  }
6.  EnvelopeFullPathTraverse(Me:Envelop, s:GUIState, w:Walk, Ť :TestSuite)
7.  {
8.        for each t ∈ Me.T ∧ from(t) = s
9.        {
10.           if (t ∉ w){
11.              w' = w ∪ {t};
12.              if(to(t) ∈Me.G){
13.                 Testcase Ť' = createtestcase(w'); Ť = Ť ∪ {Ť'}
14.              }
15.              else EnvelopeFullPathTraverse(Me, to(t), w', Ť)
16.           }
17.       }
```

18. }

Figure 5.12 Algorithm for envelope based full path test case generating

The algorithm in Figure 5.12 exhaustively traverses all possible paths which start from s_0 and end in one of the goal states. This is only suitable for small envelopes. Not all envelopes of use cases are small. As a matter of fact, in many GUI-based applications such as clinic software systems, typical scenarios of use cases for different tasks may need tens of steps and involve many GUI states. The number of exhaustive paths of events grows exponentially as the number of states increases. Practically, only a subset of paths can be selected from the full collection, according to given coverage criteria. Since each test case generated from the envelope needs to go from s_0 to a goal state, the length-n coverage criterion in Definition 3.10 cannot be used in envelope based test case generation. Other criteria in Definitions 3.7, 3.8, 3.9, 4.1 and 4.2 can be used for the generation algorithm. The functional object coverage criterion and interactive object coverage criterion defined in Definitions 4.1 and 4.2 are preferable due to their large possible proportion of defects. To make use of these coverage criteria, the function EnvelopeFullPathTraverse in Figure 5.12 is simply modified to check the coverage criterion. It exits when the given coverage criterion is satisfied. Figure 5.13 shows the modified algorithm.

Algorithm EnvelopeTestCaseGeneratingWithCriterion

1. EnvelopeTestCaseGeneratingWithCriterion(M_e:Envelope, $\check{T}$:TestSuite, C:Criterion)
2. {
3. $\check{T}=\emptyset$; $w=\emptyset$;
4. EnvelopeFullPathTraverse(M_e, $M_e.s_0$, w, $\check{T}$, C)

```
5.  }
6.  EnvelopeFullPathTraverse(Me:Envelop, s:GUIState, w:Walk, Ť :TestSuite, C:Criterion)
7.  {
8.        for each t ∈ Me.T ∧ from(t) = s
9.        {
10.           if( meetcriterion(Ť, C)) return;
11.           if (t ∉ w){
12.              w′ = w ∪ {t};
13.               if(to(t) ∈Me.G){
14.                   Testcase Ť = createtestcase(w’);  Ť = Ť ∪ {Ť}
15.               }
16.               else EnvelopeFullPathTraverse(Me, to(t), w’, Ť)
17.           }
18.       }
19. }
```

Figure 5.13 Algorithm for Envelope Test Case Generating With Coverage Criterion

5.5 Experiment

For each subject application, some typical use cases were selected. Use cases were then turned into UCBB and encapsulated into envelopes. Due to the difference between the subject applications, the number of test cases generated varies significantly. Because there is only one window in Calculator, only two use cases were selected: normal calculation and statistics. Normal calculation is in the form of “operand1 operator operand2 =”, with which the buttons are divided into groups. Operands constitute buttons with captions of ‘0’ to ‘9’ and ‘A’ to ‘F’. Operators are those buttons with “+”, “-“, “*”, “/”, “sin”, “cos” and the like. When using the “Closure” method to generate the closure set, we used length=3, and λ=1 as parameters. Because events in normal calculation scenarios don’t change the GUI state, the backbone of the use case was unclear which made the closure method unable to collect

an efficient subset. Almost all the events were selected for the closure set, which made the envelope of this use case almost the same size as the full GUITAM model. When generating test cases with this envelope, the number of test cases was still very large. In ScreenDrawer, 10 use cases were used. While much more than in Calculator, only 324 test cases were generated for testing. The reason is that the scenarios of the use cases cover more states which form distinguishing backbones in the GUITAM model. With the clear backbone, envelopes of the use cases are much more efficient for generating effective test cases. Detailed information for the selected use cases and test cases is shown in Tables 5.1 and 5.2

Table 5.1 Use cases selected for each subject application

Subject application	Number of use cases	Use case names
Calculator	2	1. Calculation 2. Statistics
EasyWriter	9	1. Open file 2. Save file 3. Save as 4. Print 5. Print Setup 6. Copy-Paste 7. Set font 8. Set font color 9. Set background color
EnglishStudy	9	1. Select book-words 2. Select book-sentence 3. Maintain books 4. Maintain words 5. Maintain sentences 6. Import words 7. Import sentence 8. Delete words 9. Delete sentences

ScreenDrawer	10	1. Adjust Icon position 2. Set alarm 3. Calculator 4. Draw pencil 5. Draw rectangle 6. Select/Copy/Paste 7. Select/Move/Resize 8. Open saved image 9. Save image to file 10. Change pen size

Table 5.2 Use case and test case information for subject applications

Subject application	Number of use cases	Number of test cases
Calculator	2	5623
EasyWriter	9	724
EnglishStudy	9	626
ScreenDrawer	10	312

Test oracle information was also generated automatically from the base versions of the subject applications. By executing the selected test cases on the base version of each application, the state information was retrieved and saved after each event in each test case was performed. Table 5.3 shows the oracle information for each subject application.

Table 5.3 Oracle information for each subject application

Subject application	Test cases	Oracles size (MB)
Calculator	5623	582
EasyWriter	724	39
EnglishStudy	626	26
ScreenDrawer	312	16

Because the total number of test cases generated by the task-oriented method is much smaller than those used in Chapter 3, the oracle size of each test suite for each application was consequently much smaller, with the exception of Calculator. This is because most objects used in Calculator are buttons and the functions are mainly contained in one form which makes the envelope of each use case almost the same size as the whole GUITAM. The test case space cannot be reduced noticeably. Obviously, the task-oriented method proposed in this chapter is not suitable for applications such as Calculator, which provide only one form for all functionalities. Figure 5.14 shows the comparison between oracle sizes used in the Length-n method used in Chapter 3, the Classification Directed method used in Chapter 4 and the task-oriented method used in this chapter.

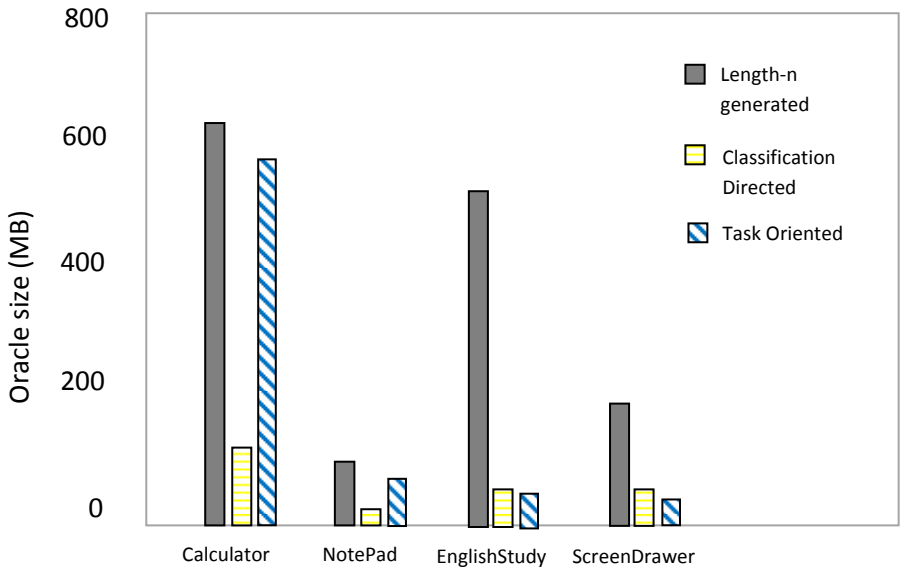


Figure 5.14 Comparison of oracle sizes generated by different methods

The effectiveness of task-oriented test cases depends on the use cases selected. Because the test cases are near the given task, the defects that are at a remove from the task usually won't be detected. If the use cases selected don't cover sufficient area of the subject application, the test cases generated from the envelopes of the use cases will be insufficient for testing. In Calculator, we only selected two use cases, but because most events don't lead to state transition, in the GUITAM of Calculator, most transitions are just a circle to the main state. The lengths from one event to another are almost all just one step, which is close enough to any backbone of the use case. The envelope of the given use case actually encapsulated almost all states and transitions in the GUITAM. The test cases generated for this envelope were similar to the test cases generated from the whole GUITAM. The defects detecting effect for Calculator in this chapter is also similar to the result detected by the method used in Chapter 3. Figure 5.15 shows the results of defects detected by the use case envelope based test cases.

This experiment still used the same faults seeded into the four subject applications. From Figure 5.15, even though only two use cases were used in Calculator, the fault detecting effect is almost the same as the results in Chapter 3. From Table 5.2, we found the reason why the test cases generated for Calculator have little relation to the use cases. The number of test cases selected for Calculator is almost the same as the number used in Chapter 3. This number is much larger than the numbers for the other applications. For EasyWriter and EnglishStudy, the faults detected in this experiment were fewer than the faults detected in Chapter 3. The reason for this is that the use

cases selected for these two applications didn't have sufficient coverage of all functions of the applications. In ScreenDrawer, the functions are organized in dialogs which make the paths in the GUITAM more distinct from each other. The use cases selected cover almost all the functions of the application. The results show that, even though the number of test cases is very small, the number of faults detected in this experiment was higher than that detected in Chapter 3. The comparison of test cases used and faults detected by different methods are shown in Figure 5.16 and Figure 5.17 respectively.

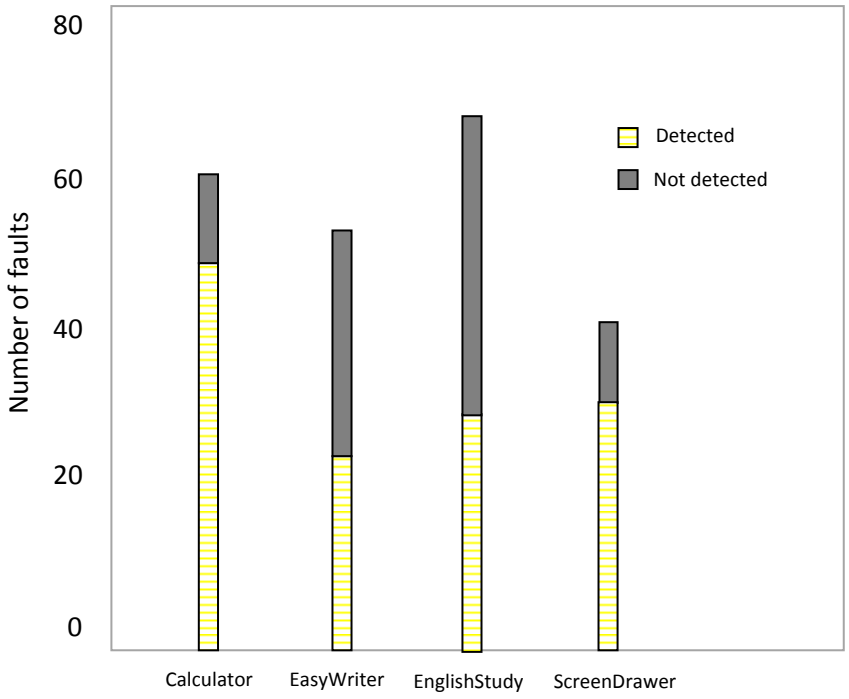


Figure 5.15 Envelope based test case faults detecting results

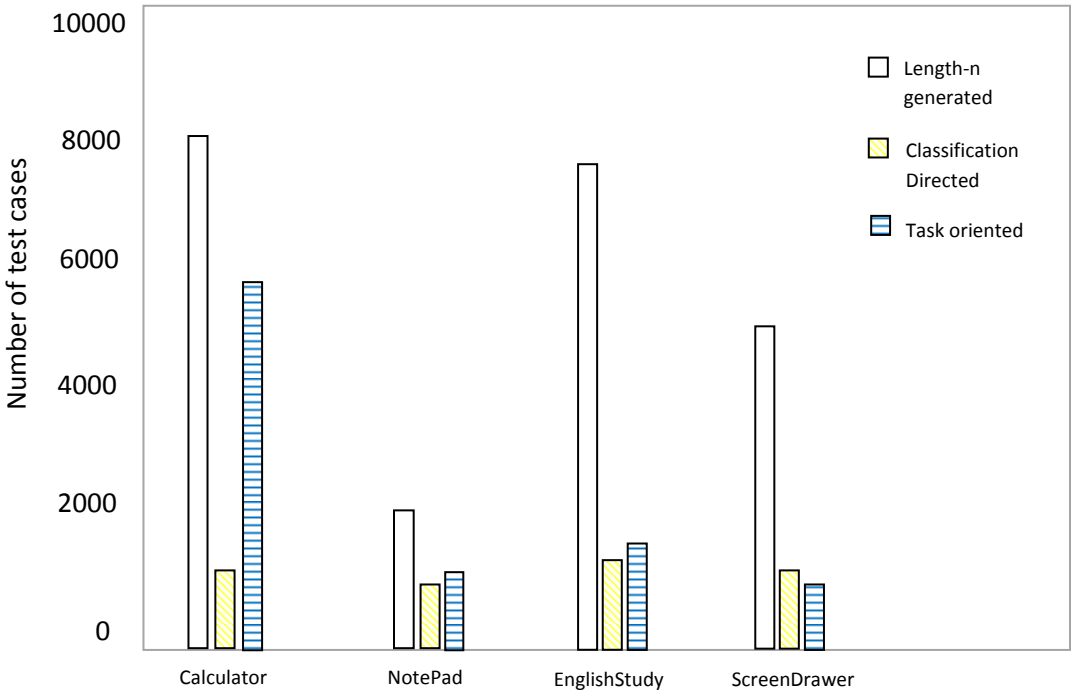


Figure 5.16 Numbers of test cases generated by different methods

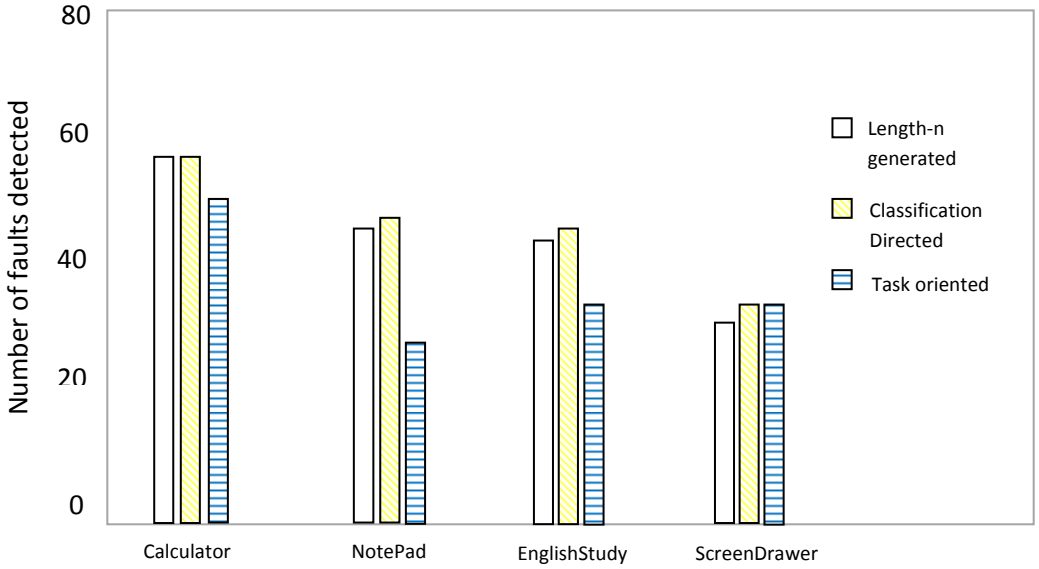


Figure 5.17 Number of faults detected by different methods

5.6 Conclusion

This chapter presented Long Use Case Closure Envelope Model for generating task-oriented long test cases by making use of use cases. By analysing traditional UML based use case activity diagrams and sequence diagrams, the corresponding formal definitions of a use case, use case activity diagram and use case sequence diagram (UCSD) were presented. To facilitate more efficient conversion, by modifying UCSD, this chapter also proposed a new model which is called detailed use case sequence diagram (DUCSD) to build in more detailed information about GUI states and events. Based on UCSD, a software engineer can easily create DUCSDs with a tool provided with the GUI Runner platform. From either an activity diagram, sequence diagram, or detailed sequence diagram, a small subset of GUITAM can be generated by selecting use case relevant states and transitions from the complete GUITAM of an AUT. This subset constitutes a use case backbone (UCBB). Algorithms were proposed to make these conversions. Because a use case is just an episode of an AUT, in general, the UCBB describing the corresponding use case contains much smaller numbers of states and transitions, which lead to exponentially decreased space of combinations of test cases. A UCBB simply contains the states and transitions that are described in an activity diagram, a sequence diagram, or a detailed sequence diagram, which are not enough for covering all possible user operations around the task. Based on the UCBB, a method called ‘Closure’ is used to collect the possible transitions and states which are ‘close’ to the backbone and form a broader set of UCBB. All the base set and the

expanded closure set are encapsulated in one envelope which contains an initial state, a set of goal states, intermediate states and transitions which connect the states.

An envelope is a self contained mini GUITAM with a set of goal states. Use case based test cases can be generated from the use case envelope rather than from the full GUITAM of the whole AUT. Each test case must cover a path that starts from the initial state and end in one of the goal states. Each test case is long enough to cover the full task from the beginning to the end. For a small envelope, exhaustive paths from initial state to goal state can be generated for test cases. For a larger envelope, test cases can be generated by applying certain coverage criteria.

Experiments were also carried out for generating task-oriented test cases. The results show that task-oriented test cases are effective and efficient for detecting task-related defects, especially for those applications which have distinct long use cases.

Chapter 6

Conclusions and Future Work

Graphic User Interfaces (GUIs) are widely recognized as a critical component of today's software. GUIs account for more than 45% of software code, which makes GUI testing paramount for providing quality software. Due to the characteristics of GUIs, conventional software testing methods are not suitable for GUI testing. GUI testing automation is needed because manual GUI testing is very laborious. This thesis has presented a unified solution to GUI testing automation. This solution includes the GUI modelling and testing automation model (GUITAM) for characterizing GUI states and its inherent logic, generating test cases based on given coverage criteria, creating test oracles, executing test cases, and verifying the execution results. To reduce the number of test cases without losing the testing quality, this research has also proposed Defect Classification Directed Test Cases Generation and a Long Use Case Closure Model for task oriented test case generation. The following sections will give a summary of our major contributions and future work.

6.1 Major contributions

The thesis presented a unified solution to GUI Testing Automation and proposed various models for effective test case generation. The main contributions of the thesis are as follows:

1) GUI Representation and GUI Testing Automation Model (GUITAM). At any given time, a GUI is represented as a forest of objects. The root of each tree in the forest is a window or a dialog. Other objects (widgets) are organized in a hierarchical manner and each object forms a node in the tree. Each object has properties and its corresponding values.

At a given moment, the GUI of an application is called a state. A GUI based application has a series of GUI states. Events can be performed on GUIs and trigger the transitions between GUI states. In this thesis, GUI states and transitions between states were modelled in the GUITAM, which characterizes the intrinsic logic of an application. The GUITAM of an application can be generated automatically. The GUITAM model is the critical element for the whole testing automation process. Our test cases generation, test oracle information creation and test case execution were all based on the GUITAM model.

2) Defect Classification Directed Test Case Generation. Because of the huge permutations of events, the space for all possible test cases is extremely large. This makes it impossible to exhaustively test GUI-based applications. To generate more effective test cases, defect classification and defect classification directed test case

generating algorithms have been proposed. This greatly reduces the number of test cases needed without losing the ability to detect GUI faults.

3) Long Use Case Closure Envelope Model for task oriented test case generation. Typical tasks provided in an application are usually made up of a number of steps. Existing methods limit the length of each test case to certain steps, normally three, to finish the tests at a practical time. Three steps are normally not enough to cover a task. This thesis proposed a Long Use Case Closure Enveloping Model to generate task-oriented long test cases for efficiently testing tasks. The model is especially efficient for long use cases. Each test case generated within the envelope covers at least a route from the initial state to one of the goal states. The test cases are very effective for detecting functional defects lurking inside the tasks.

4) To evaluate the effectiveness and efficiency, a system called GUITAM Runner has been implemented as a platform for GUITAM based testing. The system provides modules of GUITAM generation, test cases generation, test oracle generation, test case execution and validation. Four subject applications were selected for the evaluation and the results show that GUITAM based testing is practical and efficient. The testing results also show that the defect classification directed test cases method can greatly reduce the number of test cases without losing effectiveness. The use case envelope model was also proven to be practically efficient and effective for long task-oriented testing.

The contributions address two major challenges in GUI testing.

1) GUI testing is known to be laborious, costly, extremely time-consuming and difficult to automate. A unified solution has been provided to automate the GUI testing procedure which includes automatic GUITAM generation, automatic test case generation, automatic test oracle information generation, and automatic test case execution and validation.

2) Test case explosion. Defect Classification and the Long Use Case Closure Envelope Model were proposed in order to generate defect classification directed test cases and task-oriented test cases, which reduces the number of test cases exponentially without losing effectiveness.

6.2 Future work

Although GUITAM based testing achieves a high level of testing automation, there are still many obstacles to overcome in order to make it widely accepted, especially by software industries. The main obstacles to GUITAM-based testing are:

1) GUI reading techniques. GUITAM relies on the technique of reading GUI widgets information. In our platform, we use Microsoft C# in Microsoft Visual Studio .Net and the UIAutomation library to access the components of GUIs. We tested this method on other software, such as Genie and ZedMed (both of them are clinic software used in Australia), and found that not all the information for the GUIs can be read properly. To read GUI information for these kinds of software, we need to

find another method of scraping the GUI. Fortunately, the GUITAM uses highly abstract concepts which can be easily fitted with different GUI reading techniques.

2) Specification based oracle and test case generation. Languages used in specifications are not designed for testing purpose which makes it very difficult to automatically harness the information in the specifications. Manual work is still involved to convert them to an intermediate description for the automatic conversions.

3) State space explosion. In a GUITAM, a state-comparing function is used to distinguish states from each other. Judging whether two states are the same depends on the criteria of comparison. Different criteria lead to drastic changes in the number of states. For example, if any difference in any object's property values are considered, the number of possible states in a GUITAM can be infinite. In this thesis, only the bi-valued properties were considered, such as the 'Checked' value of checkboxes or radio buttons, and the continuous-valued properties were ignored, such as "Text' value in editboxes, 'Value' of a track-bar. Even just considering the bi-valued objects, the number of states may still be very large. Supposing there are n bi-valued objects, the combination of all possible values will be 2^n. When there are too many bi-valued objects, the number of states in a GUITAM will become extremely large, which is impractical for real testing.

Our future research work will focus principally on finding solutions to the obstacles listed above.

BIBLIOGRAPHY

[1] J. Gray, “What next? A few remaining problems in Information Technology”, ACM Federated Research Computer Conference, Atlanta, GA, May 1999

[2] A. M. Davis, Software requirements: objects, functions, and states. NJ, USA: Prentice-Hall, Inc., 1993.

[3] C. Kaner, J. Falk, and H. Q. Hguyen, "Testing Computer Software," in International Thompson Computer Press. London, UK, 1993.

[4] R. Mahajan and B. Shneiderman, "Visual & textual consistency checking tools for graphical user interfaces", University of Maryland, College Park May 1996 1996.

[5] B. A. Myers, "User Interface Software Technology", ACM Computing Surveys, vol. 28, pp. March 1996.

[6] F. Belli, A. Hollmann, and N. Nissanke, "Modeling, analysis and testing of safety issues – an event-based approach and case study", presented at Proceedings of the 26th Int. Conf. Computer Safety, Reliability, and Security, 2007.

[7] T.-H. Chang and R. C. Miller, "GUI testing using computer vision", presented at Proceedings of the 28th international conference on Human factors in computing systems, New York, USA 2010.

[8] O. E. Ariss, D. Xu, and B. Vender, "A Systematic Capture and Replay Strategy for Testing Complex GUI Based Java Applications", presented at Proceedings

of the 2010 Seventh International Conference on Information Technology, Las Vegas, Nevada, USA, 2010.

[9] I. Alsmadi, "Automatic Model Based Methods to Improve Test Effectiveness", Universal Journal of Computer Science and Engineering Technology, vol. 1, pp. 41-49, 2010.

[10] C. Bertolini and A.Mota, "Using ProbabilisticModel Checking to Evaluate GUI Testing Techniques", presented at 7th IEEE International Conference on Software Engineering and Formal Methods (SEFM), 2009.

[11] C. Bertolini and A. Mota, "A Framework for GUI Testing Based on Use Case Design", presented at Proceedings of the 2010 Third International Conference on Software Testing, Verification, and Validation Workshops, Paris 6-10 April 2010, 2010.

[12] K.-Y. Cai, L. Zhao, and F. Wang, "A Dynamic Partitioning Approach for GUI Testing", presented at 30th Annual International Computer Software and Applications Conference (COMPSAC'06), Chicago, Illinois September 17-September 21, 2006.

[13] C. Lowell and J. Stell-Smith, "Successful Automation of GUI Driven Acceptance Testing", Lecture Notes in Computer Science, vol. 2675-2003, 1011-1012, 2003.

[14] J. Andersson and G. Bache, "The Video Store Revisited Yet Again: Adventures in GUI Acceptance Testing", presented at Proceedings of the 5th International Conference on eXtreme Programming and Agile Processes in Software Engineering, LNCS 3092, 2004.

[15] S. Dutta, "Abbot – A friendly JUnit extension for GUI testing", Java Developer Journal, vol. 8, pp. 8-12, 2003.

[16] QTP, http://qtp.blogspot.com/

[17] Abbot, http://sourceforge.net/projects/abbot/

[18] Selenium, http://seleniumhq.org/

[19] RFT, http://www-01.ibm.com/software/awdtools/tester/functional/

[20] Win Runner http://www.loadtest.com.au/Technology/winrunner.htm

[21] Silk Test http://www.borland.com/us/products/silk/silktest/index.html

[22] Robot, http://www-01.ibm.com/software/awdtools/tester/robot/

[23] R. K. Shehady and D. P. Siewiorek, "A method to automate user interface testing using variable finite state machines", presented at Proceedings of The Twenty-Seventh Annual International Symposium on Fault-Tolerant Computing (FTCS'97), Washington - Brussels - Tokyo, 1997.

[24] L. White and H. Almezen, "Generating test cases for GUI responsibilities using complete interaction sequences", presented at ISSRE ’00: Proceedings of the 11th International Symposium on Software Reliability Engineering (ISSRE’00), Washington, DC, USA, 2000.

[25] A. J. Offutt and J. H. Hayes, "A semantic model of program faults", presented at ISSTA ’96: Proceedings of the 1996 ACM SIGSOFT international symposium on Software testing and analysis, New York, NY, USA, 1996.

[26] A. C. R. Paiva and J. C.P, "A Model-to-Implementation Mapping Tool for Automated Model-Based GUI Testing", Lecture Notes in Computer Science, vol. 3785/2005, 2005.

[27] A. C. R. Paiva, "Automated GUI Testing in Informática", XIII Convención Y Feria Internacional, 2009.

[28] X Yuan, A. M. Memon, "Generating Event Sequence-Based Test Cases Using GUI Runtime State Feedback", IEEE Transactions on Software Engineering, vol. 36, no. 1, pp. 81-95, Jan./Feb. 2010

[29] X. Yuan and A. M. Memon, "Generating Event Sequence-Based Test Cases Using GUI Runtime State Feedback", IEEE Transactions on Software Engineering, pp. 81 - 95, 2010.

[30] A. C. R. Paiva, J. C. P. Faria, and P. M. C. Mendes, "Reverse Engineered Formal Models for GUI Testing", Formal Methods for Industrial Critical Systems, Lecture Notes in Computer Science, vol. 4916, pp. 218-233, 2008.

[31] A. M. Memon, "An Event-Flow Model to Test EDS", Software Engineering and Development, (Enrique A. Belini, ed.), 2009.

[32] A. M. Memon, "Using Reverse Engineering for Automated Usability Evaluation of GUI-Based Applications", Software Engineering Models, Patterns and Architectures for HCI, 2009.

[33] F. Belli, "Finite-State Testing and Analysis of Graphical User Interfaces", presented at 12th International Symposium on Software Reliability Engineering (ISSRE'01), Hong Kong, China. November 27-November 30, 2001.

[34] F. Belli, C. J. Budnik, and N. Nissanke, "Finite-State Modeling, Analysis and Testing of System Vulnerabilities", presented at Proc. of Organic and Pervasive Computing Workshops (ARCS) 2004, Lecture Notes in Informatics (LNI), Augsburg, Germany, 2004.

[35] T. Tuglular, C. A. Muftuoglu, O. Kaya, F. Belli, and M. Linschulte, "GUI-Based Testing of Boundary Overflow Vulnerability", presented at 33rd Annual IEEE International Computer Software and Applications Conference, Seattle, Washington, USA, 2009.

[36] B. Das, D. Sarkar, and S. Chattopadhyay, "Model Checking on State Transition Diagram", presented at ASP-DAC '04 Proceedings of the 2004 Asia and South Pacific Design Automation Conference, Piscataway, NJ, USA 2004.

[37] A. M. Memon and Q. Xie, "Studying the fault-detection effectiveness of GUI test cases for rapidly evolving software", IEEE Transactions on Software Engineering, vol. 31, pp. 884-896, 2005.

[38] M. Vieira and o. Leduc, "Automation of GUI testing using a model-driven approach", presented at AST '06 Proceedings of the 2006 international workshop on Automation of software test, NY, USA, 2006.

[39] A. Kervinen and M. Maunumaa, "Model-Based Testing Through a GUI", Lecture Notes in Computer Science, vol. 3997/2006, pp. 16-31, 2006.

[40] A. C. R. Paiva, J. C. P. Faria, and R. F. A. M. Vidala, "Towards the Integration of Visual and Formal Models for GUI Testing", Electronic Notes in Theoretical Computer Science, vol. 190, pp. 99-111, 2007.

[41] H. Reza, S. Endapally, and E. Grant, "A Model-Based Approach for Testing GUI Using Hierarchical Predicate Transition Nets", presented at ITNG '07. Fourth International Conference on Information Technology, Las Vegas, NV 2-4 April, 2007.

[42] A. M. Memon, "A Comprehensive Framework for Testing Graphical User Interfaces", in Department of Computer Science. Pittsburgh: University of Pittsburgh, 2001.

[43] A. M. Memon, M. L. Soffa, and M. E. Pollack, "Coverage Criteria for GUI Testing", presented at Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering, Vienna, Austria, 2001.

[44] A. M. Memon, I. Banerjee, and A. Nagarajan, "GUI Ripping: Reverse Engineering of Graphical User Interfaces for Testing", presented at Proceeding of 10th Working Conference in Reverse Engineering., 2003.

[45] A. Memon, O. Banerjee, N. Hashmi, and A. Nagarajan, "DART: a framework for regression testing "nightly/daily builds" of GUI applications", presented at Proceedings of International Conference on Software Maintenance, 2003. 22-26 Sept. 2003. pp.410 - 419, 2003.

[46] A. Memon, I. Banerjee, and A. Nagarajan, "What Test Oracle Should I Use for Effective GUI Testing?", presented at 18th IEEE International Conference on Automated Software Engineering, Montreal, Quebec, Canada 2006.

[47] A. M. Memon, "Automatically repairing event sequence-based GUI test suites for regression testing", ACM Transactions on Software Engineering and Methodology (TOSEM), vol. 18, 2008.

[48] Q. Xie and A. M. Memon, "Automated model-based testing of community-driven open source GUI applications", presented at ICSM '06: Proceedings of the 22nd IEEE International Conference on Software Maintenance, Washington, DC, USA, 2006.

[49] Q. Xie and A. M. Memon, "Designing and comparing automated test oracles for GUI-based software applications", Transactions on Software Engineering and Methodology (TOSEM) vol. 16, 2007.

[50] Q. Xie and A. M. Memon, "Using a pilot study to derive a GUI model for automated testing", ACM Trans. on Softw. Eng. And Methodology, vol. 18, pp. 1-35, 2008.

[51] X. Yuan and A. M. Memon, "Using GUI Run-Time State as Feedback to Generate Test Cases", presented at 29th International Conference on Software Engineering, Minneapolis, MN 20-26 May 2007, 2007.

[52] H. Zhu, W. E. Wong, and F. Belli, "Advancing test automation technology to meet the challenges of model-driven software development", in report on the 3rd workshop on automation of software test, ICSE, 2008.

[53] P. Brooks, B. Robinson, and A. M. Memon, "An Initial Characterization of Industrial Graphical User Interface Systems", presented at International Conference on Software Testing Verification and Validation, 2009. ICST '09. , Denver, CO, 2009.

[54] J. Strecker and A. M. Memon, "Testing Graphical User Interfaces", Encyclopedia of Information Science and Technology, Second ed, 2009.

[55] S. Huang, M. Cohen, and A. M. Memon, "Repairing GUI Test Suites Using a Genetic Algorithm", presented at Proceedings of the 3rd IEEE International Conference on Software Testing, Verification and Validation, Washington, DC, USA, 2010.

[56] F. Belli, C. J. Budnik, and L. White, "Event-based modelling, analysis and testing of user interactions: approach and case study", SOFTWARE TESTING, VERIFICATION AND RELIABILITY, vol. 16, pp. 3-32, 2006.

[57] F. Belli and M. Linschulte, "Event-Driven Modeling and Testing of Web Services", presented at Annual IEEE International Computer Software and Applications Conference. Software Engineering. Addison-Wesley, 6th edition, Sommerville Ian, 2000.

[58] IEEE., "IEEE Standard Glossary of Software Engineering Terminology", ANSI/IEEE Std 610.12-1990, 1996.

[59] G. M. Kapfhammer and M. L. Soffa, "A family of test adequacy criteria for database-driven applications", presented at Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering New York, NY, USA, 2003.

[60] B. Pettichord, "Seven steps to test automation success", presented at Proceedings of the International Conference on Software Testing, Analysis, and Review, San Jose, CA, 1999.

[61] H. Zhu, P. A. V. Hall, and J. H. R. May, "Software unit test coverage and adequacy", ACM Computing Surveys (CSUR) Surveys Homepage archive, vol. 29, 1997.

[62] R. V. Binder, "Testing object-oriented software: a survey", Software Testing, Verification and Reliability, vol. 6, pp. 125-252, 1999.

[63] M. E. Fagan, "Design and Code Inspections to Reduce Errors in Program Development", IBM Systems Journal, vol. 15, pp. 182-211, 1976.

[64] B. Brykczynski, "A survey of software inspection checklists", ACM SIGSOFT Software Engineering Notes, vol. 24, pp. 82, 1999.

[65] O. Laitenberger and C. Atkinson, "Generalizing perspective-based inspection to handle object-oriented development artefacts", presented at Proceedings of the 21st international conference on Software engineering, 1999.

[66] F. Shull, I. Rus, and V. Basili, "Improving software inspections by using reading techniques", presented at Proceedings of the 23rd International Conference on Software Engineering, 2001.

[67] M. J. Harrold and G. Rothermel, "Performing data flow testing on classes", presented at Proceedings of the 2nd ACM SIGSOFT symposium on Foundations of software engineering, New York, USA 1994.

[68] G. Rothermel and M. J. Harrold, "Experience With Regression Test Selection", Empirical Software Engineering, vol. 2, pp. 178-188, 1996.

[69] G. Rothermel and M. J. Harrold, "A safe, efficient regression test selection technique", ACM Transactions on Software Engineering and Methodology (TOSEM), vol. 6, Issue 2, pp. 173 – 210, 1997.

[70] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold, "Test Case Prioritization: An Empirical Study", Proc. Int'l Conf. Software Maintenence, pp. 179-188, 1999.

[71] G. M. Kapfhammer, "Software Testing", in Department of Computer Science: Allegheny College, 2004.

[72] E. J. Weyuker, "The applicability of program schema results to programs", International Journal of Computer and Information Sciences vol. 8, pp. 387-403, 1979.

[73] E. J. Weyuker, "Translatability and decidability questions for restricted classes of program schemas", SIAM Journal on Computing vol. 8, pp. 587–598, 1979.

[74] E. J. Weyuker, "The evaluation of program-based software test data adequacy criteria", Communications of the ACM, vol. 31, Issue 6, pp.668-675, 1988.

[75] E. J. Weyuker, "Evaluating Software Complexity Measures", IEEE Transactions on Software Engineering, vol. 14, pp. 1357-1365, 1988.

[76] S. Rapps and E. J. Weyuker, "Data flow analysis techniques for test data selection", presented at ICSE '82: Proceedings of the 6th international conference on Software engineering, 1982.

[77] S. Rapps, E.J. Weyuker, "Selecting Software Test Data Using Data Flow Information", IEEE Transactions on Software Engineering, vol. 11, no. 4, pp. 367-375, Apr. 1985.

[78] J. Steven, P. Chandra, B. Fleck, and A. Podgurski, "jRapture: A Capture/Replay tool for observation-based testing", presented at Proceedings of

the 2000 ACM SIGSOFT international symposium on Software testing and analysis, New York, USA, 2000.

[79] A. K. Onoma, W.-T. Tsai, and H. Suganuma, "Regression testing in an industrial environment", Communications of the ACM, vol. 41, 1998.

[80] T. Ball, "On the limit of control flow analysis for regression test selection", presented at Proceedings of the 1998 ACM SIGSOFT international symposium on Software testing and analysis, New York, USA 1998.

[81] F. I. Vokolos and P. G. Frankl, "Empirical evaluation of the textual differencing regression testing technique", presented at Proceedings. International Conference on Software Maintenance, Bethesda, MD, USA 1998.

[82] B. A. Myers and M. B. Rosson, "Survey on user interface programming", presented at Proceedings of the SIGCHI'92, 1992.

[83] E. Bernard, B. Legeard, X. Luck, and F. Peureux, "Generation of test sequences from formal specifications: GSM 11-11 standard case study", Software Testing, Verification and Reliability, vol. 34, pp. 915-948, 2004.

[84] K. Bogdanov, et al., “Working together: Formal Methods and Testing”, FORTEST landscapes document, December 2003

[85] S. Rayadurgam and M. P. E. Heimdahl, "Test-Sequence Generation from Formal Requirements Models", presented at Proceedings of the Sixth IEEE High Assurance in Systems Engineering Workshop, Florida, 2001.

[86] M. Y. Ivory and M. A. Hearst, "The State of the Art in Automating Usability Evaluation of User Interfaces", ACM Computing Surveys, vol. 33, pp. 470-516, 2001.

[87] C. Kaner, "Improving the Maintainability of Automated Test Suites", presented at Proceedings of the Tenth International Quality Week, San Francisco, CA, 1997.

[88] C. Kaner, J. Bach, and B. Pettichord, "Lessons Learned in Software Testing: A Context-Driven Approach", John Wiley & Sons, 2002.

[89] C. Kaner, "Cem Kaner on Scenario Testing: The Power of 'What-If…' and Nine Ways to Fuel Your Imagination", Better Software, vol. 5, pp. 16-22, 2003.

[90] C. Kaner, "What is a Good Test Case? ", STAR East Conf. 2003, May 2003. Online: http://www.testingeducation.org/a/testcase.pdf, 2003.

[91] C. Kaner, J. Falk, and H. Q. Hguyen, "Testing Computer Software", New York, NY, USA: John Wiley & Sons, Inc, 1999.

[92] K. Zambelich, "Totally Data-Driven Automated Testing", Whitepaper(http://www.sqa-test.com/White_Paper.doc), conferred in October, 2010.

[93] N. Nyman. "Using monkey test tools", Software Testing and Quality Engineering, January/February: 18-21, 2000.

[94] N. Nyman, "In Defense of Monkey Testing", conferred in May, 2006.

[95] T. Dabóczi, I. Kollár, G. Simon, and T. Megyeri, "How to test Graphical User Interfaces", IEEE Instrumentation & Measurement Magazine, pp. 27-33, 2003.

[96] JUnit. www.junit.org

[97] NUnit. www.nunit.org

[98] T. Tuglular, C. A. Muftuoglu, F. Belli, and M. Linschulte, "Event-Based Input Validation Using Design-by-Contract Patterns", presented at the 20th annual International Symposium on Software Reliability Engineering (ISSRE 2009), Mysuru, India, 2009.

[99] M. Zitser, "Securing Software: An Evaluation of Static Source Code Analyzers", in Massachusetts Institute of Technology. MA: Cambridge, 2003.

[100] F. Belli and M. Linschulte, "On Negative Tests of Web Applications", Annals of Mathematics, Computing & Teleinformatics, vol. 1, pp. 44-56, 2007.

[101] F. Belli and M. Beyazit, "Mutation of Directed Graphs – Corresponding Regular Expressions and Complexity of Their Generation", EPTCS, vol. 3, pp. 69-77, 2009.

[102] F. Belli and B. Güldalı, "Software Testing via Model Checking", Computer and Information Sciences - ISCIS 2004,Lecture Notes in Computer Science, vol. 3280, pp. 907-916, 2004.

[103] Calculator, http://www.codeproject.com/KB/applications/caclulater.aspx

[104] EasyWriter , http://www.codeproject.com/KB/cs/notepad.aspx

[105] T. Illes and B. Paech, "An Analysis of Use Case Based Testing Approaches Based on a Defect Taxonomy", in IFIP International Federation for Information Processing, vol. 227, Software Engineering Techniques: Design for Quality, ed. K. Sacha, (Boston: Springer), pp. 211-222, 2006.

[106] T. Illes and B. Paech, "An Analysis of Use Case Based Testing Approaches Based on a Defect Taxonomy", Software Engineering Techniques: Design for Quality IFIP International Federation for Information Processing, vol. 227, pp. 211-222, 2007.

[107] S. Ramanna, R. Bhatt, and P. Biernot, "Software Defect Classification- A Comparative Study with Rough Hybrid Approaches", Rough Sets and Intelligent Systems Paradigms. Lecture Notes in Computer Science, vol. 4585, 2007.

[108] B. Robinson and P. Brooks, "An Initial Study of Customer-Reported GUI Defects", presented at Proceedings of the IEEE international Conference on Software Testing, Verification, and Validation Workshops, 2009.

[109] R. Chillarege, I. S. Bhandari, and J. K. Chaar, "Orthogonal defect classification-a concept for in-process measurements", IEEE Transactions on Software Engineering, vol. 18, 1992.

[110] Medical Director, http://www.hcn.com.au/

[111] L. Briand and Y. Labiche, "A UML-based Approach to System Testing", Carleton University 2002.

[112] I. Krsul, "Software Vulnerability Analysis", in Department of Computer Sciences, vol. Ph.D. COAST TR 98-09: Purdue University, 1998.

[113] K. Weidenhaupt, L. Pohl, J. Jarke, and P. Haumer, "Scenario Usage in System Development. A Report on Current Practice", presented at Third International Conference on Requirements Engineering, Colorado Springs, CO , USA, 1998.

[114] M. L. Lough, "A Taxonomy of Computer Attacks with Applications to Wireless", vol. PhD: Virginia Polytechnic Institute, 2001.

[115] G. J. Meyers, "The Art of Software Testing", John Wiley & Sons, New York, 1979.

[116] C. Phillips, E. Kemp, and S. M. Kek, "Extending UML Use Case Modelling to Support Graphical User Interface Design", presented at Proceedings of Software Engineering Conference, Canberra, ACT , Australia, 2001.

[117] A. Jain and B. D. Chaudhary, "A Use Case Driven Formal Approach to Check Consistency between UI Requirement and Implementation", in IEEE Region 10 and the Third international Conference on Industrial and Information Systems (ICIIS). Kharagpur INDIA 2008.

[118] M. Smit, E. Stroulia, and K. Wong, "Use Case Redocumentation from GUI Event Traces", in 12th European Conference on Software Maintenance and Reengineering. Athens, 2008.

[119] P. Mateo, D. Sevilla, and G.Mart´ınez, "Automated GUI testing validation guided by annotated use cases", presented at Proceedings of the 4th Workshop on Model-Based Testing (MoTes ’09) in Conjunction with the Annual National Conference of German Association for Informatics (GI ’09), L¨ubeck, Germany, 2009.

[120] J. M. Almendros-Jimenez and L. Iribarne, "Designing GUI components from UML use cases", presented at Proceedings of the 12th IEEE International Conference and Workshops on the Engineering of Computer-Based Systems (ECBS’05), 2005.

[121] D. Coleman, "A Use Case Template: Draft for discussion", Fusion Newsletter, April 1998. Available at http://www.hpl.hp.com/fusion/news/apr98.ppt., 1998.

[122] "UML Specification", http://www.rational.com/ referenced in March 2010.

[123] E. Hendrickson. “Making the right choice: The features you need in a GUI test automation tool", STQE Magazine, pages 20–25, November/December 2003.

[124] B. A. Myers, "User interface software tools", ACM Transactions on Computer-Human Interaction, vol. 2, pp. 64-103, 1995.

[125] A. C. R. Paiva, N. Tillmann, J. C. P. Faria, and F. A. M. Vidal, "Modeling and Testing Hierarchical GUIs", presented at Proceedings of the 12th International Workshop on Abstract State Machines, University of Paris 2005.

[126] B. Robinson and P. Brooks, "An Initial Study of Customer-Reported GUI Defects.Software Testing", presented at International Conference on Verification and Validation Workshops, Denver, CO 2009.

[127] Jemmy. www.jemmy.netbeans.org

[128] PIMENTA, ANA CRISTINA. “Automated Specification-Based Testing of Graphical User Interfaces", PhD Thesis, 2006

[129] GUITAR, http://guitar.sourceforge.net.

[130] Automation Anywhere, http://www.automationanywhere.com.