ABSTRACT

Title of Dissertation: EVENT-CODE INTERACTION DIRECTED TEST CASES
Ishan Banerjee, Doctor of Philosophy, 2016

Dissertation directed by: Professor Atif M. Memon, Department of Computer Science

The Graphical User Interface (GUI) is an integral component of contemporary computer software. A stable and reliable GUI is necessary for correct functioning of software applications. Comprehensive verification of the GUI is a routine part of most software development life-cycles. The input space of a GUI is typically large, making exhaustive verification difficult. GUI defects are often revealed by exercising parts of the GUI that interact with each other. It is challenging for a verification method to drive the GUI into states that might contain defects.

In recent years, *model-based* methods, that target specific GUI interactions, have been developed. These methods create a formal model of the GUI's input space from *specification* of the GUI, *visible GUI behaviors* and *static analysis* of the GUI's program-code. GUIs are typically dynamic in nature, whose user-visible state is guided by underlying program-code and dynamic program-state.

This research extends existing model-based GUI testing techniques by modelling interactions between the visible GUI of a GUI-based software and its underlying program-code. The new model is able to, efficiently and effectively, test the GUI in ways that were not possible using existing methods. The thesis is this: *Long, useful GUI testcases can be created by examining the interactions between the GUI of a GUI-based application and its program-code.*

To explore this thesis, a model-based GUI testing approach is formulated and evaluated. In this approach, program-code level interactions between GUI *event handlers* will be examined, modelled and deployed for constructing *long* GUI testcases. These testcases are able to drive the GUI into states that were not possible using existing models. Implementation and evaluation has been conducted using GUITAR, a fully-automated, open-source GUI testing framework.

EVENT-CODE INTERACTION DIRECTED TEST CASES

by

Ishan Banerjee

Dissertation submitted to the faculty of the Graduate School of the
University of Maryland, College Park, in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2016

Advisory committee:
Professor Atif M. Memon, Chair
Professor Rance Cleaveland
Professor Michel Cukier
Professor Mihai Pop
Professor James Purtilo

DEDICATION

the mountain top beckons me so,
come forth hither, 'tween the clouds,
smell the climes, feel the snow,
above the land, sprawled there proud.

my steps walk thither, at crests forlorn,
where are the treasures, promised afore?
no gold that shines, like sun at morn,
pearls all white, there are no more.

those new peaks, hidden from view,
jewels myriad, 'twere unseen from below,
calls me forth to tread anew,
shape a passage for thee to follow.

ACKNOWLEDGEMENTS

I would like to thank Professor Atif Memon for accepting me as a student and guiding me through the long journey. This dissertation is a product of his enthusiasm and encouragement.

I would like to thank past and present members of the GUITAR group at the University of Maryland, College Park. This dissertation has its roots in the work of past GUITAR members, namely Adithya Nagarajan, Cyntrica Eaton, Jaymie Strecker, Penelope Brooks, Qing Xie, Scott McMaster and Xun Yuan. Thanks to my colleagues Bao Nguyen, Bryan Robbins, Emily Kowalczyk, Ethar Elsaka, Leslie Milton and Zebao Gao whose thoughts are reflected in this dissertation.

Thanks are due to Professor Rance Cleaveland, Professor Michel Cukier, Professor Mihai Pop and Professor James Purtilo for serving on my thesis committee, providing valuable feedback about the research and reviewing the manuscript.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Contemporary computer software provides different methods for users to interact with the software. For example, a Unix operating system provides a command line *shell* to administrators, avionics software accepts pilot's inputs from aircraft control devices and a commercial banking system provides a web-based interface to depositors.

A popular method for interacting with computer software is via Graphical User Interfaces (GUI). Graphical User Interfaces are used across a wide spectrum of: 1) devices – enterprise computer systems, personal computers, hand-held computers, vending machine, operating heavy machinery 2) platforms – Android, Linux, Microsoft Windows, Solaris, world-wide web and 3) cost – $\$100$ e-reader, $\$100$M[1] supercomputer. The wide-ranging applicability and deployment of Graphical User Interfaces warrants continued investigation into GUIs, their usage, behavioral characteristics, security and reliability.

A GUI-based software system (GUI-based software or GUI-based application) is one that provides a GUI as a method of interacting with it. GUI-based software systems are the focus of the research presented in this study. They belong to the larger class of *Event-Driven* [28] software systems. Concepts presented here may be applicable, in part or whole, to other event-driven systems.

[1] http://en.wikipedia.org/wiki/Tianhe-2

1.1 Background and terminology

The GUI of GUI-based software presents a visual interface, by means of a visual display system. A user of the software can execute actions, using an appropriate input device (such as mouse, keyboard or touch) on the GUI. Terms, that will be used to describe the interaction of the user with the GUI, and testing the GUI are defined as follows:

widget: A GUI *widget* is a primitive element of the GUI. Many GUI widgets together constitute the GUI of a GUI-based application. For example, the *'Save'* and *'Cancel'* buttons in the *'Save As...'* dialog box of Microsoft Notepad are widgets. The layout of GUI widgets are often hierarchical, where a GUI *container* (such as the *'Save As...'* dialog box) contains a set of primitive GUI widgets, or other GUI containers.

event: A GUI *event* is an instance of executing an action (e.g.: *click*, *select*) by the user on a *widget* (e.g.: *'Save'*, *Checkbox*).

event sequence: An ordered set of GUI events is an *event sequence*. An event sequence can be executed on the GUI by the user or by a software agent such as a GUI testing tool. As an example, *'File'* $\longrightarrow$ *'Save As...'* $\longrightarrow$ *'Save'* is an event sequence that can be executed on Microsoft Notepad immediately after it has been launched.

testcase: A GUI event sequence that can be executed on a GUI-based application is a *testcase*. A testcase is typically executed on the application at a time when the application's GUI state is known, for example, immediately after the application is launched.

length-n testcase: A GUI testcase containing n *consecutive* GUI events of *interest*, is a *length-n testcase*. A length-n testcase may be prefixed with additional GUI events that make it possible to execute the first event in the length-n testcase. For example, *'Paste'* $\longrightarrow$ *'Edit'* $\longrightarrow$ *'Select All'* is a length-3 testcase in Microsoft Notepad. However, this may be prefixed with the event *'Edit'* to reach the *'Paste'* event, after the application is launched.

In this study, the terms *event* and *widget* may be interchanged, when the context is clear. For example, a statement 'the *Save* event was executed' would expand to 'the *click* event was executed on the *Save* button widget'.

1.2 GUI testing challenges

Functional verification of the GUI of GUI-based applications has always been an important component of their development life-cycle. This verification does not focus on the underlying 'business-logic' of the application, although defects in that part of the software may also be revealed. As with other software components, verification of the GUI has its own unique set of challenges

Verification of the GUI typically entails execution of a selected set of testcases on the GUI of the application, and comparing the resulting state of the GUI with an *expected* state (using an *oracle* [32]). Given a GUI, with hundreds or thousands of events, the possible number of input event sequences can be very large, potentially infinite (for example, a user can execute *'Edit'* $\longrightarrow$ *'Paste'* repeatedly on Microsoft Notepad). Hence, it becomes challenging to 1) identify the complete set of executable testcases 2) prioritize the testcases in a suitable order 3) execute testcases, manually or automatically, within a reasonable time.

In manual verification, a human tester typically 1) launches the application 2) identifies a testcase 3) clicks on widgets following the prescribed order of the testcase 4) repeats steps $2-3$ or $1-3$ until a testing criteria has been achieved. When this method is employed for testing the application, it becomes challenging for the tester to 1) identify all testcases 2) prioritize and select testcases according to some criteria 3) execute the selected testcases without error or omission.

Automated verification also has similar challenges. *Capture/Replay* (record/replay) [20] is a popular method for automated verification of the GUI. In this method, a human tester first executes a set of testcases, on the GUI, which are observed and *captured* by the *capture tool*. At a later time,

the *replay tool* executes the same testcases on the GUI to detect potential regressions. Using this method however, the human tester may still miss out 'important' testcases. Besides capture/replay, exhaustive execution of all possible testcases, in an automated manner, is typically impractical, owing to the large number of testcases and limited time.

In the last decade, *Model-based* testing [39, 42, 6, 34, 47, 43, 37, 38] has been adopted as a popular method for GUI testing. Some of the model-based methods have produced practical methods for 1) reverse engineering the GUI of a GUI-based software to extract its structure 2) creating a semantic model of the GUI such as EFG [43], EIG [34] from the extracted structure 3) generating 'important' testcases based on the semantic models. For example, an EFG identifies `follows` relationship between pairs of GUI events. An event e_y `follows` event e_x if e_y is available for execution immediately after executing event e_x. Models such as EFG and EIG have relied on structural information available from the visible run-time state of the GUI, for example, 'is the *Cancel* button enabled after editing a textbox'? Such testcases were shown to be effective at revealing defects in the application [34, 43, 47].

A GUI testcase typically drives the GUI of a GUI-based application from a starting state into different user-accessible states. As events are executed on the GUI, the visible state of the GUI changes. Naturally, executing longer event sequences on the GUI could make the GUI reach states that are not possible with a shorter event sequence. Executing a longer event sequence on the GUI increases chances of interactions between participating event handlers. It may execute program-code that depend on each other. Executing long event sequences therefore would be good at exercising interacting program-code and at revealing software defects. For example, in Microsoft Notepad, the event sequence – *'Edit'* $\longrightarrow$ *'Select All'* $\longrightarrow$ *'File'* $\longrightarrow$ *'New'* $\longrightarrow$ *'Edit'* $\longrightarrow$ *'Paste'* – would be more likely to reveal a software defect than the short event sequence *'Edit'* $\longrightarrow$ *'Select All'*.

Long event sequences are intuitively more likely to reveal software defects. At the same time, it becomes challenging for the human tester to select a long event sequence for executing on the

GUI. This is because, as longer event sequences are considered, the number of such executable event sequences grows rapidly. Not all of those event sequences may be equally good at driving the GUI into new states or at revealing defects. In addition, the number of available long event sequences is prohibitively large. Hence, prioritizing or selecting useful event sequences becomes an important part of the verification process.

The above discussion motivates the necessity of generating 'long' and 'useful' GUI testcases. Such testcases which are chosen from the large input space of the GUI, would drive the GUI into states that are not possible using shorter testcases.

1.3 Thesis statement

Software defects in the GUI of a GUI-based application may be revealed by executing long sequences of events on the GUI. This study presents a method to create long events sequences that are useful testcases for a GUI-based application. Program-code level interactions between GUI events are used as the basis for selecting the events sequences.

Program-code executed in response to a GUI event, called *event handler*, often share program-code components (such as functions or sub-routines) and state. Therefore, it is possible that the event handler of an event e_x influences the subsequent execution of the event handler for event e_y, thereby making it behave differently. Defects in the event handlers for such *interacting* events may be revealed by executing such events together.

This study seeks to 1) define program-code interactions between event handlers in GUI-based software 2) establish that such interactions exist in GUI-based software 3) describe a simple, practical method for detecting such interactions 4) create a formal model of the program-code interactions 5) generate long testcases based on such interactions 6) evaluate the effectiveness of such testcases at revealing defects. This information is leveraged to execute targeted GUI testcases that can reveal defects in the program code of GUI event handlers.

The thesis statement is:

Long, useful GUI testcases can be created by examining the interactions between the GUI of a GUI-based application and its program-code.

The following terms are defined for usage in the remainder of this study:

Definition: A *long GUI testcase* is defined as a testcase that contains 3 or more GUI events selected from a GUI model. ■

Definition: A *useful GUI testcase* is defined as a testcase that can exercise a GUI in a manner that reveals defects in its event handlers. ■

1.4 Approaches

This section lists approaches for modelling interactions between the GUI of a GUI-based application and its program-code (see Section 3.2). The models can be used for creating long testcases.

Radio Button Demo: In this and subsequent sections, a simple Java application, shown in Figure 1.1, called `Radio Button Demo` will be used as a running example. Different GUI models for this application will be constructed and explained. The application has one *top-level window*, `Radio Button Demo` and one *modal window*, `Exit Confirmation`, which is invoked by clicking the `Exit` button. Widgets of interest on the application are labeled w_0 through w_8. A GUI event can be executed on all these widgets except for w_4, which can only display GUI content. The program-code of this application is listed in Appendix A.

Approach 1: The following two event sequences are executed on the `Radio Button Demo` application (each after a fresh application start):

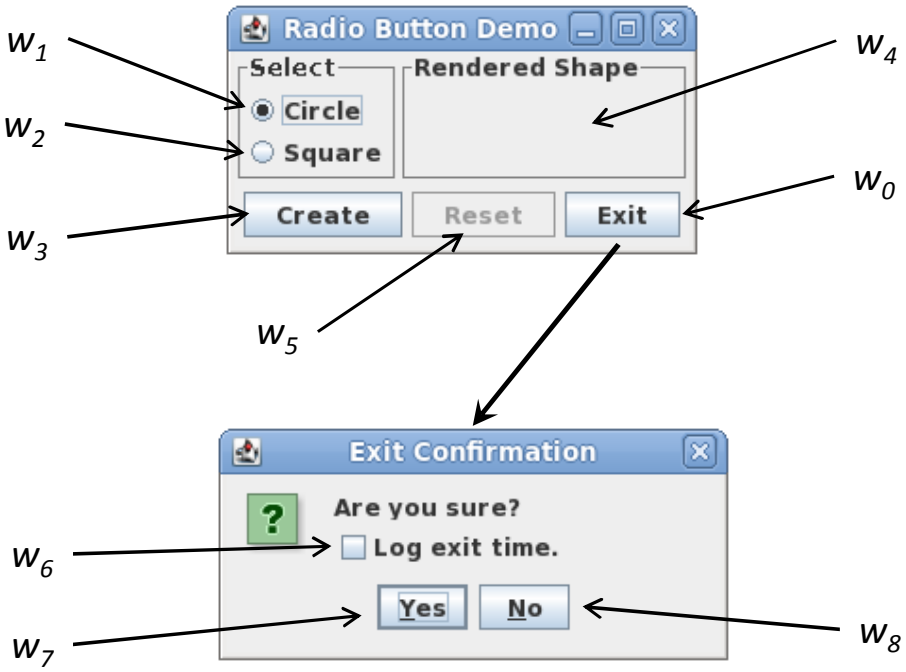


Figure 1.1: A Java application, `Radio Button Demo`. It has one top-level window, `Radio Button Demo` and one modal window, `Exit Confirmation`, that is opened by the `Exit` button. Nine widgets are labeled, $w_0 - w_8$.

(B) $circle \longrightarrow \underline{create}$

(C) $\underline{create}$

Columns (B) and (C) of Appendix A, show the *lines-of-code* executed by the event handler for the event $\underline{create}$ for the two cases. It can be seen that they are different. In fact, they differ at lines:

$$\{36, 37, 133, 137, 138, 141, 194, 195, 196, 200, 201, 220, 221, 222\}$$

This intuitively indicates that the event handler for $circle$ might have influenced, or interacted with, the event handler for $\underline{create}$ to make it behave differently. A defect in the interacting program-code can lead to incorrect execution. For example, the following defect in line 137:

original: `shape = new CirclePanel();`

defect: `shape = new SquarePanel();`

would lead to incorrect behavior of the application. This defect could be revealed by executing a testcase containing the event sequence $circle \longrightarrow \underline{create}$.

This approach detects events that interact at the program-code level. The interacting event sequence ($circle \longrightarrow \underline{create}$) is a good candidate to include in a model for modelling the interactions between the GUI of a GUI-based application and its program-code. In the above example, the defect can be revealed by generating event sequences containing interacting events.

Approach 2: In the preceding example, the event handler for event $circle$ was shown to interact with the event handler for event $create$. It is possible that a set of events do not independently interact with another event, However, those events may together interact with the other event.

For example, the events handlers for $circle$ and $create$ do not interact with the event handler for $\underline{exit}$. However, when the events $circle$ and $create$ are executed in sequence, it may interact with the event $\underline{exit}$. Here, the composite event $\{circle, create\}$ interacts with the event $\underline{exit}$, shown as $\{circle, create\} \longrightarrow \underline{exit}$.

In this approach, the event sequence $circle \longrightarrow create \longrightarrow exit$ is a good candidate for inclusion into the model. This approach may be used for identifying deeper interactions between GUI events at the program-code level.

It will be shown later that the GUI models created using the above approaches can be used to create long, useful GUI testcases. This study evaluates long testcases created using **Approach 1**.

1.5 Related work

Capture/replay and *model-based approaches* have been the most prominent methods of testing the GUI. Other techniques such as specification based, usage profile based, random testing and symbolic execution have also been reported.

Capture/replay has been a traditional method for testing the GUI. In this method, a human tester executes GUI events – such as mouse clicks and text-field entries – on the GUI. The *capture* tool observes and records these actions. At a later time, the recorded GUI events can be automatically executed on the GUI (potentially a newer version) by the *replay* tool. At that time, the replay tool can check for failure by observing the GUI or checking the application's execution log.

Early capture tools captured and recorded mouse coordinates and keystrokes as test scripts. As a result, the replayer would break if the screen coordinates or resolution changed during the replay phase. Modern capture/replay tools such as Abbot[2], Quick Test Pro[3] and Selenium[4] identify a GUI object using its GUI properties – such as name, color, width and height – and are more tolerant to changes in execution environment.

The research community has amply studied capture/replay tools and have augmented it in various ways. Ostrand et al. [36] develop a capture/replay tool to represent the capture actions as a flowchart. The flowchart can be altered by the tester and replayed back on the GUI. Ariss et al. [2] target Java applications where they augment a capture and replay tool with a model and automated test oracles, resulting in improved test coverage. Grechanik et al. [18] use the platform's *accessibility technologies* to better access GUI objects and their properties from the GUI. This has been used for emulating a human tester and automatically execute testcases on the GUI. Derezinska et al. [14] combine the capture/replay method with extraction of GUI properties from the application's binary executable. Chang et al. [10] develop a tool, Sikuli, based on the capture/replay method. Sikuli uses computer-vision technology to identify GUI objects during the replay phase and accurately identify GUI objects. Chen et al. [12] have combined capture/replay and specification based methods in GTT, a tool for testing Java applications. Hellmann et al. [19] use capture/replay to capture low-quality GUI images from hand-sketched prototypes, in a test-driven-development environment, and use replay for verifying the GUI implementation.

[2]http://sourceforge.net/projects/abbot/
[3]http://en.wikipedia.org/wiki/HP_QuickTest_Professional
[4]http://docs.seleniumhq.org/

During the last decade, model-based testing of GUI-based applications has received substantial attention [5]. Graph models such as Event-Flow Graph (EFG) [43], Event-Interaction Graph (EIG) [34], Event-Semantic Interaction Graph (ESIG) were created by *reverse engineering* the GUI using a GUI Ripper [26, 27]. The GUI Ripper was an enabling technology for daily/nightly regression testing [31, 25] of GUI-based applications. The GUI Ripper also enabled creating different levels of detail for GUI *oracles* [32]. Other models such as Covering Arrays, Complete Interaction Sequence and AI Planning models were also developed. Model-based testing of GUI applications are discussed in Chapter 2.

In *random testing* of GUI applications, random GUI events are typically executed on the application. This often serves as an effective smoke test during the development process. Hu et al. [22] test Android application by executing random events on it. In a similar approach, Dabczi et al. [13] tests MATLAB by generating random inputs to simulate a user. Takala et al. [41] model Android applications as a state machine and then use Monkey[5] to randomly choose transitions in this model.

Several other techniques have been developed for testing GUI applications. Chen et al. [11] have developed a tool to aid specification based testing of GUI applications. Ganov et al. [16] use symbolic execution to prune the event and data input space. Memon [29] monitors the usage of a GUI application to create testcases which can be replayed on a newer version of the same application. This can be used for automated regression testing during development. Garg et al. [17] generate user's GUI interaction usage characteristics to simulate a real user. Arlt et al. [3, 4] examine bytecodes of Java applications to infer static data dependencies between event handlers. These dependencies yield a *Event Dependency Graph* that can be used for directed testing.

The techniques discussed above largely rely on the visible GUI of the application under test (AUT) to create a model and generate testcases. There is no existing research that studies the interaction of visible GUI objects and its corresponding program code. While most GUI events directly or indirectly execute *event handlers*, the interaction between GUI events and event han-

[5] http://developer.android.com/tools/help/monkey.html

dlers and that between different event handlers are not known. This study attempts to understand how program code of an AUT behaves in response to GUI events and leverages this information to generate long testcases that are more likely to reveal GUI defects.

1.6 Challenges

To generate long testcases, this study introduces a new paradigm, *Event Code Interaction* (ECI), that models the effect of executing a GUI event e_1, on the execution of a subsequent GUI event e_2. This study is restricted to GUI-based applications. However, the concepts developed here may be applicable to other systems as well, especially those that can be considered to be *event-driven*.

When events are executed on the GUI of a GUI-based application, program code is executed within the application. This includes a) *Event Handlers*, which respond to the GUI event and 2) *Business Logic*, which performs the tasks expected from the application. The program state of an application is typically *read*, *modified* and *written* by the program code, typically in response to GUI events.

Program state may affect the program code being executed in response to GUI events. That is, the execution of a GUI event e_1 may affect the program state, that in turn *affects* the program code p being executed in response to a subsequent event e_2. The effect on p can be classified as 1) effect on program state, that is, the program state changed by e_2 may be different in the presence of e_1 2) program code, that is, the program code being executed by e_2 may be different in the presence of e_1.

If the program-state or program-code behavior of event e_2 is affected by the prior execution of event e_1, then event e_1 is said to *interact* with event e_2 at the *code-level*. In this study, the interaction is restricted to effects on program code only. It is shown as $e_1 \longrightarrow e_2$. This interaction is modelled as an *Event Code Interaction Graph* (ECIG). Vertices, for example e_1, e_2, in this graph

model GUI events and edges represent *code interaction*, for example, $e_1 \longrightarrow e_2$.

Edges in the ECIG are a subset of the edges from the EFG (or EIG) that have been identified based on program-code level interactions. The ECIG is a sparser graph than the EFG (or EIG). Therefore, using the ECIG, it is possible to generate a targeted testsuite containing *longer* (albeit *fewer*) testcases *and* execute them all in a reasonable time.

Code coverage has traditionally been a good metric for evaluating the effectiveness of a test-suite, or a testcase generation method. It has been shown in this study, that long GUI testcases may not exhibit increased code coverage, compared to similar shorter GUI testcases. This necessitates the use of a different metric to determine the usefulness of a testsuite. In this study, GUI state coverage has been used as a metric to determine the usefulness of a testsuite.

This study identifies the following challenges in testing GUI-based software and makes the corresponding contributions:

Challenge 1: It is challenging to identify *interactions* between program code corresponding to GUI events.

Contribution 1: A new paradigm, called *Event Code Interaction*, is defined, which models interactions between program code corresponding to GUI events (see Section 3.3).

Challenge 2: It is challenging to identify events in a GUI whose combined execution can reveal defects.

Contribution 2: Based on the ECI paradigm, an *Event Code Interaction Graph* is created for the GUI of a GUI-based application. This models GUI events which contain interaction at the program code level. Event sequences executed from this graph are likely to test combined execution of events which interact with each other (see Section 4.2.2).

Challenge 3: It is challenging to generate long event sequences that are more likely to reveal program defects.

Contribution 3: The ECIG is a sparse graph. Long testcases can easily be generated by traversing this graph, targeting specific parts of the GUI event space (see Section 5.3).

Challenge 4: It is challenging to identify new GUI states reached by an application when a test-suite is executed on it.

Contribution 4: A method is developed to identify new GUI states that are reached by an application when a testsuite is executed on it. These states are then compared to a baseline testsuite.

1.7 Outline

The remainder of this dissertation is structured in five chapters. Chapter 2 surveys existing model-based GUI testing techniques. Chapter 3 describes Event-Code Interaction, related concepts and models. Chapter 4 describes tools and testbeds deployed in this study. Chapter 5 presents results of evaluating ECI directed GUI testing. Chapter 6 identifies directions for future research, based on this work.

Chapter 2

Background

Verification of the GUI of GUI-based software have been performed using different approaches. Some of the popular techniques that have been attempted are:

1. feedback based [9]
2. specification-based [40]
3. genetic algorithms [45]
4. capture/replay-based [7, 2, 23]
5. model-based [38, 30, 47]
6. random-testing [8, 21]
7. execution-trace driven [1]

The methods developed in this study belongs to the *model-based* family of testing techniques. To provide a context for this research, a survey of model-based GUI testing techniques is presented in this chapter.

Model-based GUI testing can be broadly partitioned into two distinct phases. In the first phase, a *model* of the GUI is created. Creating the model may be *automated* or *manual* or a combination of both. In the second phase, testcases are generated (manually or automatically) based on the model. Most model-based GUI testing techniques are concerned with developing modelling techniques which detect more defects with a minimum number of testcases. Table 2.1 lists the GUI modelling techniques discussed in this section and the family of modelling techniques it belongs to.

Family	Model	Section
State Machine	Finite State Machine	2.1
	Variable Finite State Machine	2.2
	Complete Interaction Sequence	2.3
	Faulty Complete Interaction Sequence	2.4
Workflow	Event-Flow Graph	2.5
	Event-Interaction Graph	2.6
	Event-Semantic Interaction Graph	2.7
Event Sequence	AI Planning	2.8
	Genetic Algorithm	2.9
Combinatorial	Coverage Arrays	2.10

Table 2.1: Techniques developed to model the GUI of a GUI-based application. Testcases are typically generated based on the model and executed on the application.

SequenceLength-n: A popular (graph) model-based testcase generation method is called *SequenceLength-n*. In this method a depth-first-walk of the graph is executed starting from an event of interest. The path walked from the starting event is collected and produced as a testcase when depth n is reached. A SequenceLength-n testsuite contains *all possible* testcases of length-n generated from the graph of interest. The SequenceLength-n method of testcase generation will be used in this study.

2.1 Finite State Machine

Esmelioglu et al. [15] have modelled the GUI of a GUI-based application as a *Finite State Machine* (FSM). An FSM is defined as $FSM = (S, I, O, T, \Phi)$, where S is the set of finite GUI states, I is the set of inputs to the GUI, O is the finite set of outputs, T is the transition function

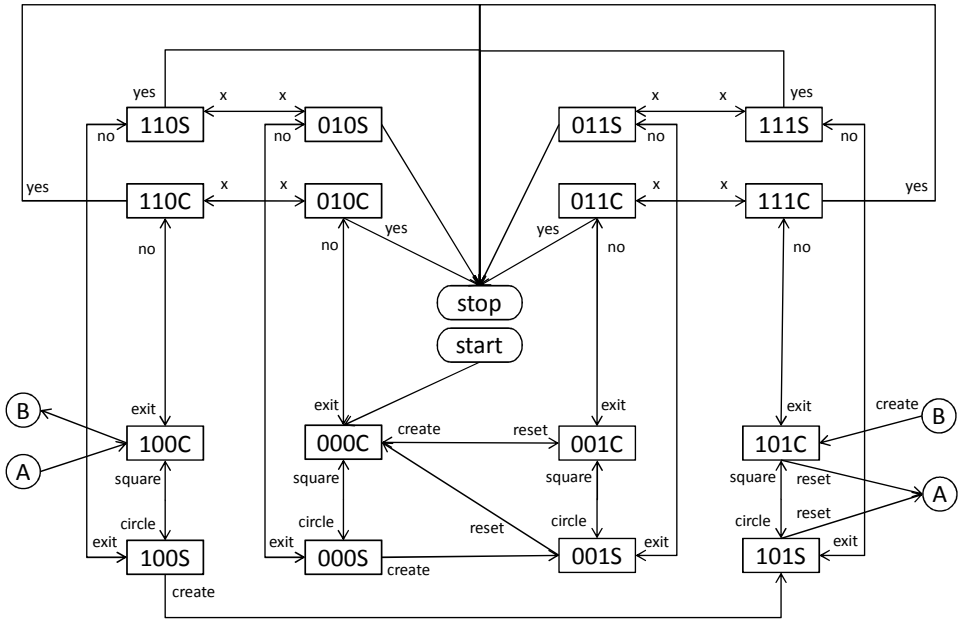


Figure 2.1: Finite State Machine for the `Radio Button Demo` application. State is represented with $LECS$ – L create log, E `Exit Confirmation` window opened, C shape created, S Circle/Square selected. Transitions are labeled as: $SRC \longrightarrow DST$, with $input$ marked at tail.

$S \times I \rightarrow S$ defining the next state based on the current state and input, Φ is the output function $S \times I \rightarrow O$ defining the output from a transition.

An FSM for the `Radio Button Demo` application is shown in Figure 2.1. In this example, each state represents 4 GUI elements – L, E, C, S, where $L = 0/1$ indicates that w_6 is (un)checked, $E = 0/1$ indicates that the `Exit Confirmation` window is closed/opened, $C = 0/1$ indicates that a shape is cleared/rendered, $S = C/S$ indicates that a circle/square radio button has been selected.

Deploying FSM-based GUI testing suffers from certain practical problems. First, the FSM may require a large number of states to represent the GUI. This is tedious to create, both manually and automatically, and also difficult to maintain for a GUI under development or maintenance. The states and transitions are difficult to intuitively map onto the actual GUI, resulting in greater maintenance effort.

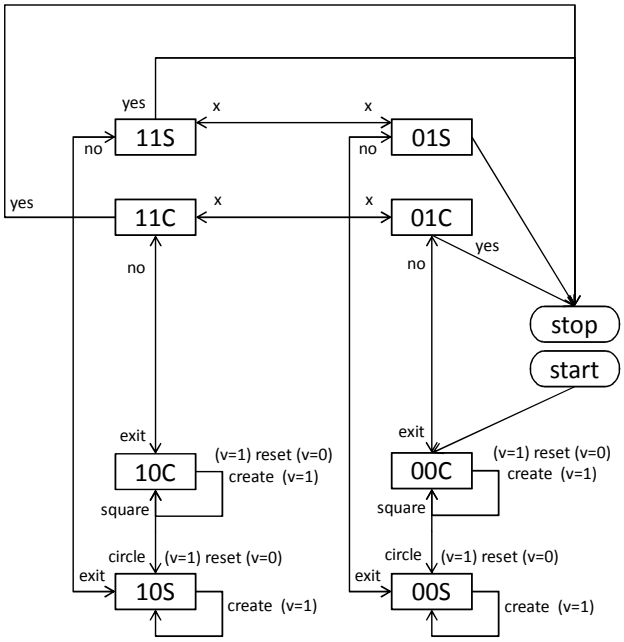


Figure 2.2: Variable Finite State Machine for the `Radio Button Demo` application. State is represented with LES – L create log, E `Exit Confirmation` window opened, S Circle/Square selected. Transitions are labeled as: $(precondition)SRC \longrightarrow DST(effect)$, with $input$ marked at tail. The variable V models the *created* state.

2.2 Variable Finite State Machine

Variable Finite State Machine (VFSM) is a technique employed by Shehady et al. [39] to represent the GUI states of a GUI-based software application. The authors shows that VFSMs require fewer states to represent the GUI, are more intuitive for representing a GUI and can be easily used to generate testcases and detect faults in a GUI-based application.

The key difference between FSM and VFSM is that VFSMs use of a set of global *variables*. The current value of these variables affect state transitions, which in turn may alter the value of the variables.

A VFSM is represented as a 7-tuple, $VFSM = (S, I, O, T, \Phi, V, \zeta)$. The symbols S, I, O posses the same semantics and properties as described for FSM (see Section 2.1). In addition, V is a set of n variables defined as $V = \{V_1, V_2, ..., V_n\}$, where each V_i is a set of values the $i - th$

global variable may assume. T is a state transition function, $T = D_T \rightarrow S$; Φ is an output function, $\Phi = D_T \rightarrow O$; where $D_T \in S \times I \times V_1 \times V_2 \times ... \times V_n$. This indicates that the state transitions, T and the output function Φ both are function of the global variables V. In addition, ζ is a state transition function that determines if the state of the global variables are altered as the result of a transition.

Figure 2.2 shows the VFSM for the `Radio Button Demo` application. In this figure, the state component C has been removed and is modelled with the variable V. A transition is labeled as $precondition \longrightarrow effect$, where the transition takes place only if the precondition is `true` with the postcondition being affected after the transition.

VFSMs produce smaller state machine which are more compact than FSMs, while retaining a similar state space. The VFSM is converted into an equivalent FSM in order to generate testcases. This is done by expanding the set of states, S and set of transitions, T using V and Φ.

2.3 Complete Interaction Sequence

White et al. [42] use Complete Interaction Sequence (CIS) as a method to prune the state space of a GUI-based application. The authors define a GUI *responsibility* as a GUI activity consisting of GUI objects that produces an observable effect on the GUI's environment – memory usage, peripheral device activity, underlying business logic response. A responsibility is identified manually by the tester. For each identified responsibility, the sequence of GUI events that lead to that responsibility is called the Complete Interaction Sequence for that responsibility.

A CIS can be tested individually by 1) manually identifying the GUI responsibilities in a GUI-based application 2) identifying the CIS for each responsibility 3) creating an FSM for a CIS 4) converting an FSM to a *reduced* FSM 5) testing a CIS using the reduced FSM to generate and execute testcases on the GUI-based application.

The conversion of the FSM to the reduced FSM is an interesting modelling abstraction. This

conversion is done by identifying *subFSM*s in the FSM for the CIS. A set of states $S = \{S_1, ..., S_n\}$ in an FSM form a subFSM if there exists a directed path from S_i to S_j, where S_i, $S_j \in S$. In addition, a subFSM S possesses *structural symmetry* if 1) in S, there exists states S_1 with only one incoming transition and S_2 with only one outgoing transition and there exists multiple paths from S_1 to S_2 2) states outside S do not affect transitions within S 3) choice of paths taken between S_1 and S_2 do not affect states outside S.

A subFSM with structural symmetry can be replaced with a single *superstate* in the FSM. The subFSM can now be tested in isolation. Identification and replacement of subFSMs reduce the overall complexity of the FSM for a CIS, resulting in a *reduced FSM*. The reduced FSM can also be tested by using any one path through a subFSM when its superstate is encountered in the reduced FSM.

Using the reduced FSM, the authors generate *design tests* – which assume that the FSM was implemented as designed and *implementation tests* – which attempt to execute transitions not present in the design of the FSM.

2.4 Off-nominal Finite State Machine

While most model-based GUI testing methods attempt to generate and execute testcases that invoke valid event sequences in the GUI of a GUI-based application, testing the GUI for invalid event sequences is also important. These testcases form a *negative* testsuite that verifies that the GUI does not permit a user to execute disallowed actions. Intuitively, generating invalid event sequences from a graph model of a GUI is straightforward. One needs to select an event-pair which does not share an edge and generate a testcase which contains this pair as an event sequence.

Belli et al. [6] generate testcases for event sequences that are invalid. They argue that these sequences should be tested in addition to valid sequences. A CIS (see Section 2.3) is used for creating a *Faulty Complete Interaction Sequence* (FCIS). Given a GUI, a CIS and its corresponding

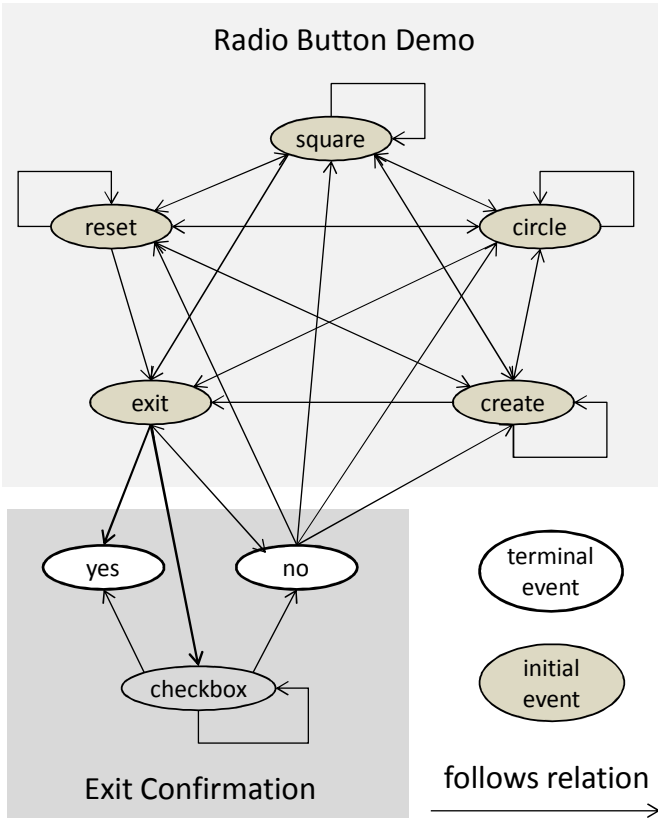


Figure 2.3: Event-Flow Graph for the `Radio Button Demo` application.

FSM with valid transitions is constructed. Missing edges in the FSM, termed *Faulty Interaction Pairs* (FIPs), are identified. A testcase containing an FIP can be easily generated by first generating a testcase leading to the first event in the FIP. This testcase is then prefixed to the FIP, creating a testcase with an invalid event sequence.

2.5 Event-Flow Graph

The Event-Flow Graph [43] is a GUI model, of a GUI-based application, that represents GUI events, and their sequences, that can be executed on the GUI. The Event-Flow Graph is a directed graph, where a vertex represents an *event* executable on the GUI. An edge $e_i \rightarrow e_j$ from vertex i

to j indicates that the event j is executable *immediately* after executing event i. Event j is said to *follow* event i. Intuitively, the Event-Flow Graph models the possible execution paths on the GUI.

Formally, an Event-Flow Graph is defined as a triple $< V, E, B >$, where V is a set of vertices representing events on the GUI of a GUI-based application; $E \in V \times V$ is a set of directed edges representing the `follows` relation; $B \in V$ is a set of vertices representing *initial events*, events that are available for execution immediately after the application is launched.

The Event-Flow Graph for the `Radio Button Demo` application is shown in Figure 2.3. In this figure, events are shown as ovals, shaded ovals represent initial events, directed edges represents the `follows` relation. From this figure, it can be seen that event *yes* can be executed after *exit*. However, *yes* cannot be executed after *create*. These corresponds to the GUI itself. Valid testcases can be easily generated by traversing the Event-Flow Graph, for example $square \rightarrow create \rightarrow circle \rightarrow exit \rightarrow yes$.

The Event-Flow Graph of a GUI-based application can be extracted from the run-time state of the GUI, by a process of reverse engineering [26]. In this process, a monitor application, called the *GUI Ripper* launches the application, identifies its top-level windows and GUI state, executes all visible events and continues identifying new windows which may be created. The GUI Ripper continues the process of executing events and extracting the GUI state of windows until as much of the GUI as possible has been traversed. This Event-Flow Graph is an approximation to the complete Event-Flow Graph, since the GUI Ripper may miss out some GUI windows and widgets [26].

The Event-Flow Graph represents valid executable event sequences on the GUI. Testcases can be easily generated from the Event-Flow Graph using different graph traversal algorithms starting from the initial events. Examples of graph traversal algorithms are goal-directed [33], random-walk [34]. Graph-pruning techniques based on the GUI's behavior such as EIG [34] and ESIG [47], may also be used to reduce the state space of the Event-Flow Graph.

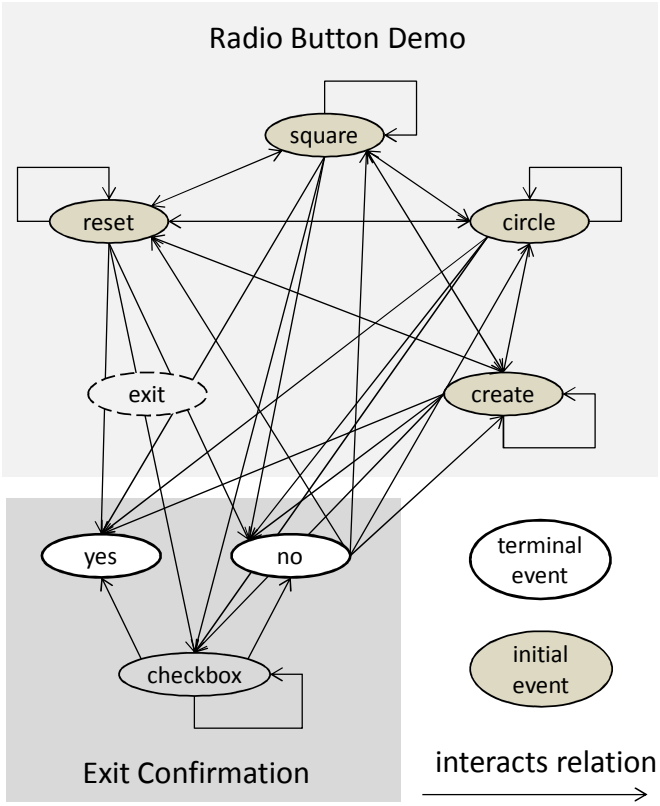


Figure 2.4: Event-Interaction Graph for the Radio Button Demo application.

2.6 Event-Interaction Graph

The Event-Flow Graph models all events and all possible event-sequences on the GUI. For typical GUI's the number of event sequences of a given length can grow exponentially even with small event sequence lengths. This leads to large testsuites with impractical execution times.

Xie et al. [43] empirically developed a method to prune the state space of the EFG. This increased the feasibility of generating testsuites with longer testcases and practical suite sizes. In their study the authors empirically concluded that *structural GUI events* – open/close menu item, open/close modeless windows – typically do not reveal defects in the application. This is likely because these events are typically executed by popular libraries which are well tested and defect-free.

On the other hand, *termination events* – where model windows are closed – and *system interaction events* – where the GUI interacts with the underlying business logic – are more likely to reveal defects. The Event-Interaction Graph was developed to model this logic.

Intuitively, an Event-Interaction Graph contains only *termination* and *system interaction* events. An edge, $x \rightarrow y$, between two events, x and y, indicate that event y is executable (not necessarily immediately) after event x. The edges in the Event-Interaction Graph models the *interacts with* relation.

An Event-Interaction Graph can be easily derived from an Event-Flow Graph based on GUI properties of each event [44]. The Event-Interaction Graph for the `Radio Button Demo` application is shown in Figure 2.4. This was obtained from the Event-Flow Graph by removing the *exit* event, since this event opens a modal window and is classified as a structural event.

Testcases are generated from the Event-Interaction Graph using graph traversal algorithms. Testcases generated directly from the Event-Interaction Graph may need to be augmented with missing structural events in order to make the testcase executable on the GUI.

2.7 Event-Semantic Interaction Graph

The GUI model represented using an Event-Interaction Graph can be further pruned based on semantic relationship between GUI widgets. Yuan et al. [47] created a sparse graph, where an edge was present between two GUI events only if executing one influenced the execution result of the other. The resulting representation called *Event Semantic Interaction Graph* greatly reduced the state space and was useful for generating longer testcases with a practical testsuite size.

Intuitively, if executing an event x affects the visible result of executing event y, then they are likely to share common program code or program state. In the `Radio Button Demo` application, the event handlers for the events *square* and *create* have common program state in the variables `created` and `currentShape`. As a result, executing the event sequences – 1) $square$

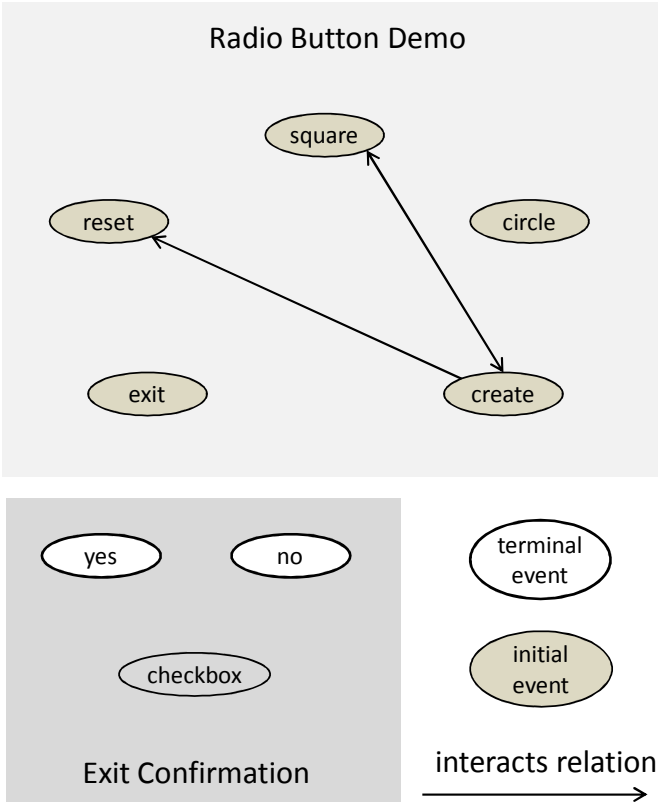


Figure 2.5: Event-Semantic Interaction Graph for the Radio Button Demo application.

in isolation 2) $create$ in isolation 3) $square \rightarrow create$ – show different resulting visible states on the GUI (Figure 2.6). In this figure, (a) is the initial state of the application. The resulting visible states in (b) and (c) are different from (d). The event *square*, when executed before the event $create$ makes the latter behave differently. The event $square$ is said to interact with the event *create*.

Such interacting events, in a GUI can be identified and modelled as an *Event Semantic Interaction Graph* (ESIG). The ESIG for the `Radio Button Demo` application is shown in Figure 2.5. As expected, there is an edge from the event $square$ to the event $create$. The ESIG typically has fewer edges than the Event-Flow Graph and Event-Interaction Graph. Using this representation it is possible to generate longer testcases which specifically test interacting events – for example, (*create, square)* and (*create, reset*).

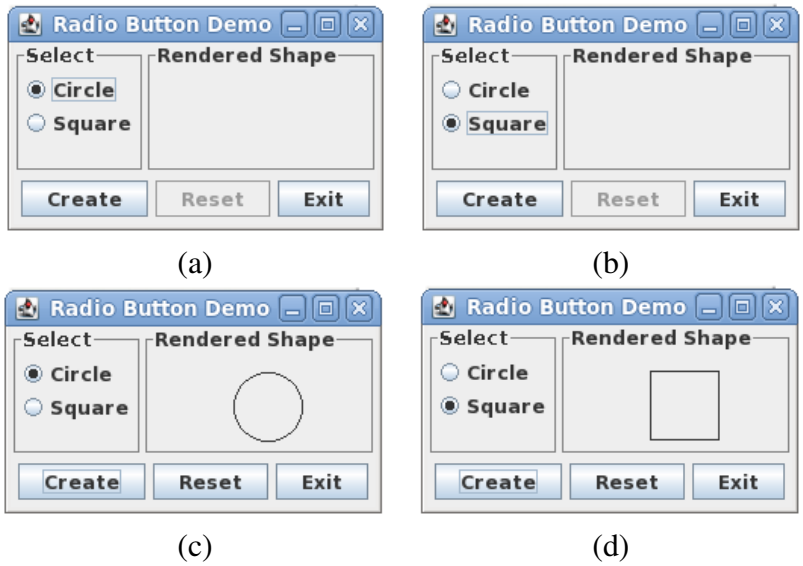


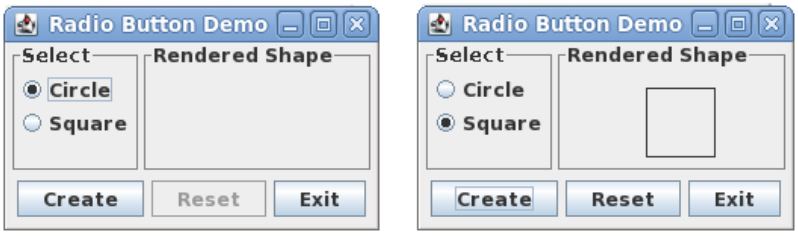
Figure 2.6: ESI relationship. (a) Initial application state (b) $square$ executed (c) $create$ executed (d) $square \rightarrow create$ in sequence. $square$ influences behaviour of $create$.

2.8 Planning

Plan Generation [33] has been used by Memon et al. to generate testcases for GUI-based applications. This is a goal-driven approach where the tester creates testcases by specifying the intended GUI task to be performed. The test generator produces the sequence of events that brings the application from the given *initial state* to the *goal state*.

The intuition in developing goal-driven testcase generation is that testers often find it easier to specify *what* needs to be done rather than *how* it needs to be done. Often, a GUI might present more than one, often convoluted, path for performing a task, which a tester may fail to test. Plan generation takes as input the starting GUI state and produces all possible paths for reaching the goal GUI state.

Generating testcases using Plan Generation works in two phases. In the first phase, the tester uses domain knowledge of the GUI to create *pre-condition* and *effect* Plan Operators. Plan Operators are building blocks for defining state transitions in the GUI. An operator models the state



Initial Goal

(a)

$$Initial \rightarrow Square \rightarrow Create \rightarrow Goal$$
$$Initial \rightarrow Create \rightarrow Square \rightarrow Goal$$

(b)

action	pre-condition	effect
$Square$	$isAccessible$(`Radio Button Demo`)	$focus$(`Square`) if `created` $\rightarrow$ $render$(`square`)
$Create$	$isAccessible$(`Radio Button Demo`)	$created \leftarrow$ `true` if focus(`square`) $\rightarrow$ $render$(`square`) if focus(`circle`) $\rightarrow$ $render$(`circle`)

(c)

Figure 2.7: Plan generation for Radio Button Demo application (a) Initial and goal states (b) Two generated plans (c) Plan operators for GUI.

of GUI entities that may trigger the operator. It also defines the effect of the operator on the state of GUI entities. In the second phase, the tester identifies tasks that need to be performed on the GUI by specifying the initial and goal state of the GUI. Thereafter, a plan generator, such as an AI planner, uses the plan operators and tasks to produce a set of testcases for each task.

Figure 2.7 (a) shows an initial and goal state of the `Radio Button Demo` application. There are many paths, in theory infinite, of transitioning the GUI from this initial state to goal state. The tester can specify a testing goal with this pair of states. The Planner will generate a testcase shown in (b). Figure (c) shows an example of how plan operators for this GUI may be defined.

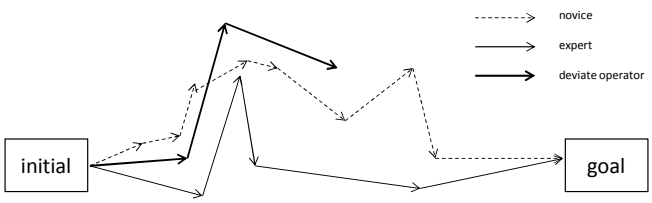


Figure 2.8: Genetic algorithm emulating a novice user's behavior to generate GUI testcase.

2.9 Genetic algorithm

Testcases on a GUI are sequences of events. It is possible to model a testcase as a *gene*, where each event is an *allele* (or chromosome). The goal of the genetic algorithm is to produce a set of event sequences that satisfy a *good* testing or coverage criteria.

Kasik et al. [24] have applied genetic algorithms to generate GUI testcases. In this work, the authors have attempted to emulate the event sequence a novice user would execute on the GUI. They argue, that a novice user would typically execute more events on the GUI in order to achieve a task than an expert user. The novice user's event sequence would execute unpredictable paths which the developer would not have predicted. This could thus trigger untested execution sequences and hence reveal defects in the GUI.

The algorithm begins creating a set of initial alleles, generating a *reward* score for each allele, replicating *good* alleles to the next generation of genes, applying mutation and crossover operators to enable better exploration of the search space. The goal of the genetic algorithm is to promote testcases that best resemble novice users. In this work, the authors claim that devising the best reward strategy for the alleles was a challenge. The reward system implemented a *deviate* strategy from the expert user's path to emulate novice users.

Figure 2.8 shows a typical path traversed by an expert user and by a novice user, to complete the same task. The genetic algorithm attempts to emulate the novice user by rewarding events that make the testcase *deviate* from the expert user's path.

2.10 Covering arrays

Yuan et al. [46] use *covering arrays* to generate long testcases. This approach can be used for testing an event in the *context* of other events. This is because an event, e_1, may behave differently when executed after another event e_2. In addition, a set of events, e_1, e_2 may behave differently when another event e_3 executed before either of them.

A *covering array*, $CA(N; t, k, v)$, is an $N \times k$ array on a set of v symbols such that every $N \times t$ sub-array contains all ordered subsets of all v symbols, at least once. This means that any t-columns of the array will contain all t-combinations of the v symbols. Using covering arrays, testcases are generated for testing the execution of a set of v events in t-way interaction such that each event occurs at each position in length-k testcases.

The testcase generation workflow operates by partitioning a GUI of a GUI-based application. Typically, a GUI windows forms a partition. Each event in a partition is considered as a symbol in v. Thereafter, testcases of length k are generated by creating a covering array of dimension $N \times k$. Each row of this array is a testcase. N, the number generated testcases, is minimized using optimization techniques. The final output is a set of N, length-k testcases.

Table 2.2 shows all 2-way interaction of events in the `Radio Button Demo` application's `Exit Conformation` GUI window (partition) with three events – *yes, no, (un)check*. Length-4 testcase are generated to test the effect of placing each event at sequence location 1 through 4. To test the placement of 3 events in all positions of a length-4 testcase, $P_3^4 = 81$ testcases are required. However, using a covering array, $CA(9; 2, 4, 3)$, only 9 testcases are required.

2.11 Summary

Model-based testing of the GUI of GUI-based applications have received considerable attention over the past decade [5]. Most techniques create a representation of the GUI based on automated or

2-way interaction	
yes	yes
yes	no
yes	$(un)check$
no	no
no	$(un)check$
no	yes
$(un)check$	$(un)check$
$(un)check$	yes
$(un)check$	no

$CA(9; 2, 4, 3)$ testcases			
yes	yes	yes	yes
yes	no	no	$(un)check$
yes	$(un)check$	$(un)check$	no
no	no	yes	no
no	$(un)check$	no	yes
no	yes	$(un)check$	$(un)check$
$(un)check$	$(un)check$	yes	$(un)check$
$(un)check$	yes	no	no
$(un)check$	no	$(un)check$	yes

Table 2.2: Covering Array $CA(9; 2, 4, 3)$ for the `Exit Confirmation` window of `Radio Button Demo` application. Only 9 testcases are required for testing 2-way interaction of 3 events at all 4 positions in a testcase. Exhaustive testing would require 81 testcases.

manual analysis of the GUI. Subsequently, testcases for testing the GUI are generated based on this representation. Researchers have focused on 1) obtaining a concise and accurate representation of the GUI 2) developing state space reduction techniques and 3) study the generation of effective testcases based on the model.

To the best of my knowledge, there has been no study that examines the interaction between GUI events and their interaction with program code of the GUI application being tested. The work presented in this study sets the stage for gaining a better understanding of this interaction.

Chapter 3

Event-Code Interaction

This chapter discusses different approaches that were considered for generating long GUI testcases. It also describes two models that are used for generating the testcases and evaluating their characteristics. The first is the Event-Code Interaction (ECI) model. It is used for modelling the GUI of an application. The second is hierarchical signatures. It is a method for enumerating the visible GUI states of an application.

3.1 Goals

The goal of this study is to develop a method for generating long testcases that can detect faults in event handlers of GUI-based applications. Testcases are generated based on the *Event Code Interaction* model created by observing code coverage metrics, from a seed testsuite, in in the program code of the application. The methods presented in this study have the following characteristics.

Model: This study seeks to find a relationship between events executed on the GUI of a GUI-based application and the program code executed on its behalf. Specifically, *interactions*, called *Event-Code Interactions*, between the program code for different event handlers are examined. These interactions are modelled as an *Event-Code Interaction Graph* (ECIG).

Effectiveness: Testcases generated based on the ECIG must be able to detect faults that are not detectable by other comparable techniques. The nature of generated testcases will target

the application, with better testcases, where other techniques may be inadequate. In other words, the model will generate long, useful testcases.

Efficiency: The steps required to analyze the application, create the ECIG model, use the model for testcase generation and execution must complete within a reasonable time. In addition, the generated testsuite must not contain duplicate or non-executable testcases.

Automation: The goal of this study is to develop a technique that is applicable to real-world, large-scale applications. Such applications may need to be tested daily (or more frequently) in a regression testing environment. The workflow must be fully automated to make this technique amenable for integration into third-party test harnesses.

The above goals aim to enhance existing model-based testcase generation techniques. In addition, it seeks to establish a new area of investigation for studying the relationship between events and program code that respond to the events.

3.2 Approaches

This work studies the relationship between events and corresponding program code of an application by first executing a *seed* testsuite on the application. The seed testsuite is an EIG-based SequenceLength-2 testsuite, that can be easily generated and executed. Code coverage metrics reported from execution of the seed testsuite are analyzed to infer ECI relations between events.

The following sections list questions that were asked before arriving at the above strategy.

3.2.1 Approach 1: Does one event influence another event?

During the execution of an event e_1, its event handler might *read*, *modify* and *write* program variables. Some of these program variable may also be accessible by the event handler of another event e_2.

It would be interesting to study if the event handler for e_1 affects the event handler for e_2. First, an event e_2 is executed immediately after launching an application and its behavior is recorded. In a subsequent execution, after launching the application, event e_1 is executed followed by e_2. Will e_2 *behave* differently in the second execution instance? If there is a difference, then execution of event e_1 influences the execution of event e_2 and executing these two events in a testcase may reveal faults that are not triggered when either event is executed in isolation.

This approach forms the basis for the Event-Code Interaction method presented in this study.

3.2.2 Approach 2: Do multiple events in combination influence another event?

Given a set of events $E = \{e_1, ..., e_n\}$, each individual event e_i, $0 < i < n$, may not posses any interaction with another event e. However, when the events in E are executed together, in that order, they may influence the behavior of event e. In this instance, the combination of a set of events interact with another event. Hence, a testcase that contains all events in E, in that order, followed by the event e, exercises the interaction between E and e.

This approach is an alternative to the Event-Code Interaction method and seeks to find complex interactions between events at the program-code level. It is not evaluated in this study (see Chapter 6).

3.2.3 How can a variation in event execution be detected?

This section lists four approaches using which an event e_1 can be detected as interacting with another event e_2. An event, e_1, is said to interact with another event, e_2, if the execution of e_1 immediately before e_2 makes e_2's behavior different than when e_2 is executed in isolation. Different metrics may be employed to signal the difference in behavior of e_2 in these two instances.

1. **Approach L1 (unique-lines-of-code):** In this method, the unique lines of program-code executed, as a result of invoking an event, is recorded. A distinction between single and multiple-executions of the same line is not made. In addition, the order of execution of the lines is not recorded. Consider the following code fragment, taken from an event handler:

```
1       sub a(N)                        4       sub b(N)
2       for (i = 0; i < N; i++)         5       for (i = 0; i < N; i++)
3         s += i                        6         s += i
```

   In this code, `a` and `b` are two functions. The lines of code executed by an event handler in three different invocations `A`, `B`, `C` by calling the functions `a`, `b`, with different input parameters, are given below. The metric of code coverage, **L1**, will not distinguish between the three different executions, since the set of unique lines executed is the same, $\{1, 2, 3, 4, 5, 6\}$, in all cases.

A	a(1); b(2)	1,2,3, 4,5,6,5,6
B	a(2); b(2)	1,2,3,2,3, 4,5,6,5,6
C	b(2); a(1)	4,5,6,5,6, 1,2,3

2. **Approach L2 (line-hit-count):** In this method, the number of times a program line is executed, is recorded. The sequences in which lines are executed are not recorded. In the example above, `A`, `C` are considered identical and distinct from `B`.
3. **Approach L3 (line-sequence):** In this method, the order of execution of program lines is recorded. Two executions are considered distinct if there is a difference in the recorded execution order. In the example above, `A`, `B`, `C` are all considered distinct.
4. **Approach S (program-state):** In this method, program state is recorded after invoking an event handler. The program state may include global data structures, heap and static storage. In the example above, the variable `s` can be considered as the only program state. By recording the value of `s` after each event handler terminates, `A`, `C` (`s`$= 3$) are considered identical and distinct from `B` (`s`$= 4$).

3.3 Event-Code Interaction

An event in a GUI-based application is typically associated with an *event handler*. An event handler is a program function that is executed by the application or operating system when an event is executed on the application. This function may in turn execute other functions, spawn threads and trigger timer-based code execution. Hence, an application-level event causes a set of lines of program code to be executed in the application and possibly in the platform. This study is restricted to program code that executes at the application level. Different events will trigger their own event handlers to be executed. The program code executed in response to different events may contain shared pieces of code. They may also share application state such as objects and data structures, which are accessed by the event handler when an event is executed.

Shared program code and program state between different event handlers create a possibility of *interactions* between them. For example:

Program code : An event handler may acquire a lock which is also required by another event handler sharing code. Hence, execution of one event handler may delay the execution of the other event handler.

Program state : An event handler may modify an object required by another event handler. This modification may affect the execution of the other event handler.

The `Radio Button Demo` application shown in Figure 1.1 will be used as a running example in this chapter. Appendix A lists the program code for the `Radio Button Demo` application. The `Radio Button Demo` application is written using the Java programming language. It consists of one file, *RadioButtonDemo.java*. This file contains one main class, *RadioButtonDemo*. The main class contains eight sub-classes. Of these eight sub-classes, five sub-classes define events handlers – *W0Listener*, *W1Listener*, *W2Listener*, *W3Listener*, *W5Listener*. The event handlers correspond to the widgets w_0, w_1, w_2, w_3, w_5, shown in Figure 1.1. Three sub-classes define GUI widgets – *CirclePanel*, *SquarePanel*, *EmptyPanel*. The file *RadioButtonDemo.java* has

239 uncommented lines of source code.

Appendix A also shows lines of code executed in response to certain events. For example, when the `Radio Button Demo` application is launched and the event sequence $square \longrightarrow create$ is executed, the lines of code in the column **A** are executed. An $\times$, on a given row indicates that that line was executed at least once. In this study, code coverage measurements follow the method **Approach L1 (unique-lines-of-code)** described in Section 3.2.

3.3.1 Event-Code Interaction

An event e_1, belonging to an application under test, A, is said to interact with another event e_2, belonging to A, at a program code level, if the execution of event e_1 alters the lines of code executed by event e_2. This interaction is represented as an edge $e_1 \longrightarrow e_2$ from event e_1 to event e_2. Altered lines of code execution is determined using the **Approach L1 (unique-lines-of-code)**.

3.3.2 Event-Code Interaction Graph

An ECIG is a graph representing ECI relations between the events of an application under test, A. In this graph, vertices represent events belonging to the GUI of the application. An edge is present from an event e_1 to another event e_2 if there exists an ECI relation from e_1 to e_2.

Definition: An Event-Code Interaction Graph, $ECIG$, for an application under test is defined as a triple, $ECIG = (V, E, I)$, where:

1. V = $\{v_1, ..., v_n\}$ is a set of n vertices such that each vertex represents an event in the application under test.

2. E = $\{e_1, ..., e_m\}$ and $E \in V \times V$ is a set of m directed edges. A directed edge e from vertex v_1 to v_2, represented by $v_1 \longrightarrow v_2$ represents an ECI relation between events represented by the vertices v_1 and v_2.

3. $I \subseteq V$ is a set of vertices representing *initial* events that are executable immediately after the application is launched. ■

Figure 3.1 shows the ECIG for the `Radio Button Demo` application. This ECIG was empirically determined. In this figure, V = $\{square, circle, create, exit, reset, yes, no, checkbox\}$, E = $\{square \longrightarrow create, circle \longrightarrow create, create \longrightarrow square, create \longrightarrow circle\}$, I = $\{square, circle, create, exit, reset\}$.

Algorithm: Given an application on which events can be executed, a simple algorithm can be followed to determine if one of its event, e_1, interacts with another of its event, e_2:

$e_1 \longrightarrow e_2$: Execute event e_1 as the first event immediately after launching the application. This might require *initial events* to be executed in order to reach the event e_1. Follow e_1 immediately with event e_2. Record the lines of code executed by event handler for e_2 – call it set X.

e_2: Execute event e_2 as the first event immediately after launching the application. This might require *initial events* to be executed in order to reach the event e_2. Record the lines of code executed by the event handler of e_2 – call it set Y.

ECI predicates: The following three predicates indicate whether the execution of event e_1 interacts with the execution of event e_2.

1. $X - Y \neq \Phi$. This indicates that certain lines of code were executed when the sequence $e_1 \longrightarrow e_2$ was executed (X), but were not executed when e_2 was executed in isolation (Y).
2. $Y - X \neq \Phi$. This indicates that certain lines of code were executed when e_2 was executed in isolation (Y), but were not executed when the sequence $e_1 \longrightarrow e_2$ was executed (X).
3. $X \neq Y$. This condition indicates that the lines of code executed by e_2 in isolation and by $e_1 \longrightarrow e_2$ are identical. However, there exists a set of lines, for which the hit count between X and Y are different.

In this study, ECI predicates 1 and 2 will be used to detect interaction. Hence, if ECI predicates 1 or 2 are true then the event e_1 is said to interact with the event e_2 at the program code level. ■

Example: As an illustration, consider the program code for the `Radio Button Demo` application. Columns (B) and (C) from Appendix A shows the following two code coverage information:

B : Shows the lines of code executed by the event handler for $create$ when the event sequence $circle \longrightarrow create$ is executed after launching the application.

C : Shows the lines of code executed by the event handler for $create$ when the event $create$ is executed after launching the application.

It can be seen that the lines of code executed by the event $create$ in isolation (C) are different from the lines of code executed by the same event handler when the event sequence $circle \longrightarrow create$ (B). The state variable `currentShape` affects the execution of $create$ in (B) at lines 133, $137 - 8$, 141. Additional lines of code – such as the method `CirclePanel.paintComponent()` – are executed in (B). The program state `currentShape` is set to `Shape.CIRCLE` by the event $circle$ influencing the event $create$ to execute different lines of code. This example shows that the event $circle$ interacts with the event $create$. Hence an edge $circle \longrightarrow create$ will be added to the ECIG for the `Radio Button Demo` application in Figure 3.1. The other 3 remaining edges are added based on a similar observation. ■

Some event pairs do not have any ECI relations. For example, the event $square$ does not interact with the event $circle$. In Appendix A, column (D) shows the lines of code executed by the event handler for $circle$ when the event sequence $square \longrightarrow circle$ is executed. This is identical to column (E) when $circle$ is executed in isolation. Since (D) and (E) are identical, the event $square$ has no interaction with the event $circle$.

Contribution 1: The ECI paradigm defined in this section models interactions between program-code (event handler) corresponding to GUI events.

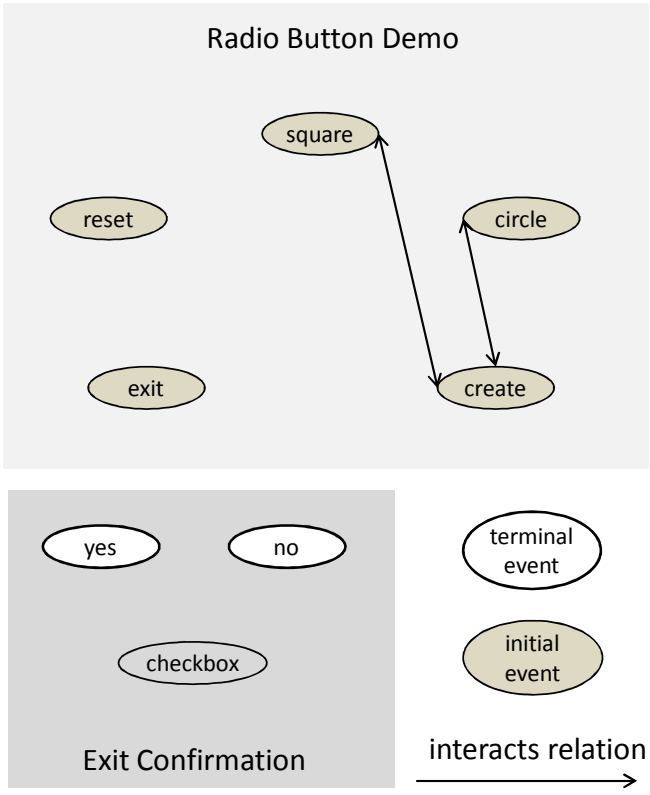


Figure 3.1: Event-Code Interaction Graph for the `Radio Button Demo` application.

3.3.3 Composite Event-Code Interaction

The concept of Event-Code Interaction, where an event e_1 is said to interact with another event e_2, can be extended to *composite events*.

Definition: A composite event, CE, in an application under test, is defined as an ordered set of events, $\{E_1, E_2, ..., E_n\}$ where each constituent event E_i, for $1 \leq i \leq n$, is executed in sequence without executing any other event in between two successive constituent event. ■

Composite ECI predicates: A composite event $CE_1 = \{E_1, E_2, ..., E_n\}$ is said to interact with another event e_2 of the application under test, at the program code level, if both the following predicates are true:

- The execution of the composite event E_1 alters the lines of code executed by the event e_2.

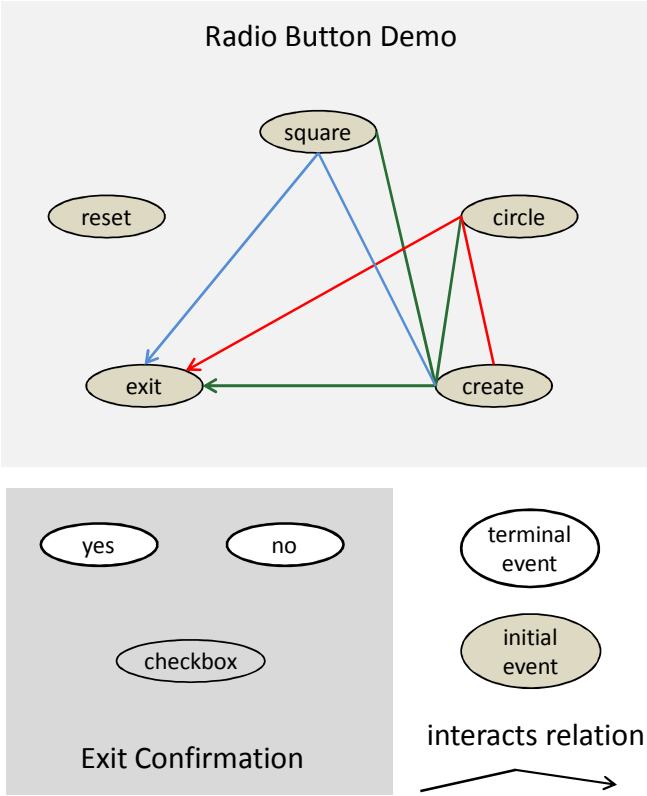


Figure 3.2: Event-Code Interaction Graph for length-2 composite events in the `Radio Button Demo` application.

- There are no ECI relations for any ordered subset of CE_1 ($E_1 \longrightarrow e_2$, ..., $E_n \longrightarrow e_2$, $\{E_1, E_2\} \longrightarrow e_2$ and so on).

Intuitively, this means that a set of constituent events independently do not interact with another event e_2. However, when the constituent events are executed as a composite event, their combined execution does interact with the event e_2. ■

Example: Consider the `Radio Button Demo` application example. Composite events in this application of different lengths are like – $\{circle, create\}$ of length-2, $\{create, exit, no\}$ of length-3, $\{create, circle, reset, exit\}$ of length-4. For simplicity of description, consider composite events of length-2. Figure 3.2 shows the *interacts* relation of the `Radio Button`

`Demo` application with length-2 composite events. The following three sets of *interacts* relations were empirically determined:

1. $\{create, circle\} \longrightarrow exit$
2. $\{create, square\} \longrightarrow exit$
3. $\{square, create\} \longrightarrow exit$; $\{circle, create\} \longrightarrow exit$

In this figure, each set from the list above is shown in a distinct color. An *interacts* relation is shown as a set of two edges with the same color. The first edge is undirected and connects the two vertices of the composite event. The second edge is directed and connect the last event of the composite event and the interacting event. For example, the composite event $\{circle, create\}$ interacts with $exit$. It is shown as $circle$ — $create \longrightarrow exit$. In this example, none of the constituent events – $create$, $circle$, $square$ – of the composite events interact with the event $exit$.

As an illustration of a composite event interacting with another event, consider the ECI $\{circle, create\} \longrightarrow exit$. In Appendix A, column (F) shows the lines executed by the event handler for the event $exit$ when the event sequence $circle \longrightarrow create \longrightarrow exit$ is executed. Column (G) shows the lines of code executed by the event handler for $exit$ when executed in isolation. It can be seen that (F) and (G) are different. In (F) the line 163 is executed, in addition to the lines from (G). This indicates that the composite event $\{circle, create\}$ interacts with the event $exit$. From Figure 3.1, it can be seen that the constituent events, $circle$ and $create$ do not interact with the event $exit$. ■

Composite ECI relations described in this section are not evaluated in this study. It is a concept that can be investigated in future (see Chapter 6).

3.4 GUI states

A visible GUI state (simply, GUI state) consists of all visible GUI windows and their widgets. A GUI moves from one visible state to another as events are executed on it.

The usefulness of a testsuite, under study, is typically measured by comparing its execution results with that of another testsuite, obtained by a known method. Typical metrics that help assess the usefulness are 1) code coverage 2) fault detection with mutation testing and 3) performance metrics such as execution time and testsuite size. Execution of long GUI testcases can, intuitively, lead the application to reach GUI states that are not reachable using short testcases. In this study, ECIG based testcases will be considered useful, if they can drive the GUI, of a GUI-based application, into states that were not possible using other comparable methods.

This section describes a *fast* method to detect if new GUI states were reached, while executing a target testcase. In this method, a baseline testsuite is first executed on the application under test and all GUI states reached by the application are recorded. Thereafter, a target testsuite is executed. If GUI states are reached, using the target testsuite, that were not seen while executing the baseline testsuite, then those states are considered as *new* GUI states.

3.4.1 Challenges

Comparing two sets of GUI states, to determine if one set contains states not present in the other set, poses the following practical challenges:

Resource: The GUI state reached by the GUI of an application under test is recorded after each event of a testcase is executed. For example, a testsuite with 1000 testcases where each testcase contains 2 events will require $2 * 1000 = 2000$ GUI states to be recorded. When a target testcase with 2 events is executed, the GUI state after executing each event must be compared with 2000 recorded states to determine if the target testcase drove the GUI into a state that was not en-

countered by the baseline testsuite. For a target testsuite with 1000 testcase of length-2 each, this translates to $1000 * 2 * 1000 * 2 = 4$ million comparison of GUI states. This is an $O(n^2)$ algorithm and is computationally resource intensive.

Platform: Comparing two specific GUI states to determine if they are different (or identical) poses certain challenges. Since GUIs are hierarchical in nature, each state may contain GUI containers that may contain sub-containers that may finally contain GUI widgets. Let us consider two hypothetical and visually identical GUI states – `A` and `B` – that contain only widgets – `a, b, c, d` – as follows:

```
A = {a, b, c, d}
B = {a, b, d, c}
```

The two GUI states contain the same widgets and are visually identical. However, the software layer that renders and extracts the GUI components may report a different ordering. The algorithmic complexity to determine if the two states are different (or identical) is non-trivial. This is because a real-life GUI may contain many levels of hierarchy and ordering differences at any level. For example, if `A` and `B` were given as:

```
A = { {a, b}, {c, d}, { {e, f}, {g, h} } }
B = { {c, d}, { {g, h}, {e, f} }, {a, b} }
```

`A` and `B` could be visually identical. However, a software layer may change the ordering of the components when queried.

The above two challenges motivate the development of a *fast* and *simple* method to compare a GUI state with a set of known GUI states.

3.4.2 Hierarchical signature

A GUI widget is associated with a set of attributes and corresponding values. Consider a button widget called `Exit` that has been extracted from the GUI state of an application. The widget has the attributes `width, height, color, text` with corresponding values `100, 30, 0xfff, Exit`. It is possible to create a signature from these values as follows:

$$\phi = \{\texttt{width}, \texttt{100}\}, \{\texttt{height}, \texttt{30}\}, \{\texttt{color}, \texttt{0xfff}\}, \{\texttt{text}, \texttt{Exit}\} \tag{3.1}$$

where ϕ is the set of attributes and corresponding values of the attributes, of the widget `Exit`.

$$\phi' = \{i, j\}, \{k, l\}, \{m, n\}, \{o, p\} \tag{3.2}$$

where ϕ' is a set containing a signature, computed from the set ϕ, such that i = $hash$(`width`), j = $hash$(`100`) and so on. Here, $hash$ is a suitable hash function to convert the attribute and value into an integer.

$$h = i + j + k + l + m + n + o + p \tag{3.3}$$

where h is a signature computed from the attributes and values of the widget `Exit`.

Using the above three steps, a signature can be computed for any widget of a GUI application. As a result of the commutative operator $+$, the signature h is insensitive to the order in which the attributes appear.

Extending the above concept, it is possible to compute a signature for each component of the hierarchical GUI structure of a GUI state. Computing the signature begins at the leaf widget and continues upwards until the GUI windows are reached.

Figure 3.3 (a) shows the set of signatures computed for the GUI component hierarchy of `RadioButton Demo`'s initial GUI window. Figure 3.3 (b) shows the hierarchical nature of

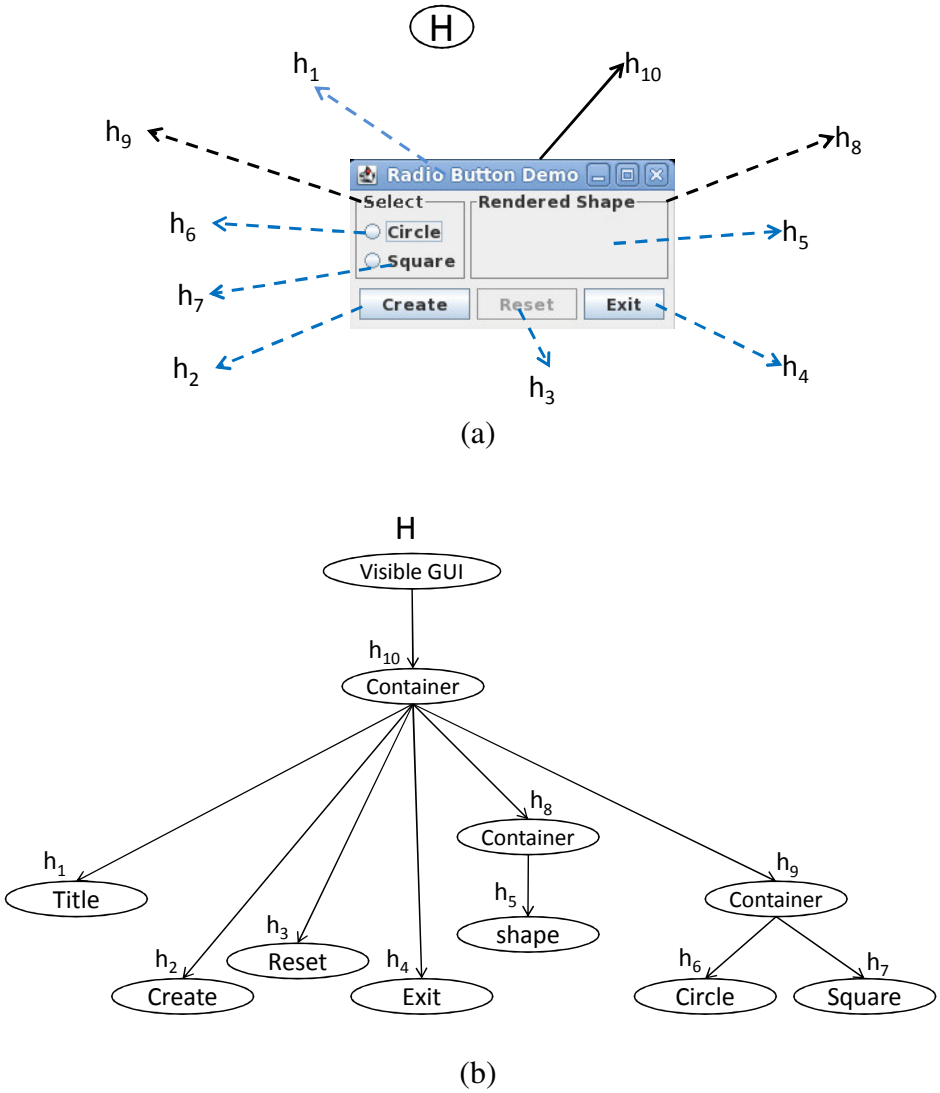
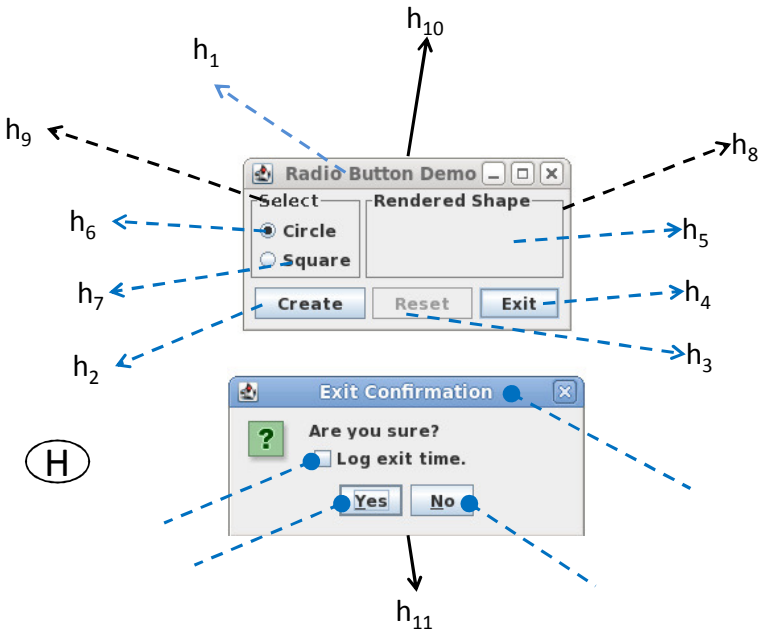
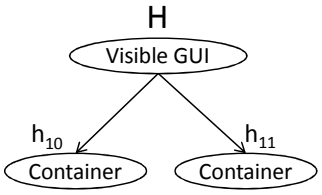


Figure 3.3: Signatures computed for the hierarchical GUI structure of `RadioButton Demo`'s initial state. It contains one GUI window. (a) Visible GUI window with signatures of widgets and containers. (b) Tree showing hierarchical nature of GUI windows with signature of widgets and containers.



(a)



(b)

Figure 3.4: Signatures computed for the hierarchical GUI structure of `RadioButton Demo` after executing the `Exit` button. It contains two GUI window. (a) Visible GUI windows with signatures of widgets and containers. (b) Tree showing hierarchical nature of GUI windows with signature of widgets and containers.

this GUI window, with the corresponding signatures. In these figures, h_1 – h_7 are signatures computed for the leaf-level widgets. Intermediate GUI containers are shown with signatures h_8 and h_9. The sole GUI window has the signature h_{10}. The final signature of the complete (with only one GUI window) GUI state is H.

Figure 3.4 (a) and (b) shows the computation of signature for `RadioButton Demo` when the `Exit` dialog box has been created. In this state, the GUI has two windows. Figure 3.4 (a) shows the two windows with corresponding signatures. In this figure, h_{10} and h_{11} are the signatures of the GUI windows. The final signature of the GUI state is H, obtained by combining h_{10} and h_{11}.

This section shows a method to compute signatures for a hierarchical GUI that is visible during the execution of a testcase. The next section shows how the signatures can be used to determine the presence of a new GUI state, during the execution of a testcase.

3.4.3 New GUI states

Hierarchical signatures provide a *fast* and *simple* method to identify the presence of new GUI states reached by the GUI, when comparing the executions of a baseline and target testsuite. The following three steps are followed:

Step 1: First, the baseline testsuite is executed. After executing an event from a testcase in this testsuite, the procedure `sigUpdate` from Figure 3.5, is executed as:

$$\texttt{sigUpdate}(\texttt{baselineSignature}, G) \tag{3.4}$$

where `baselineSignature` is a table for storing all signatures obtained from the baseline testsuite and `G` is the complete state of the GUI after executing the event.

The procedure `sigUpdate` accepts as input 1) G, a set of GUI windows of the application, that are visible after executing an event from the testcase and 2) `signature`, a reference to the

```
PROCEDURE sigUpdate
IN: signature = { signatures table }
IN: G = { GUI windows}
H = 0
foreach g ∈ G                                  1
   H+ = hash(g)                                2
if (!exists(signature, H))                     3
   insert(signature, H)                        4

PROCEDURE hash
IN: C: widget or container
OUT: hash value of C
h = 0
if isWidget(C)                                 1
   A = attributeList(C)                        2
   foreach a ∈ A                               3
      v = value(a)                             4
      h+ = CRC32(a)                            5
      h+ = CRC32(v)                            6
else
   G = componentList(C)                        7
   foreach g ∈ G                               8
      h+ = hash(g)                             9
return h
```

```
PROCEDURE listNew
IN: B: signature table of baseline
IN: T: signature table of target
foreach t ∈ T                                  1
   if (!exists(B, t))                          2
   print new state t                           3

Algorithm run-times:
   O(log m) - exists, insert
   O(n log m) - sigUpdate
   O(1) - hash
where:
   n - # testcases in a testsuite
   m - # unique signatures in signature table

Algorithm run-times:
   O(n log m) - listNew
where:
   n - # testcases in target testsuite
   m - # testcases in baseline testsuite
```

Figure 3.5: Procedures to detect if an application reached new GUI states after executing a target testsuite. Procedure `sigUpdate` computes a signature of the GUI state after the execution of an event. Procedure `listNew` compares the signatures of a target testsuite with a baseline testsuite.

table where the computed signatures will be stored by the procedure. In lines $1-2$, the `hash` procedure is invoked for each GUI window from the set G. In line $3-4$, the computed signature of the complete visible GUI state is inserted into the table, if the signature was not already present.

The procedure `hash` accepts as input C, a GUI container (such as a GUI window) or leaf-level widget. It returns the computed signature of that container or widget. The variable `h` stores the running value of the signature for C. Lines $1-6$ process C if it is a leaf-level widget. In line 2, a list of attributes of the widget is obtained. For each attribute a, lines $4-6$ obtain the value associated with the attribute and update `h` with the signature of the attribute and value. The `CRC32` algorithm is used for computing the signature. Lines $7-9$ process C if it were a GUI container

such as a window. For every sub-container, g in C, its signature is computed and updated into `h`.

During execution of the baseline testsuite, the procedure `sigUpdate` computes the signature of the complete visible state of the GUI after execution of each event in a testcase. It is then stored in a table called `baselineSignature`. When execution of the baseline testsuite completes, the table `baselineSignature` contains signatures of the GUI state visible after execution of each event of every testcase.

Step 2: Second, the target testsuite is executed in a similar manner as the baseline testsuite. In this case, the procedure `sigUpdate` is executed as:

$$\texttt{sigUpdate}(\texttt{targetSignature}, G) \tag{3.5}$$

where `targetSignature` is a table for storing signatures of the visible GUI state after executing each event of every testcase from the target testsuite.

Step 3: Finally, after both baseline and target testsuites have completed execution, an analysis is made to determine if new GUI states were reached during the execution of the target testsuite. This is done by invoking the procedure `listNew` in Figure 3.5. The procedure `listNew` accepts as input, two tables, the first containing a set of all signatures from the baseline testsuite and the second containing a list of all signatures from the target testsuite. It is invoked as:

$$\texttt{listNew}(\texttt{baselineSignature}, \texttt{targetSignature}) \tag{3.6}$$

In lines $1 - 3$, if a signature from the target table is not found in the baseline table, then it is determined to be a new GUI state.

The procedures `sigUpdate` and `listNew` provide a fast and simple method to enumerate GUI states that were encountered using a target testsuite, but were not encountered using the baseline testsuite. These new states are ones that the target testsuite drove the GUI into, but the baseline testsuite was not able to. Procedure `sigUpdate` has complexity $O(n\texttt{log}m)$, where n is

the number of events in the given testsuite and m is the number of unique GUI signatures that are encountered by the testsuite. The procedure `listNew` similarly has a complexity of $O(n\log m)$, where n and m are the number of unique GUI signatures encountered in the baseline and target testsuite respectively.

This section introduces hierarchical signatures, to identify new GUI states that are encountered while executing ECIG-based testsuites. Identification of new GUI states will show that the ECIG-based testsuites are able to drive the application into GUI states that was not possible using a comparable method.

3.5 Example

This section shows a simple example of an ECIG model-based *long* testcase that is *useful* in detecting a software defect. The example:

1. finds a new GUI state
2. shows why the ECIG model chose to create this testcase
3. shows how a software defect in the event handler could manifest as a defect in the GUI state.

1) Find new GUI state: A length-3 testcase based on ECIG model was created for the Java application JEdit (see Chapter 5). The testcase has the following event sequence:

$$Search \longrightarrow Find \longrightarrow \underline{Auto\,Wrap} \longrightarrow \underline{Regular\ Expressions} \longrightarrow Search \longrightarrow \underline{Whole\,Word}$$

The underlined events form the length-3 event sequence generated by the model, as $Auto\,Wrap \longrightarrow Regular\ Expression \longrightarrow Whole\,Word$. The intermediate events are filled in so that underlined events are reachable from the previous underlined event. This is considered as a *long* testcase,

since typical EFG (or EIG) based testcases are of length-2. It was executed immediately after launching JEdit, that is, from a clean known state of the application.

Figure 3.6 shows visible states of JEdit taken after executing events *Auto Wrap* (as e_1, at bottom-left), *Regular Expressions* (as e_2, at bottom-right) and *Whole Word* (as e_3, at top-right). After executing the first event, the checkbox *Auto Wrap* becomes checked. It is labelled with a large 1. After executing the second event, the checkbox *Regular Expressions*, labelled with a 2, becomes checked. After executing the third event, two checkboxes *Whole Word*, labelled with 3 and 4, become checked. Altogether 4 checkboxes, $1 - 4$, are checked. The final GUI state, achieved after executing the third event, is a GUI state that cannot be reached by executing any length-2 event. For example, executing the length-2 testcase, *Auto Wrap* $\longrightarrow$ *Regular Expressions* would result in only two checkboxes, 1 and 2, being checked. Executing *Auto Wrap* $\longrightarrow$ *Whole World* would result in only checkboxes 1, 3 and 4 being checked. Hence, in this example, the ECIG-based length-3 testcase is able to drive the application to a GUI state that is not possible with a length-2 testcase.

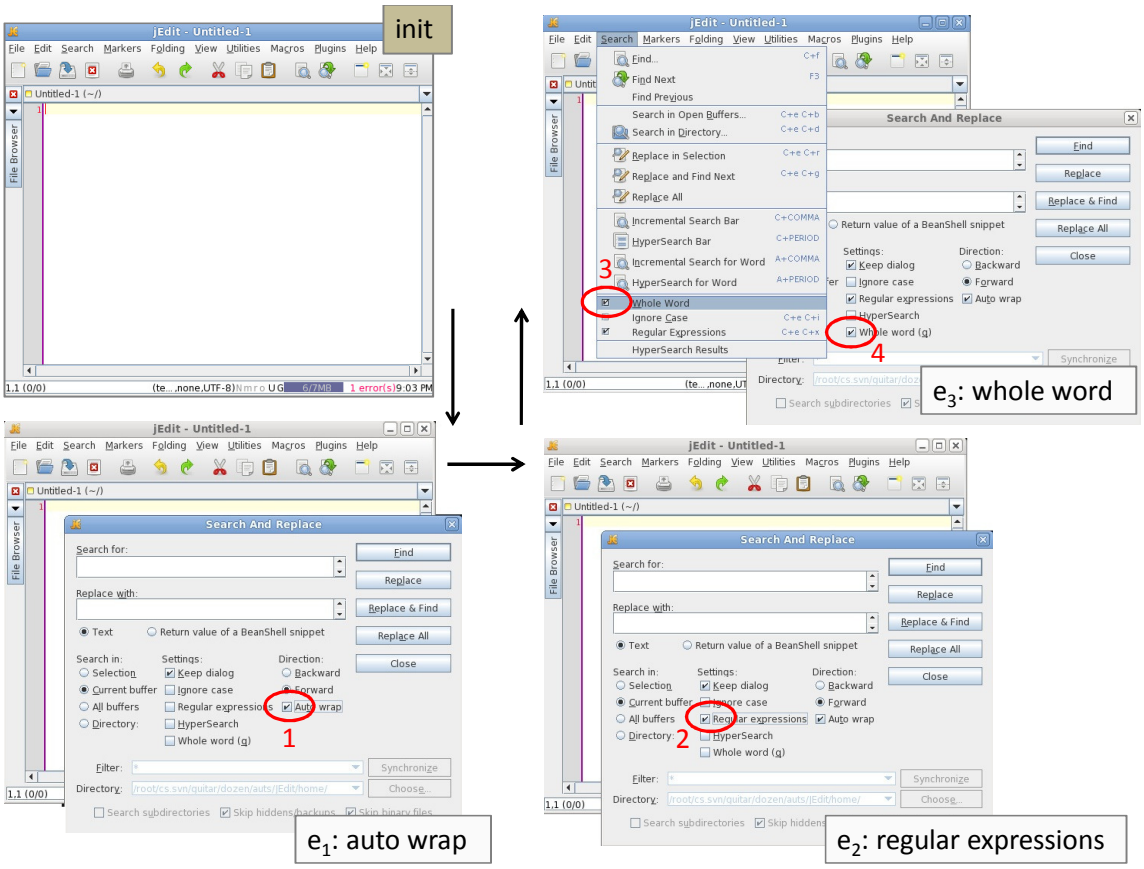


Figure 3.6: ECIG-based length-3 testcase – *Auto Wrap* $\longrightarrow$ *Regular Expressions* $\longrightarrow$ *Whole Word* – executed on JEdit. Resulting GUI states are shown – initial state (top-left), after executing *Auto Wrap* (bottom-left), after executing *Regular Expressions* (bottom-right), after executing *Whole Word* (top-right). Four checkboxes, labelled $1-4$, become checked as a result of the testcase execution. The final state, is a *new* GUI state, reached by the length-3 testcase. It is not reachable using existing length-2 testcase.

2) Why this testcase?: The ECIG model selected the two event sequences *Auto Wrap* $\longrightarrow$ *Regular Expressions* and *Regular Expressions* $\longrightarrow$ *Whole Word*. These two event sequences are edges in the ECIG.

Q: Why were these two edges (or event sequences) present in the ECIG?

A: Because the preceding event influences the succeeding event at the program-code level.

As an example, consider the event sequence (or ECIG edge) *Regular Expressions* $\longrightarrow$ *Whole Word*. Table 3.1 shows part of the event handler for the event `Whole Word`, in JEdit's codebase. During the construction of the ECIG for JEdit, the event sequences (a) *Regular Expressions* $\longrightarrow$ *Whole Word* and (b) *Whole Word* were independently executed on the initial state of JEdit, and the code coverage was recorded after executing each event, in each instance. The figure shows part of the lines of code that were executed by the event handler for *Whole Word*. The highlighted (with $*$) lines of code were executed in instance (a) but not in instance (b). This is because execution of the event *Regular Expressions* before the execution of the event *Whole Word* influences the latter. In fact, the function `handleSearchSettingsChanged` is invoked by `Whole Word`, only if the dialog box `Search and Replace` is created before its execution. Since `Regular Expressions` first creates the dialog box, the execution of `Whole Word` is affected. Hence, `Regular Expressions` influences (or interacts with) `Whole Word` at the program-code level. This merits inclusion of the edge `Regular Expressions` $\longrightarrow$ `Whole Word` in the ECIG for JEdit.

3) Detect software defect: The testcase in this example can detect a defect. Consider the seeded fault shown in Table 3.2. This is a modified version of the code from Table 3.1. The fault is seeded in line 249, where the `!` is removed in the defective code. This line of code will be executed by any testcase containing the event sequence `Regular Expressions` $\longrightarrow$ `Whole Word`. When executed by the testcase in this example, the resulting final GUI state is shown in Figure 3.7. In this figure, the checkbox labelled 4 is not checked, as it should be (from Figure 3.6, top-right).

JEdit: SearchDialog.java

```
247         public void handleSearchSettingsChanged(EBMessage msg)
248         {
249*          if (!saving)
250*          load();
251*        }

924         private void load()
925         {
926*          wholeWord.setSelected(SearchAndReplace.getWholeWord());
927*          ignoreCase.setSelected(SearchAndReplace.getIgnoreCase());
...*          ...
984*          keepDialog.setSelected(jEdit.getBooleanProperty(
985*                                 "search.keepDialog.toggle"));
986*        }
```

Table 3.1: Partial code from event handler of `Whole Word` (`Edit` menu) checkbox event.

This defective GUI state can be detected by comparing the defective state with an expected (oracle) state using the hierarchical signature method described in the previous section.

This section shows the example of an ECIG-based long testcase that drives the GUI of the application into a new state and proves useful in detecting a fault.

JEdit: SearchDialog.java

```
247         public void handleSearchSettingsChanged(EBMessage msg)
248         {
249*          if ( saving)
250           load();
251         }

924         private void load()
925         {
926           wholeWord.setSelected(SearchAndReplace.getWholeWord());
927           ignoreCase.setSelected(SearchAndReplace.getIgnoreCase());
...           ...
984           keepDialog.setSelected(jEdit.getBooleanProperty(
985                                  "search.keepDialog.toggle"));
986         }
```

Table 3.2: Fault is seeded at line 249. The ! has been removed.

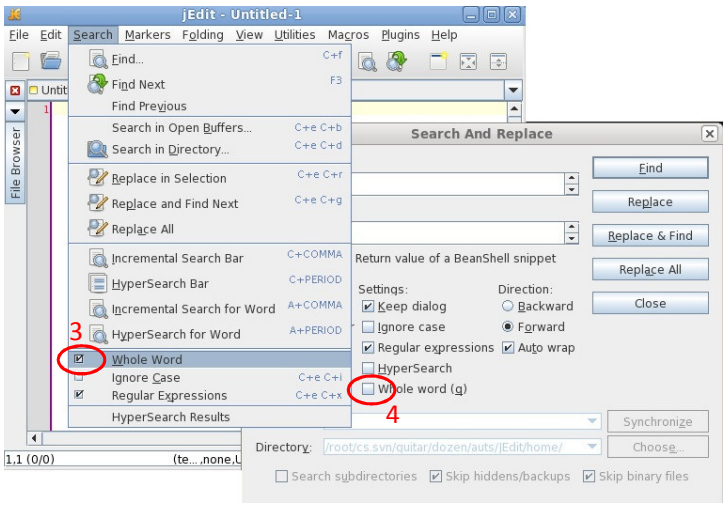


Figure 3.7: Software defect being manifested as a GUI defect. The checkbox 4 is not checked because of the defect.

Chapter 4

Tools and testbeds

This chapter describes components required for an empirical evaluation of ECIG-based test-cases. The evaluation requires software components, a workflow that joins these components and a testbed where the software is executed. Open source software components were used for the evaluation. They were integrated into a production quality, fully automated testbed.

4.1 Tools

Two open source tools, Cobertura and GUITAR, were used in the evaluation. Gephi[1], an open source graph management tool was used for rendering the EFG, EIG and ECIG for visualization.

4.1.1 Cobertura

A code-coverage tool, Cobertura[2], is used for recording the lines of code executed in response to events. For each uncommented line of program code, Cobertura records the number of times the line was executed.

[1] gephi.github.io
[2] cobertura.sourceforge.net

4.1.2 GUITAR

GUITAR[3], a GUI Testing frAmewoRk, is a tool for modelling and testing GUI-based applications. The flexible *plugin*-based architecture [35] of GUITAR 1) enables developers to implement their own new GUI models and 2) generate testcases based on the new models. The ECIG has been implemented as a GUITAR plugin. It has been used in the evaluation in this study.

At a high level GUITAR contains four component tools – GUI Ripper, Graph Converter, Test Case Generator and Replayer. The GUI Ripper and Replayer are components that contain platform-dependent hooks that interact with the platform's GUI APIs. The Graph Converter and Test Case Generator are platform-independent components that work with abstract GUI models and do not need to interact with the platform. All components of GUITAR operate in a fully automated manner and integrate well with software testing harnesses. GUITAR works with 6 GUI platforms – Java (JFC, SWT), Web, iOS, Android and UNO (Open Office).

The GUI Ripper is a tool that extracts structural and semantic GUI information from the runtime execution state of a GUI-based application under test. The GUI Ripper 1) automatically launches the application, 2) identifies its top-level GUI windows, 3) extracts structural and semantic information about each window and its constituent widgets, 4) invokes executable events on widgets, 5) detects new GUI windows that may be created as a result of the executed events and repeats steps $3 - 5$ on any newly created GUI windows. After repeating these steps, the GUI Ripper produces a GUI Tree, modelling structural information about the GUI of the application under test. The GUI Ripper contains platform-dependent components that invoke platform's GUI APIs to extract GUI information from the runtime application state and also execute events on the GUI.

The Graph Converter extracts semantic information about GUI widgets and events from a GUI Tree and models it as a graph, such as an EFG (see Section 2.5). It works on GUI models and does not require any interaction with the platform. Other semantic models, such as EIG, can be

[3] guitar.sourceforge.net

extracted and modelled by implementing an appropriate plugin for the Graph Converter.

The Test Case Generator is a platform-independent component that generates executable test-cases based on a GUI model such as EFG. The Test Case Generator traverses the GUI model and produces testcases which are sequences of events executable along the traversal path. Plugins for the Test Case Generator can be implemented by the tester to realize different testcase generation logic such as a random walk and all-event coverage.

The Replayer is a platform-dependent component that executes testcases on the application under test. It invokes platform-specific GUI APIs to identify widgets on which GUI events from the testcase needs to be executed and executes the event. It records whether the application encountered an unhandled exception or was simply terminated as a result of an exception, while the testcase was being executed. It also records the resulting GUI states after executing each event in the testcase. The GUI state recorded by the Replayer shows how the GUI was explored by the testcase while it was being executed. These GUI state are provided as input to the procedure `sigUpdate`, as described in Section 3.4. In this study, the usefulness of a testcase is measured by its ability to explore new GUI states.

The four GUITAR components described above can be integrated into test harnesses, such as Jenkins[4], to produce automated workflows for testing the GUI of GUI-based applications.

4.2 Workflow

GUITAR is designed using independent components that can be linked to tailor a suitable workflow. In this study, two GUITAR workflows have been used. The *standard workflow* was used to execute the *seed* testsuite. The *ECIG extension* was used to execute ECIG-based long testcases.

[4]jenkins-ci.org/

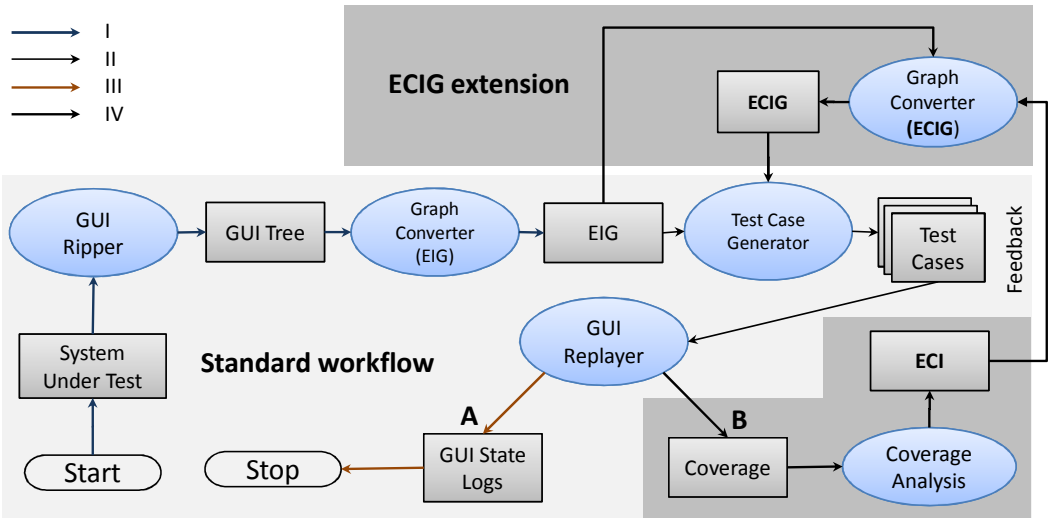


Figure 4.1: Typical GUITAR workflows. The *Standard workflow* enables common GUI testing activities. An extension of the standard workflow is developed as the *ECIG extension*. I–IV represent 4 distinct segments of the workflows.

4.2.1 Standard workflow

Figure 4.1 shows a schematic diagram of typical workflows that use GUITAR to test the GUI of a GUI-based application. This figure shows two workflows – *standard workflow* and *ECIG extension*.

The standard workflow follows the segments I–III shown in Table 4.1, which corresponds to Figure 4.1.. It is a basic testing strategy that can be used for testing the GUI of GUI-based applications. In this workflow, the four GUITAR components – GUI Ripper, Graph Converter (with *EIGConverter plugin)*, Test Case Generator (with *SequenceLength Generator plugin)*, Replayer – are shown in oval, with the testing artifacts they consume and produce, in rectangles.

In Segment I, the GUI Ripper takes the application under test and reverse engineers it to produce a GUI model, the GUI Tree. A Graph Converter plugin consumes the GUI Tree to produces a graph. The *EIGConverter* is used, hence an EIG is produced. The EIG models interactions between events that are not structural GUI events.

I	*Start* $\longrightarrow$ Application Under Test $\longrightarrow$ *GUI Ripper* $\longrightarrow$ GUI Tree $\longrightarrow$ *EIG Graph Converter* $\longrightarrow$ EIG $\longrightarrow$
II	*SequenceLength Test Case Generator* $\longrightarrow$ Testcases $\longrightarrow$ *Replayer* $\longrightarrow$
III	Logs $\longrightarrow$ *Stop*

Table 4.1: Standard workflow path corresponding to Figure 4.1. Each segment of the path is labeled I–III. The workflow is: I $\longrightarrow$ II $\longrightarrow$ III.

I $\longrightarrow$ II $\longrightarrow$	
IV	Coverage $\longrightarrow$ *Coverage Analysis* $\longrightarrow$ ECI $\longrightarrow$ *ECIG Graph Converter* $\longrightarrow$ ECIG $\longrightarrow$
II $\longrightarrow$ III	

Table 4.2: ECIG extension corresponding to Figure 4.1. The workflow is I $\longrightarrow$ II $\longrightarrow$ IV $\longrightarrow$ II $\longrightarrow$ III.

In Segment II, a Test Case Generator plugin, the *SequenceLength Generator*, traverses the EIG to generate a testsuite that contains all possible testcases of a specified length. For example, when the specified length is 2, the testsuite will contain testcases that together cover all *event pairs* interactions from the EIG. The Replayer executes all testcases on the application under test. It records the resulting GUI state of the application, after executing each event.

In Segment III, logs produced by the Replayer are automatically analyzed, shown with the edge **A**, to detect exceptions and crashes. The GUI state recorded by the replayer in the standard workflow can be used as the baseline against which the ECIG-based testsuite is compared. The GUI states can be provided as input to the procedure `sigUpdate` and stored in the table `baselineSignatures`.

The entire workflow is production quality and is fully automated, requiring no human intervention.

4.2.2 ECIG extension

This section describes the steps to reverse engineer a GUI model of a GUI-based application. These include identifying ECI relations, creating an ECIG and testing the application based on its

ECIG relations. The steps build upon the *Standard Workflow* described in the previous section. The *ECIG extension* workflow, in Figure 4.1, consumes test artifacts from the standard workflow.

In this workflow segments I and II of the standard workflow are first executed (see Table 4.2). As a result, the GUI Tree and EIG of the application under test is produced. The SequenceLength Generator produces an EIG-based SequenceLength-2 testsuite that contains all possible length-2 testcases. This testsuite serves as a seed testsuite for determining ECI relations between events. The seed testsuite is executed by the replayer, producing code coverage information. Segment IV contains the core logic for ECI detection and ECI-based testing. It consumes the code coverage report from the seed testsuite. This segment consists of *Coverage Analysis* and *ECIG Graph Converter* (see Table 4.2).

Coverage analysis: The SequenceLength-2 seed testsuite and its corresponding code coverage are analyzed to determined ECI relations amongst the events of the GUI. The code coverage obtained from the code coverage tool, Cobertura, is converted to use the **Approach L1 (unique-lines-of-code)** method of code coverage.

Table 4.3 shows a representative example of a length-2 EIG-based testsuite and its code coverage files. Each testcase is named with the two event names of the testcase - $first$, $second$ - as $e_{first}\_e_{second}$. The code coverage files are named in a similar manner. The folder for the coverage files is named as $e_{first}\_e_{second}$. The coverage files placed in it for the two events, $first$ and $second$, are named $first$.xml and $second$.xml respectively.

Procedure `ECI` in Figure 4.2 is then applied to the seed testsuite and its code coverage. Inputs to the procedure are the list of testcase names T and the corresponding list of code coverage records C, which is a 2-dimensional record accessed by the ($first$, $second$) event names of a testcase. The ECI relation is returned in the output table ECI.

In order to determine if an event e interacts with another event the *native* code coverage for e, when it is executed in isolation immediately after launching the application, is required. This

#	Testcase events $first, second$	Testcase name (T) (e_{first}-e_{second})	Per-event code coverage (C) ($first$.xml, $second$.xml)
1	a, b	e_a-e_b	a.xml, b.xml
2	a, c	e_a-e_c	a.xml, c.xml
3	b,c	e_b-e_c	b.xml, c.xml
4	b,d	e_b-e_d	b.xml, d.xml
5	c,d	e_c-e_d	c.xml, d.xml
6	c,a	e_c-e_a	c.xml, a.xml
7	d,a	e_d-e_a	d.xml, a.xml
8	d,b	e_d-e_b	d.xml, b.xml

Table 4.3: A representative SequenceLength-2 testsuite containing 8 testcase, with corresponding code coverage files for each testcase. Code coverage is collected after executing each event of every testcase.

native code coverage for all events are stored in a lookup table CC in lines $2-4$. Here, each code coverage record is checked to see if its $first$ event has been considered for its *native* code coverage. If not, then its code coverage is stored in CC as the *native* code coverage for the event $first$. At the end of this step CC contains the *native* code coverage for every event in the GUI.

Lines $5-8$ selects a testcase, t, and determines if its $first$ event interacts with its $second$ event. This is done by comparing the code coverage of $second$ for t with the *native* code coverage for $second$. If the two code coverages do not match then it indicates that $first$ might have interacted with $second$ to alter its code coverage in t. This means that event $first$ interacts with event $second$ and an edge $first \longrightarrow second$ is added to the set ECI.

For example, in Table 4.3, to determine if the interaction $e_b \longrightarrow e_d$ exists, two coverage files for the event e_d needs to be compared. The *native* coverage for e_d, d.xml, is stored in $CC[e_d]$ from testcase 7. To determine the interaction, the coverage for e_d, d.xml, from testcase 4 is compared with the *native* coverage.

PROCEDURE ECI
IN: $T = \{t \mid t \in$ EIG based SequenceLength-2 testsuite; t is like $first\_second\ \}$
IN: $C = \{c(first, second) \mid c =$ code coverage record for testcase $first\_second)\}$
OUT: $ECI = \Phi$

```
CC = Φ                                                                          1
# Compute native code coverage for each event in the EIG
foreach c ∈ C                                                                   2
    if (CC[c.first] == Φ)                                                       3
        CC[c.first] = c                                                         4
# Determine ECI interactions
foreach t ∈ T                                                                   5
    (first, second) = (t.first, t.second)                                       6
    if (C[first, second].second.coverage ≠ CC[second].first.coverage)           7
        ECI = ECI ∪ (first, second)                                             8
```

PROCEDURE ECIG
IN: $EIG =< V_1, E_1, I_1 >$
IN: $ECI =\{e \mid e \in ECI\}$
OUT: $ECIG =< V_2, E_2, I_2 >$

```
V2 = V1; E2 = Φ; I2 = I1                                                        1
foreach e ∈ E1                                                                  2
    E2 = E2 ∪ e                                                                 3
```

Figure 4.2: Procedure ECI computes ECI relations between event pairs of a GUI. Procedure ECIG produces an ECIG-based on the ECI relation.

ECIG Graph Converter The ECI information produced by the *Coverage Analysis* step produces a list of event pairs which exhibit the ECI relation. This step generates the ECIG-based on the ECI relations. The procedure ECIG in Figure 4.2 is executed to generate the Event-Code Interaction Graph. This procedure takes the EIG for the application and the ECI relation from the procedure ECI as inputs. It produces a graph $ECIG$ as the output. This graph has the same set of vertices as the EIG ($V_2 = V_1$) and the same set of initial events ($I_2 = I_1$) (Line 1). Its edges are the set ECI (Lines 2-3).

The procedures ECI and ECIG produce the ECIG as the output of Segment IV, in the ECIG extension of Figure 4.1 and Table 4.2. Thereafter, Segment II and III of the standard workflow are again executed. In Segment II, the Test Case Generator consumes the ECIG from Segment

IV and generates SequenceLength-n testcases based on the ECIG. These testcases are executed by the Replayer, which records the GUI states of the application after executing each event of every testcase.

In Segment III, logs recorded by the Replayer are analyzed to determine if the application crashed or an exception was encountered during replay. The procedure `sigUpdate` is executed on the GUI state recorded by the Replayer, to create the table `targetSignatures`. This table stores all GUI states visited by the target ECIG-based testsuite.

The GUI states reached by the target ECIG-based testsuite is then compared with the GUI states reached by the baseline EIG-based length-2 testsuite. This is done by invoking the procedure `listNew` with:

$$\texttt{listNew}(\texttt{baselineSignatures}, \texttt{targetSignatures}) \tag{4.1}$$

to determine if the ECIG-based testsuite visited *new* GUI states.

4.2.3 Integration

It was shown by Nguyen [35], that GUITAR may be integrated into an automated software testing workflows. In a similar manner, the *Standard workflow* and *ECIG extension* described in the previous sections may readily be integrated into generic software testing workflows.

Regression testing: During the process of software development, program-code changes are frequently made to the software. It is necessary to execute testcases on the changed software to determine if the behaviour of the new version of the software has regressed from the previous version. The *Standard workflow* and *ECIG extension* may be integrated into such regression testing workflows. This is done by executing the ECIG-based testcases during each regression testing

cycle.

Incremental model updates: During the process of developing a software, the behaviour, characteristics and appearance of the software may gradually change. These changes necessitate updates to the EIG and ECIG models, so that testcases may be re-created, based on the new behaviour of the application. Using the standard workflow, updates to the EIG requires executing the GUI Ripper. Using the ECIG extension, updates to the ECIG require 1) re-creating the EIG-based baseline testcases 2) coverage analysis and 3) creation of the ECIG. These steps being resource intensive, it is possible to make incremental updates to the EIG and ECIG models instead.

Based on known updates to the GUI of the application. The GUI Ripper may be modified to detect updates to the GUI compared to the previous version. The changes may be used to 1) make incremental updates the EIG 2) re-create EIG-based baseline testcase corresponding to the updated GUI 3) execute the re-created baseline testcases and 4) analyse coverage data for the re-created testcases to determine new ECI relations and incrementally update the ECIG. These steps outline a method for incremental updates to the ECIG model. They are not further explored in this study.

4.2.4 Contributions

The *standard workflow* and *ECIG-extension* stitch together the GUITAR tools to evaluate the usefulness of ECIG-based testcases. The workflows determine ECI relations, produces the ECIG, ECIG-based testcases and new GUI states. These artifacts are contributions 2 and 3 identified in Chapter 1.

Contribution 2: The ECIG is the second contribution of this study. Code-coverage analysis of the seed EIG-based SequenceLength-2 testcases identifies interacting events. The interacting events are combined into a model, the ECIG. Since the ECIG contains events that are known to interact at the program-code level, combined execution of events from this graph is likely to test interacting program-code and reveal defects therein.

Contribution 3: Comparing the EIG and ECIG, it can be seen that edges of the ECIG are selected from the edges of the EIG. In fact, the ECIG is essentially obtained from the EIG by removing those edges that do not exhibit interactions at the program-code level. Hence, the ECIG is sparse compared to the EIG. The number of event-sequences of length n that can be generated from the ECIG is fewer, compared to the EIG. The testsuite size is smaller and is typically executable in a reasonable amount of time. ECIG-based SequenceLength-n testsuite is a method to generate long testcases that can target specific parts of the GUI.

4.3 Testbed

Experiments were executed on the testbed shown in Figure 4.3. In this figure, host machines – **M1**, **M2**, **M3** – are configured with 24-2.4GHz Intel CPU, 48GB RAM, Ubuntu OS with 3.13.0-58 kernel and Java 1.7.0_45-b18. The controller machine, **C**, is configured with 64-2.3GHz AMD Opteron CPU, 256GB RAM, Linux OS with 3.10.0-229.7.2 kernel. The controller machine is configured to execute 20 concurrent testcases on each host machine for a total of 60 concurrent testcases. To execute a testcase on the host machine (control flow), the controller machine 1) sets up the environment on the host 2) launches the application 3) executes a testcase 4) records coverage reports, test logs and GUI states 5) archives results 6) terminates the application and cleans up its persistent states. These steps ensure that all testcase are executed in an identical environment with identical initial states. Test results and artifacts are stored in the controller machine (data flow).

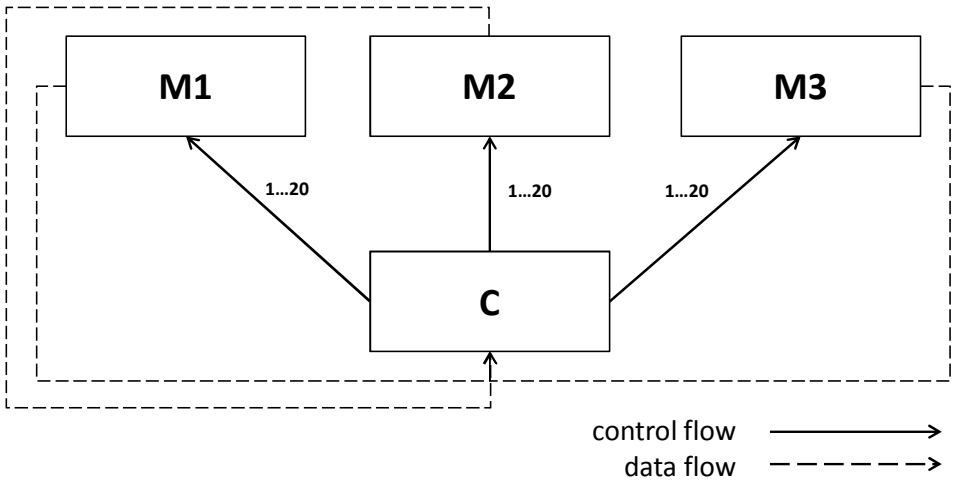


Figure 4.3: Testbed for executing testcases. Controller machine, **C**, executes 20 concurrent testcases on each worker machine, **M1**, **M2**, **M3**. Results are stored in the controller machine.

Chapter 5

Empirical evaluation

The Event-Code Interaction (ECI) model described in Chapter 3 is empirically evaluated in this study. Results of the empirical evaluation are presented in this chapter. The evaluation focuses on determining the existence of ECI relations in GUI applications and their usefulness in creating testcases for testing the applications. These results show that 1) ECI relations indeed exist in GUI-based applications 2) the methods developed in this study, make it is easy to identify such relations in real-world software testing environments 3) it is easy to create a model of the application, based on these relations 4) the model can generate long testcases that exercise the application in new ways and 5) these testcases are useful in finding defects in the application. Creation of long testcases for GUI applications has been challenging, using existing GUI models. The ECI model is shown to address this challenge with the ability to create long GUI testcases.

In this chapter, the term *state* indicates the complete visible GUI state of an application at a given instant. Typically, the state is referred to after an event of a testcase has been executed. In this evaluation, there are no references to non-GUI states, such as, memory states.

5.1 Experiment overview

Experiments were conducted on a set of 4 subject applications. Each application under test was subjected to the evaluation cycle described in *Standard Workflow* (see Section 4.2.1) and *ECIG Extension* (see Section 4.2.2). In all, four evaluation cycles were conducted, with one cycle being executed for each application under test. The cycles were executed in sequence, such that the data

Application	Abbreviation	Version	Year	LOC	Branch	Method	Class
ArgoUML	AU	0.34	1999	70,430	32,254	16,034	1,854
Buddi	BD	3.4.1.11	2006	155,960	43,670	25,684	2,751
JabRef	JA	2.10	2003	61,714	28,440	8,722	1,493
JEdit	JE	5.1.0	1998	67,761	38,845	9,881	1,288

Table 5.1: Properties of applications under test, that were used for evaluation.

for one cycle was collected and analyzed completely before the next cycle was started. Section 5.2 and Section 5.3 present results from the four cycles, consisting of the Standard Workflow and ECIG extension respectively. In these sections, results for all applications under test are collected and presented together.

5.1.1 Research questions

To guide the empirical evaluation of the ECIG-based testcases, the following research questions are designed.

RQ1: Do ECI relations exist in a GUI-based application?

RQ2: Can ECI relations be used to generate *long* testcases for a GUI-based application?

RQ3: Can ECIG-based testcases exercise the GUI of a GUI-based application in *useful* ways?

RQ4: Are ECIG-based testcases good for detecting software defects?

These questions guide the evaluation of the ECIG model and ECIG-based testcases. Results of the evaluation are analyzed to answer these questions in Section 5.3.

5.1.2 Applications under test

Four Java-based GUI applications were selected as applications-under-test, for this study. The applications are the following:

1. **ArgoUML:** An open-source[1] software design and engineering tool.
2. **Buddi:** An open-source[2] personal finance and budgeting software.
3. **JabRef:** An open-source[3] software for managing bibliography references.
4. **JEdit:** An open-source[4] text editor for computer programmers.

Characteristics of the applications are shown in Table 5.1. In this table, the column *Version* shows the version number of the application that was selected for evaluation. The latest version available at the time of evaluation was selected. The column *Year* shows the year when the application was first made available. All the applications have been present for a decade or more, indicating that an appreciable amount of code maturity would have been reached. The column *LOC* shows the number of lines of code in the application, column *Branch* shows the number of code branches, columns *Method* and *Class* show the number of Java methods and classes in the application. It can be seen that the applications are non-trivial in terms of code size and complexity.

5.1.3 Threats to validity

External validity: Threats to external validity are factors that hinder generalization of results from this study to other applications, systems and environment. Results presented in this study are based on the empirical evaluation of four Java-based GUI-centric application. Results collected from each application are consistent with the results from the other applications. However, owing to the nature of GUI applications, evaluation results with new applications may vary. In addition,

[1] www.tigris.org
[2] buddi.digitalcave.ca
[3] jabref.sourceforge.net
[4] www.jedit.org

results may vary if non-GUI centric applications are chosen for evaluation. In this study four Java-based, GUI-centric applications were evaluated in the Linux platform. Evaluation results may vary if applications from other GUI platforms such as Windows, Android or MacOS are used for evaluation.

Internal validity: Threats to internal validity are factors that may affect or bias results of evaluation. For example, the design and implementation of tools and techniques may suffer from shortcomings. During the process of reverse engineering the GUI of an application under test, event identifiers are automatically generated, for every identified event of the application. The event identifier for an event is an integer, that is created by hashing together properties of the corresponding widget. The hash function may suffer from hash collision. A hash collision will result in two distinct events of an application being assigned the same event identifier. This may result in incorrect execution of events from a testcase. The process of extracting events from the GUI of an application may be incomplete. This happens when the GUI Ripper fails to identify and extract a widget from the visible GUI of the application. In addition, the GUI Ripper may fail to access and extract some GUI properties of a widget. As a result, the EFG, EIG and ECIG models may contain partial information about the structure of the GUI. This may result in some events and testcases being abset from the corresponding EIG or ECIG testsuites. The hierarchical signature that is used for identification of GUI states may suffer from hash collision. A hash collision will result in two distinct GUI states being considered to be identical. This may result in inaccuracies in the count of GUI states reached by an application, when a testsuite is executed on it.

5.2 Standard workflow

This section reports evaluation results following the steps of the Standard Workflow outlined in Section 4.2.1. The Standard Workflow is used to create a *seed* testsuite, from which ECI relations are determined and the ECIG model is constructed. States visited by the applications during

Application	rip time	windows	widgets	code coverage
AU	4 min 32 sec	26	1,723	29.18%
BD	3 min 48 sec	20	1,077	5.49%
JA	6 min 45 sec	44	1,899	21.90%
JE	8 min 48 sec	18	910	23.38%

Table 5.2: GUI Ripper reverse engineers the GUI of an applications to create the GUI tree. The GUI tree contains structural information about the GUI.

execution of the seed testsuite form the baseline for comparing with the new states visited during the execution of the ECIG-based testsuite.

GUI Ripper: The GUI Ripper reverse engineers the GUI structure of the application under test from its visible run-time state. It launches the application, identifies its top-level windows, extracts widgets using platform-specific APIs, executes events on widgets to open new windows and continues the process until the complete application has been observed. The GUI Ripper produces the GUI tree, that represents structural information about the GUI of the application.

Table 5.2 shows the result from executing the GUI Ripper on each of the four applications under test. The table shows time to reverse engineer the GUI and create the GUI-tree (*rip time*), metrics of the application's GUI, that is, number of GUI windows that were discovered (*windows*), number of widgets that were extracted (*widgets*) and the code coverage achieved as a result of exercising the application (*code coverage*). It can be observed that a sizeable part of ArgoUML, JabRef and JEdit's code was exercised by the simple process of reverse engineering. The GUI tree is next used to create the EFG and EIG.

EFG Graph Converter: In this step semantic information about GUI events is extracted from the GUI tree obtained from the GUI Ripper. The semantic information is used to create an EFG model representing events and their relations. The EFG Graph Converter extracts all *follows* relations

Application	create time	# vertices	# edges
AU	2 sec	498	15,846
BD	1.6 sec	245	3,621
JA	2 sec	367	12,981
JE	1.6 sec	428	13,687

Table 5.3: EFG Graph Converter extracts *follows* relation from the GUI tree to create the EFG.

Length		AU	BD	JA	JE
$n = 2$	# testcases	15,846	3,621	12,981	13,687
	creation time	7 min 3 sec	1 min 56 sec	6 min 21 sec	6 min 43 sec
	execution time	13 hours	5 hours 25 min	20 hours	9 hours 2 min
$n = 3$	# testcases	678,628	92,548	521,445	447,464
	counting time	1 sec	2 sec	2.4 sec	2.2 sec
$n = 4$	# testcases	30,421,680	3,166,492	21,251,647	14,602,995
	counting time	3 sec	3 sec	6.4 sec	4.4 sec
$n = 5$	# testcases	1,377,291,937	121,486,655	865,333,263	474,185,546
	counting time	3 min 23 sec	21 sec	2 min 23 sec	1 min 52 sec
$n = 6$	# testcases	62,464,993,710	4,812,611,517	35,185,275,646	15,326,111,843
	counting time	2 hours 26 min	12 min 35 sec	1 hour 35 min	37 min 35 min
$n = 7$	# testcases	2,833,714,733,116	192,229,380,427	1,429,529,616,612	493,309,467,198
	counting time	149 hours 16 min	9 hours 7 min	68 hours 49 min	22 hours 16 min

Table 5.4: Testsuite creation time, execution time and counting time for SequenceLength-n testcase based on the EFG, where $n \in \{2, 3, 4, 5, 6, 7\}$. Counting time is the time taken to count the possible number of testcases, without actually generating them.

from the GUI tree. It creates a directed graph whose vertices are events from the GUI. Edges are added to the graph, between two events, when one event can be executed immediately after the other.

Table 5.3 shows properties of the EFG obtained for each of the four applications – time taken to create the EFG (*create time*) and number of vertices and edges in the EFG (*# vertices, # edges*). It can be seen, from the table, that the EFG is create very quickly. Figures 5.1 and 5.2 show the graph models – EFG, EIG, ECIG – for all the applications. The EFG, shown in the left column, models

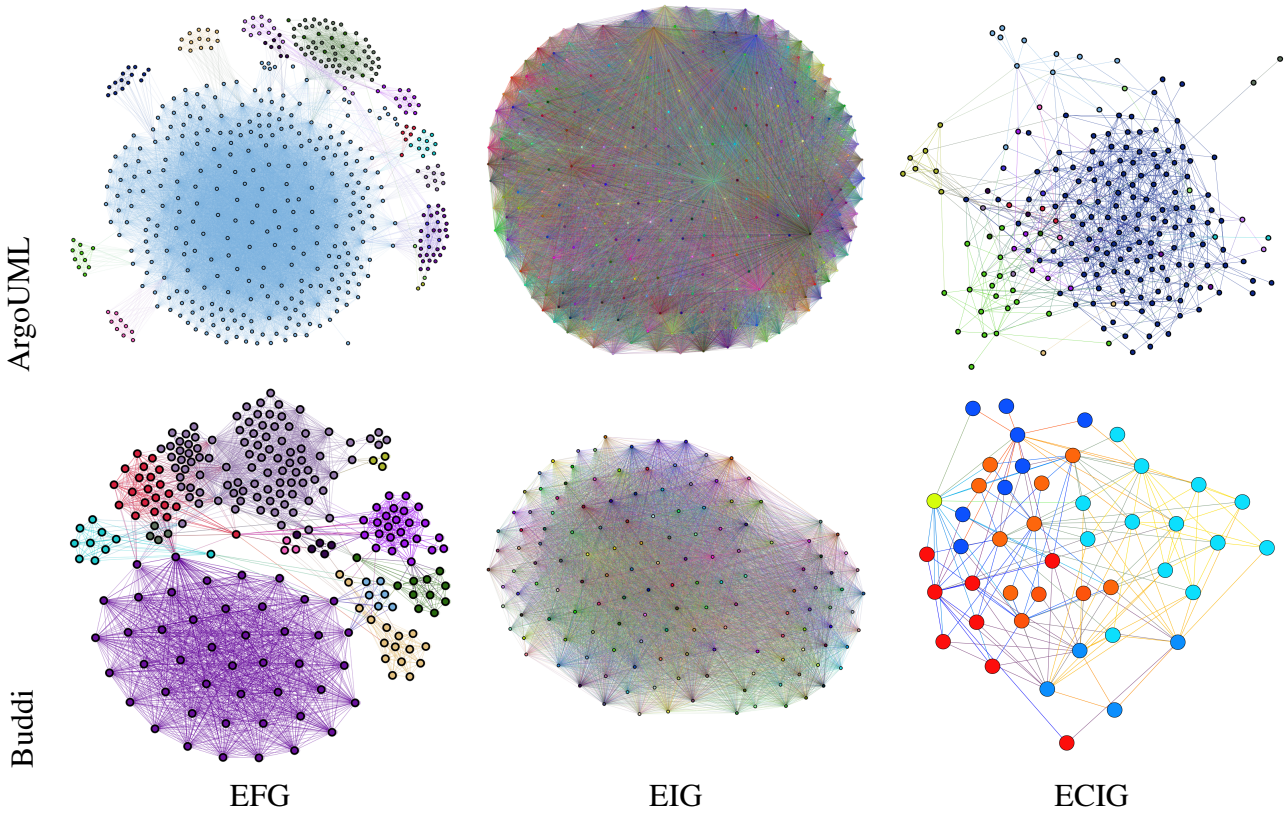


Figure 5.1: Visual representation of the EFG, EIG and ECIG of ArgoUML and Buddi. The EFG represents *follows* relation between all pairs of events. EIG represents *follows* relation between non-structural events. It contains fewer vertices and more edges. ECIG is a subset of the EIG, containing edges that interact at the program-code level.

the *follows* relation between an event e_2 that can be executed immediately after e_1. Clusters can be visually identified in the EFG that belong to GUI windows of the application.

EFG-based testcases: In model-based GUI testing methods proposed in earlier work [30], the EFG was used for generating testcases. Table 5.4 shows properties of SequenceLength-n testcases, for $n \in 2, 3, 4, 5, 6, 7$, based on the EFG. In this table, *Length* is the length of the testcase, in terms of number of events, # *testcases* is the number of testcases in a testsuite, *creation time* is the time taken to create a testsuite, *execution time* is the time to execute a testsuite, *counting time* is the time taken to count the possible number of testcases in a testsuite, without actually writing the testsuite

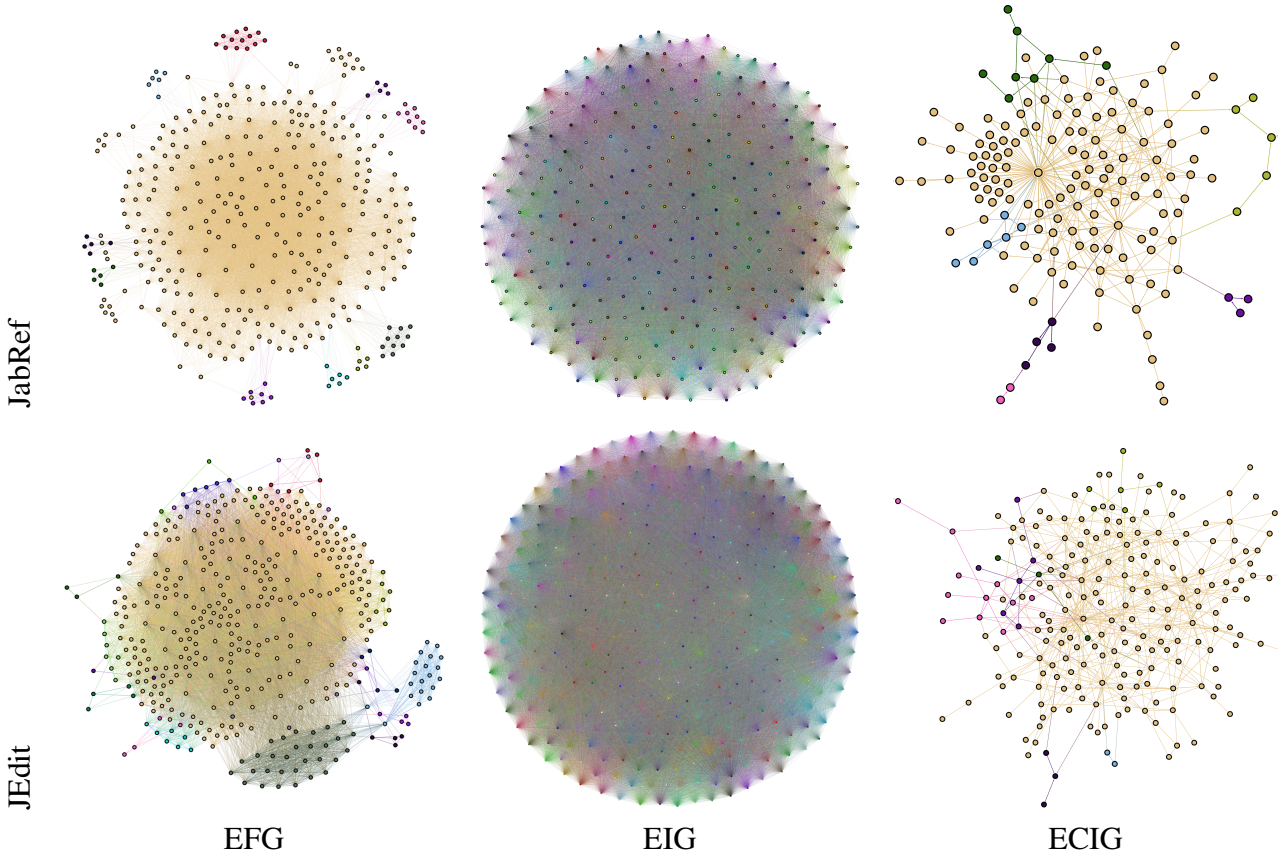


Figure 5.2: Visual representation of the EFG, EIG and ECIG of JabRef and JEdit. The EFG represents *follows* relation between all pairs of events. EIG represents *follows* relation between non-structural events. It contains fewer vertices and more edges. ECIG is a subset of the EIG, containing edges that interact at the program-code level.

into a data file. This table shows that the number of testcases grows rapidly with increasing length of the testcase. Execution of $n = 2$ testsuites consumed between 9 and 20 hours. In fact, beyond a length of 2 (or 3), the number of testcases is too large to create or execute, in a reasonable time. Hence, it is impractical to generated and execute testsuites containing long EFG-based testcases.

EIG Graph Converter: The EIG Graph Converter converts the EFG, obtained in the previous step, into an EIG. Table 5.5 shows properties of the EIG for each application. In this table, *create time* is the time taken by the Graph Converter to convert the EFG into an EIG, # *vertices* and # *edges*

Application	create time	# vertices	# edges
AU	2.2 sec	313	67,445
BD	1.6 sec	144	12,576
JA	2.1 sec	276	57,873
JE	3.4 sec	378	133,734

Table 5.5: EIG obtained from the EFG by removing structural events.

are the number of vertices and edges in the EIG. Vertices representing structural GUI events are not included in the EIG. An edge is present between two events (vertices), e_1 and e_2, if the events are not structural events and e_2 is executable along some path after executing e_1. As can be seen from this table, creation of the EIG is a fast process. In Figures 5.1 and 5.2, the middle column shows EIGs for the applications under test. Compared to the EFG, more edges are present in the EIG. This is because edges in the EIG include event pairs that do not necessarily follow each other immediately.

EIG-based testcases: Testcases created from the EIG, termed *abstract testcases*, are not immediately executable. This is because intermediate events need to be inserted so that one event is reachable from the previous. Intermediate events are inserted to convert the abstract testcases into *executable testcases*. Section 3.5 shows the example of an abstract testcase *Auto Wrap* $\longrightarrow$ *Regular Expressions* $\longrightarrow$ *Whole Word* that has been converted into an executable testcase by inserting intermediate events.

Table 5.6 shows properties of EIG-based testcases. Similar to EFG-based testcases, this table shows that the number of EIG-based testcases, in a testsuite, grows rapidly with increasing testcase length. Beyond a length of 2, the number of testcases is too large to create or execute in a reasonable time. Hence, similar to EFG-based testcases, it is practically impossible to create and execute testsuites containing long EIG-based testcases.

Seed EIG-based testsuite In this study, EIG-based SequenceLength-2 testsuites, were used as the

Length		AU	BD	JA	JE
$n = 2$ (seed)	# testcases	67,445	12,576	57,873	133,734
	creation time	1 hour 44 min	17 min 31 sec	1 hour 18 min	3 hour 24 min
	execution time	4 days 16 hours	1 day 2 hours	2 days 9 hours	9 days 23 hours
$n = 3$	# testcases	13,520,515	1,026,461	11,859,294	47,223,464
	counting time	5.22 sec	3 sec	4.7 sec	7.7 sec
$n = 4$	# testcases	2,696,435,575	82,069,590	2,450,183,082	16,673,935,504
	counting time	8 min 58 sec	15 sec	9 min 10 sec	37 min 51 sec
$n = 5$	# testcases	537,280,797,376	6,512,775,666	506,215,408,542	5,887,308,434,240
	counting time	22 hour 22 sec	16 min 16 sec	20 hour 49 min	218 hour 29 min

Table 5.6: Testsuite creation time, execution time and counting time for SequenceLength-n testcase based on the EIG, where $n \in \{2, 3, 4, 5\}$. Counting time is the time taken to count the possible number of testcases, without actually generating them.

App	# testcases	execution clock time	execution work	GUI state size (GB)	coverage size (GB)	execution log size (GB)	code coverage
AU	67,445	4 days 16 hours	224 days	91	608	1.4	30.60%
BD	12,576	1 day 2 hours	31 days	4.8	1,208	0.26	7.83%
JA	57,873	2 days 9 hours	135 days	143	2,789	0.73	25.55%
JE	133,734	9 days 23 hours	657 days	861	8,714	1.5	25.70%

Table 5.7: EIG-based SequenceLength-2 testcases is the seed testsuite. ECI relations are determined from the code coverage. GUI state forms the baseline, to check if another testsuite found new GUI states.

seed testsuite. This corresponds to the row $n = 2$ in Table 5.6. Execution of the seed testsuite by the Replayer is described in the next step.

Replayer: The Replayer executes the seed testsuite. The seed testsuite is the same testsuite from the $n = 2$ rows in Table 5.6. The Replayer executes each testcase after launching the application, that is, from a known state. Cobertura is used to record the lines of code executed after invoking each event of the testcase. The GUI state that is visible after executing each event is also recorded.

Table 5.7 shows execution results from executing the seed testsuite for each application. The

number of testcases in the seed testsuite (column # *testcases*) is equal to the number of edges in the EIG. Execution of the seed testsuite takes between 1 and 10 days, shown in column *execution clock time*. The column *execution work* shows the time taken to execute the testsuite multiplied by the number of concurrent execution threads. It estimates the time it would have taken to execute the testsuite using one execution thread. The collected artifacts for each testcase are: 1) GUI state 2) per-event code coverage data and 3) execution logs. The size of the collected artifacts are shown in the columns *GUI state size*, *coverage size* and *execution log size* respectively. The column *code coverage* shows the percentage of the application's source code exercised by the testsuite. The per-event code coverage data is used to determine ECI relations in the application. The recorded GUI state is used as the `baselineSignature` (see Section 3.4.3) to determine if ECIG-based testcases drive the GUI into new states.

This section describes the Standard Workflow. This workflow creates and executes EIG-based SequenceLength-2 testsuites (seed testsuites), on the applications under test. Artifacts from the execution of the seed testsuite are used in the ECIG extension, described in the next section.

5.3 ECIG extension

This section describes the ECIG Extension workflow outlined in Section 4.2.2. This workflow was used to 1) identify ECI relations in the applications under test 2) create the ECIG using the ECI relations 3) generate long, ECIG-based testcases 4) execute ECIG-based testcases and 5) verify that the testcases visited new GUI states in the applications.

Coverage Analysis: Coverage data obtained from executing the EIG-based seed testcases, from the Standard Workflow, were analyzed using the procedure `ECI` described in Section 4.2.2. The coverage data was first normalized, that is, all per-line positive coverage hit counts were reduced

Application	hit count normalization	ECI analysis time	# ECI vertices	# ECI edges
AU	32 hours	9 hours 22 min	205	799
BD	10 hours 38 min	1 hour 49 min	45	157
JA	25 hours 57 min	1 hour 31 min	168	355
JE	90 hours 10 min	14 hours 58 min	224	517

Table 5.8: Metrics for obtaining the ECI event pairs from coverage data, obtained by executing EIG-based seed testsuite.

to 1. This is in accordances with the **Approach L1 (unique-lines-of-code)**. The procedure `ECI`, from Figure 4.2, then produced a list of ECI event pairs from the coverage data. Table 5.8 shows metrics for producing the list of ECI event pairs, for all the applications. In this table, *hit count normalization* shows the time taken to normalize the hit counts, in the coverage data, to 1 and $ECI\ analysis\ time$ shows the time taken to execute procedure `ECI` on the coverage data. The column $\#\ ECI\ vertices$ shows the number of events that participated in the ECI relations. The column $\#\ ECI\ edges$ shows the number of ECI relations that were discovered, by the procedure `ECI`, in the coverage data. The number of ECI edges that were found for ArgoUML, Buddi, JabRef, JEdit are 799, 157, 355 and 517 respectively.

RQ1: Do ECI relations exist in a GUI-based application?

A: Column $\#ECI\ edges$ from Table 5.8 shows that ECI relations exist in a GUI-based application. Each ECI pair, represented in this column, denoted as $e_1 \longrightarrow e_2$, indicates that execution of event e_1 before e_2 influences subsequent execution of event e_2 at the program code level. Identification of these ECI relations in GUI applications is the crux of this study. ■

ECIG Graph Converter: The Graph Converter with ECIG plugin takes as input the EIG from the Standard Workflow and produces an ECIG. Edges included in the ECIG were determined by the

$n \longrightarrow$	3	4	5	6	7	8	9	10	11	
AU	319	797	1,074	1,801	2,225	2,247	2,546	2,522	2,789	
BD	65	183	368	711	1,230	2,014	2,966	4,347	5,799	
JA	125	242	421	698	1,123	1,694	2,338	2,475	2,298	
JE	234	442	717	1209	1,676	2,302	3,250	3,990	4,264	
$n \longrightarrow$	12	13	14	15	16	17	18	19	20	Total
AU	2,660	2,585	2,089	1,601	2,850	-	-	-	-	28,105
BD	5,753	5,466	3,086	-	-	-	-	-	-	31,988
JA	1,604	1,196	759	478	326	-	-	-	-	15,777
JE	3,530	2,667	1,764	1,264	800	436	184	51	7	28,787

Table 5.9: Count of testcases for ECIG-based SequenceLength-n testcase, where $n \in \{3, ..., 20\}$. Total testcases for each application is shown in *Total* column.

procedure `ECI` in the previous step. The number of vertices and edges in the ECIG are shown in $\#ECI\ vertices$ and $\#ECI\ edges$ in Table 5.8. In Figures 5.1 and 5.2, the right column shows a visual representation of the ECIG for the applications. Comparing the columns $\#ECI\ edges$ (Table 5.8) and $\#\ edges$ (Table 5.5), it can be seen that the ECIG is sparse, compared to the EIG. As a result, the number of testcases produced from the ECIG is fewer than that from the ECIG. This results in a smaller testsuite that can be executed within a reasonable time.

Testcase Generator: The ECIG can now be used to generate ECIG-based testcases. These testcase, like EIG-based testcases, are abstract in nature. Successive events in these testcase might not be directly reachable from the preceding event. Hence, intermediate events may need to be inserted, to convert the abstract testcases into executable testcases.

Table 5.9 shows the number of SequenceLength-n ECIG-based testcases of different lengths, from 3 to 20. In this table, a column shows the number of testcases of a given length for different applications. It can be seen that the ECIG generates testcases with as many as 20 events. The maximum length of testcases generated using the ECIG is a property of the application under

test. Comparing the number of testcases of a given length, in this table, with EIG-based testcases in Table 5.6, shows that the number of ECIG-based testcases is drastically less. For example, for JEdit, at length 5, there are '$5,887,308,434,240$' EIG-based testcases, whereas there are 717 ECIG-based testcases. Similarly, comparing Table 5.9 with EFG-based testcases in Table 5.4, the ECIG-based testsuite at a given length is smaller. The reduced size of the ECIG-based testsuite makes is possible to execute it within a reasonable time.

RQ2: Can ECI relations be used to generate *long* testcases for a GUI-based application?
A: From Table 5.9, it can be seen that using the ECIG, it is possible to generate SequenceLength-n testcase where n is as large as 20. These testcases are longer than SequenceLength-2 EFG or EIG-based testcases. The size of the testsuite is such that it can be executed in a reasonable time. ■

Replayer: The ECIG-based testcases, created in the previous step, were executed on the applications. Table 5.10 shows the results observed after executing the ECIG-based testcases. Since the number of testcases, of a specific length, were reasonably small, all the testcases for an application were executed together as a single testsuite. In this table $\#testcases$ corresponds to the $Total$ column in Table 5.9. Executing the testsuites took between 1 and 12 days for each application. The wall-clock time for execution is shown in the column $execution\ clock\ time$. The number of concurrent execution of testcases varied between the application. The column $execution\ work$ shows the clock time multiplied by the number of concurrent executions. It estimates the total time that would be consumed if the testcases were executed without any concurrency. The GUI state of the application was collected after executing each event of every testcase. The size of the collected state is shown in $GUI\ state\ size$. Code coverage data and execution logs were also collected. Their size is shown in $coverage\ size$ and $execution\ log\ size$. Since these testcase are *long* testcases, they are expected to exercise the application in ways that is not possible with $n = 2$, EFG or EIG-based testcases.

App	# testcases	execution clock time	execution work	GUI state size (GB)	coverage size (compressed GB)	execution log size (GB)	code coverage
AU	28,105	9 days 5 hours	267 days	45	154	0.86	27.52%
BD	31,988	12 days	588 days	131	431	0.79	6.28%
JA	15,777	1 day 18 hours	115 days	49	78	0.37	21.33%
JE	28,787	4 days 2 hours	269 days	585	151	0.82	24.36%

Table 5.10: ECIG-based SequenceLength-n execution results, where $n \in \{3, ..., 20\}$.

Column *code coverage* in Table 5.10 shows the code covered by all testcases for an application. The code covered by the ECIG-based testsuite is less than the code covered by its seed testsuite. For example, for `JEdit`, the code coverage for ECIG-based testsuite is 24.36%, whereas the code covered by its seed testsuite is 25.70% (Table 5.7). This is expected, since the ECIG is a subset of the EIG. In addition, analysis of the code coverage from the ECIG-based testsuite showed that few *new* lines of code were executed, compared to the seed EIG-based testsuite. It indicates that executing the same events in different contexts does not necessarily result in increased code coverage. Intuitively, event handlers interact with other event handlers at the program-code and program-state level. Executing them in different contexts is likely to exercise their interaction with each other. Hence, code coverage is not sufficient to quantify the coverage delivered by a testsuite.

GUI state analysis: The GUI state of the application was extracted and stored after executing each event of every testcase, from ECIG-based testsuite. These GUI states were analysed to detect if new GUI states of the application was exercised.

Contribution 4: GUI state coverage is an alternate metric to measure the effectiveness of a testsuite. Long GUI testcases may not result in new code being exercised. However, they may result in the code being exercised in new ways. This may manifest itself as new GUI states being reached by the application. Measuring the number of GUI states reached (or created) by a testsuite may be used as a metric for measuring the effectiveness of a testsuite.

The GUI state recorded while executing the EIG-based *seed* SequenceLength-2 was compared with the ECIG-based long testcases. Each application was processed as follows. First, the procedure `sigUpdate` was executed on the GUI state of the seed testsuite to produce the `baselineSignature` as described in Section 3.4.3. Second, `sigUpdate` was executed on the GUI state collected from the ECIG-based testsuite. The resulting GUI state signatures were stored in `targetSignature`. Third, the procedure `newState` was executed using both the GUI state signatures. This produced a list of ECIG-based testcases that exercised new GUI states in the application. Table 5.11 shows the time taken to execute the procedure `sigUpdate` on the baseline EIG-based testsuites, the target ECIG-based testsuites. It also shows the time taken to compare the two sets of signatures using the procedure `newState`.

The state of an application was recorded after each event of every testcase was executed. For a testcase with an executable event sequence of length n, n states were recorded. For a testsuite, whose testcases together contain m events, m states were recorded. Since different testcases of a testsuite may cause the application to reach the same state, the number of unique states reached by an application may be less than m, for a given testsuite.

Table 5.12 shows information about GUI states of the applications that were reached, by the seed EIG-based and ECIG-based testsuites. Column (A) records the total number of states that were reached when the EIG-based testsuite was executed. This counts a state after the execution of each event or every testcase. For example, for `JEdit`, the testsuite caused the execution of $589,632$ successful events, recording as many states. Column (B) shows the number of states that were unique. It is possible that events from different testcase caused the application to reach the same state. For example, for `JEdit`, of all the states reached only $2,708$ states were unique. This shows that different testcases of the testsuite with different event sequences may cause the application to move within a small set of states.

The columns (C) and (D) shows the states that were reached by the ECIG-based testsuite, and the unique states within them. For example, for `JEdit`, $543,548$ events were successfully exe-

App	EIG `sigUpdate` time	ECIG `sigUpdate` time	EIG vs ECIG `newState` time
AU	5 hours 17 min	13 hours 42 min	14 min 50 sec
BD	1 hour 40 min	37 hours 33 min	12 hours 29 min
JA	5 hours 16 min	11 hours 52 min	15 min 21 sec
JE	24 hours 45 min	18 hours 42 min	5 hours 18 min

Table 5.11: Time taken to compute GUI state signatures, using procedure `sigUpdate`, for the baseline EIG-based `baselineSignature`, ECIG-based `targetSignature` and identifying new ECIG states using procedure `newState`.

.

App	EIG		ECIG		EIG vs ECIG		
	total states (A)	unique states (B)	total states (C)	unique states (D)	testcases (E)	new states (F)	unique new states (G)
AU	157,291	4,734	461,621	6,916	24,133	247,777	6,461
BD	38,951	1,160	943,935	8,611	31,827	637,291	8,378
JA	196,469	10,037	362,511	6,920	15,509	268,512	6,578
JE	589,632	2,708	543.548	2,038	26,414	261,245	1,744

Table 5.12: States detected during execution of EIG-based seed testsuite and ECIG-based target testsuites. Number of ECIG-based testcases detecting a new state is shown in *testcases* (E). Number of unique new states detected by the ECIG-based testsuite is shown in *unique new states* (G).

cuted, collecting as many states. Of these, $2,038$ unique states were reached. Column (E) shows the number of ECIG-based testcases that found one or more new states during its execution. For example, for `JEdit` $26,414$ testcases, out of $28,787$ ECIG-based testcases (see Table 5.9), found one or more new states.

Observation 1: Although the ECIG for `JEdit` has only 517 edges, compared to $133,734$ for the EIG, the number of unique states reached by the ECIG-based testsuite (column D) is comparable to that reached by the EIG-based testsuite (column B). This is true for all the applications.

Column (F) shows the number of states that are *new* in the ECIG-based testsuite. These new states were reached by the ECIG-based testsuites, but not by the EIG-based testsuites. For example,

App	(H)	(I)
AU	93.5%	6.5%
BD	97.2%	2.8%
JA	95.1%	4.9%
JE	85.6%	14.4%

Table 5.13: Column (H) shows the percentage of states reached by ECIG-based testsuite that are new. Column (I) shows the percentage of states reached by ECIG-based testsuite that were already reached by the baseline EIG-based testsuite.

for `JEdit`, $261,245$ new states were discovered during execution of the ECIG-based testsuite. Column (G) show that, of these, $1,744$ unique states were discovered.

By comparing columns (D) and (G), it can be seen that the number of unique new states comprise a significant part of the number of unique states, reached by the ECIG-based testsuite. This indicates that most states reached by the testsuite were new, compared to the baseline testsuite. To quantify this, Table 5.13 shows statistics derived from Table 5.12.

$$H = 100 * G/D \tag{5.1}$$

$$I = 100 * (D - G)/D \tag{5.2}$$

Column (H) shows the percentage of states reached by the ECIG testsuite that are new – that is, they were not reached by the baseline testsuite. Column (I) shows the percentage of states reached by the ECIG testsuite, that were already reached by the baseline EIG-based testsuite. For all the applications column (H) shows that a large percentage of states reached by the ECIG testsuite are new states.

Observation 2: Although the ECIG contained fewer edges, it was able to drive the application to reach a large number of new states and unique new states.

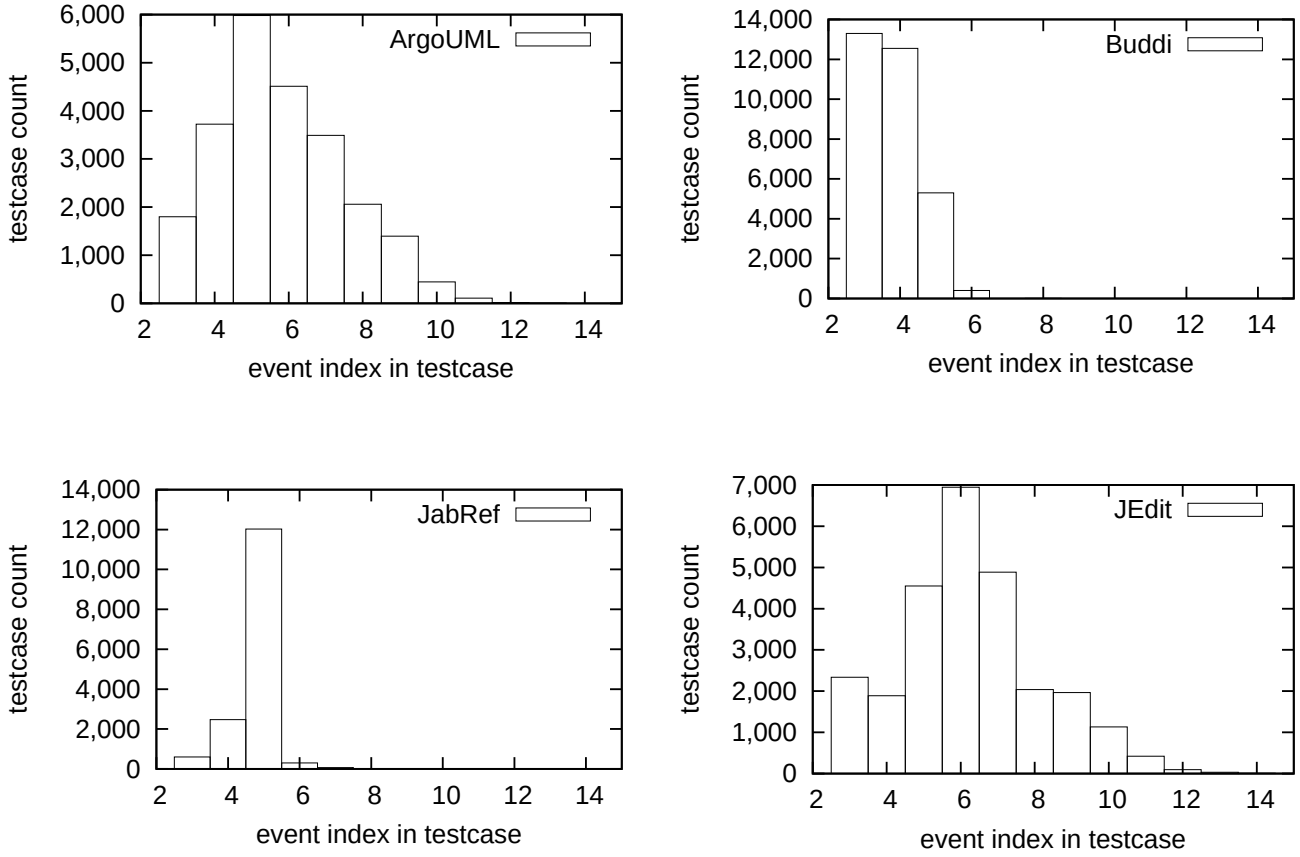


Figure 5.3: Count of testcases where the *first* new state was found after executing a specific number of events in the testcase.

RQ3: Can ECIG-based testcases exercise the GUI of a GUI-based application in *useful* ways?

A: In Table 5.12, column (G) shows the number of states that the application was able to reach, with ECIG-based testsuite, but not with EIG-based testsuite. Presence of the new states indicate that the application was exercised in new ways. The ECIG-based testsuite is able to test the application in ways that was not possible with EIG-based testcases. In addition, each unique new state could be reached using multiple testcases. ■

When an ECIG-based testcase is executed, it may find a new state in the application. The new state may be reached by any of the events in the testcase. It would be informative to know at what stage of the execution of the testcase is the *first* new state typically found. Figure 5.3 shows a count of testcases where the first new state was found after executing a specific number of events from the testcase. A plot is shown for each application. In each plot, the X-axis is the event index in the testcase and the Y-axis is the count of testcases, where the first new state was found at a specific event index. From the figure, it can be seen that the first new state is discovered after executing only a few events. For example, for `JEdit`, 2.338 testcases found the first new state after executing 3 events; 8 testcases found the first new state after executing 14 events. This shows that some new states are found after executing as many as 14 events. Hence, ECIG-based testcases of different lengths are all useful in finding new states.

RQ4: Is ECIG-based testcases good for detecting software defects?
A: A GUI state reached during the execution of events in a testcase represents the cumulative interaction of program-code until that event. Each GUI state intuitively represents one or more chains of interaction, that leads to that state. Software defects embedded in a chain of interaction may be exercised when that state is reached by the GUI. ECIG-based testcases are shown to discover new states. A newly discovered GUI state represents one or more new chains of interaction. These new interactions may contain defects that are not exercised by the baseline EIG-based testcases. When an ECIG-based testcase discovers a new state, it indicates that a new chain of interaction was exercised. Hence new states discovered by ECIG-based testcases may be deemed to correspond to software defects embedded in one or more new chains of interaction.

Since ECIG-based testcases are shown to discover new GUI states (Table 5.12), they may be deemed good for exercising new software defects.

The empirical evaluation demonstrates the existence of ECI relations in GUI-based applications. It demonstrates a practical method to identify ECI relations in an application, model it as an Event-Code Interaction Graph and generate long testcases based on the ECIG. These long testcases are shown to exercise and test the application in ways that was not possible using existing GUI models.

5.4 Discussion

Existing model-based GUI testing methods examine the characteristics of the visible state of the GUI. These characteristics have been modelled as the EFG, EIG, ESIG and state machines. A significant contribution of this study is that it connects the characteristics of the visible GUI of a GUI-based application to its program-code. The ECIG models the new relationship. Long GUI testcases generated from the ECIG model are shown to exercise the GUI in new ways.

This study has used the EIG as a model for creating the baseline testsuite. This model and its testsuites were used to detect ECI relations in an application. It is possible to use another suitable model to generate the baseline testsuite. For example, the EFG, ESIG or a state machine model may be appropriately used. The baseline testsuite from these models may help identify ECI relations that are not identified using the EIG as the baseline model. The discovery of the new ECI relations are caused by the difference between the EIG and the other baseline models. It is not owing to any inherent characteristic of the ECI concept. The ECI model is therefore sensitive to the choice of a baseline model.

The ECI relations determined from the baseline SequenceLength-2 testsuites in this study were based on **Approach L1 (unique-lines-of-code**) method of code coverage (see Section 3.2). Using a different method to characterize code coverage may alter the nature of ECI relations. This can result in the identification of a different set of event-pairs as ECI event pairs. Hence, the ECI relations are sensitive to the choice of a code coverage metric.

Chapter 6

Future work

Event-Code Interaction is the first study that examines the relationships between GUI events and program-code that respond to GUI events. It leverages these relationships to generate long GUI testcases that are able to drive the GUI into new states. This study extends existing research on GUI models such as Event-Flow Graph, Event-Interaction Graph and Event-Semantic Interaction Graph, which work with the visible, run-time GUI state of an application. This study demonstrates the following:

1. That there exists program-code level interactions, defined as Event-Code Interaction (ECI), between GUI event handlers.
2. That these interactions can be systematically identified, using practical methods.
3. That these interactions can be modelled as an Event-Code Interaction Graph (ECIG).
4. That the ECIG can be used to generate long GUI testcases, contained in a small testsuite.
5. That the ECIG-based testcases drive the GUI to reach new GUI states that can detect defects in the application.

In addition, this study develops a simple and fast method to uniquely identify GUI states. This method, defined as *Hierarchical signatures*, can be used to enumerate GUI states, as well as compare two sets of GUI states (see Section 3.4.2).

The design, development and execution of this study leaves certain questions unanswered and opens the way for further investigation.

Q1: Do GUI applications and their program-code interact in deeper ways?

Composite ECI: The ECI relation examined in this study attempts to find program-code level interaction between two GUI events. In this interaction, an event e_1 is said to interact with another event e_2 at the program-code level, if the execution of e_1 before e_2 influences the execution of e_2.

The ECI concept may be extended to 3 or more events. Consider 3 events, e_1, e_2 and e_3, where there is no ECI relation between e_1 and e_3 or e_2 and e_3. However, the combined execution of e_1 and e_2 may affect the subsequent execution of e_3. This is represented as $\{e_1, e_2\} \longrightarrow e_3$ and is called *Composite Event-Code Interaction*. Section 3.3.3 describes composite ECI with an example. A study of composite ECI relations could give a better understanding of GUI applications and event-driven systems.

Alternate seeds: ECI relations and the ECIG are created from a seed testsuite for an application. In this study, EIG-based SequenceLength-2 testsuites were used as seed testsuites. It is possible to use seed testsuites generated using different methods. For example, an EFG-based or ESIG-based testsuite may be used. The EFG, EIG and ESIG posses different properties. Seed testsuites generated from these models may yield different ECI relations. A study of EFG or ESIG-based ECI relations could provide alternative methods for creating and using an ECIG.

Alternate coverage: ECI relations were identified in this study by analyzing the code coverage of event handlers in response to GUI events. The code coverage method that was adopted is **Approach (L1) unique-lines-of-code** (see Section 3.2). In this method, the execution status of a line of program-code was considered to be a Boolean value, `TRUE` or `FALSE`. Multiple executions of the same line of code was considered as `TRUE`.

Using a different method for quantifying code coverage, and hence code coverage differences, is likely to yield deeper information about interactions between the GUI and its event handlers. Alternative code coverage methods have been proposed in Section 3.2, where **Approach L2 (line-hit-count)** distinguishes between the number of times a line was executed and **Approach L3**

(line-sequence) records the sequence of execution of each line. A different method for quantifying code coverage could yield different ECI relations.

Q2: Does the ECI concept developed in this study, apply to other event-driven systems?

The ECI concept developed in this study uses Java-based GUI applications, as examples of event-driven system, for evaluation. The visible run-time state of applications were used to determine if a combined execution of events affect the execution of an event. This concept may be generalized by: 1) expanding the nature of event-driven system and 2) considering different observable metrics of a system to detect changes in expected state. A few such systems are:

Internet-of-things: Connected devices, for example in the domain of internet-of-things, interact with each other using a variety of signals, messages and data. These inputs are likely to invoke the execution of program-code within a connected device. A specific combination of inputs from other devices may cause a device to behave differently. The observed behaviour may be in terms of the messages it sends to other devices. Examination of such behaviour would help in the development of better hardware, software and verification methods for such connected devices.

non-GUI systems: Event-driven systems - such as computer hardware components, mobile device components, Java enabled devices[1] - that are event-driven may also exhibit ECI behaviour. Applying the ECI concept to such systems may help in developing better testing strategies.

Security: Security of computer software and the hardware they control is an integral component of most software systems. The response of software to unexpected or crafted inputs is of particular interest in the security domain. Examination of ECI relations may help understand the security characteristics of a software and develop better testing strategies.

The above questions are a guidance for further investigation, based on the Event-Code Interaction concept, developed in this study.

[1] http://www.java.com/en/about/

Appendix A

Radio Button Demo

A	B	C	D	E	F	G	L	Radio Button Demo
-	-	-	-	-	-	-	0	`import java.awt.BorderLayout;`
-	-	-	-	-	-	-	1	`import java.awt.Color;`
-	-	-	-	-	-	-	2	`import java.awt.Graphics;`
-	-	-	-	-	-	-	3	`import java.awt.event.ActionEvent;`
-	-	-	-	-	-	-	4	`import java.awt.event.ActionListener;`
-	-	-	-	-	-	-	5	`.`
-	-	-	-	-	-	-	6	`import javax.swing.BorderFactory;`
-	-	-	-	-	-	-	7	`import javax.swing.Box;`
-	-	-	-	-	-	-	8	`import javax.swing.BoxLayout;`
-	-	-	-	-	-	-	9	`import javax.swing.ButtonGroup;`
-	-	-	-	-	-	-	10	`import javax.swing.JButton;`
-	-	-	-	-	-	-	11	`import javax.swing.JCheckBox;`
-	-	-	-	-	-	-	12	`import javax.swing.JFrame;`
-	-	-	-	-	-	-	13	`import javax.swing.JOptionPane;`
-	-	-	-	-	-	-	14	`import javax.swing.JPanel;`
-	-	-	-	-	-	-	15	`import javax.swing.JRadioButton;`
-	-	-	-	-	-	-	16	`.`
X	X	X	-	-	-	-	17	`public class RadioButtonDemo extends JFrame {`
-	-	-	-	-	-	-	18	`  private static final long serialVersionUID = 1L;`
-	-	-	-	-	-	-	19	`.`
-	-	-	-	-	-	-	20	`  JButton w0; // Exit`
-	-	-	-	-	-	-	21	`  JRadioButton w1; // Circle`
-	-	-	-	-	-	-	22	`  JRadioButton w2; // Square`
-	-	-	-	-	-	-	23	`.`
-	-	-	-	-	-	-	24	`  JButton w3; // Create`
-	-	-	-	-	-	-	25	`  JPanel w4; // Rendered shape`
-	-	-	-	-	-	-	26	`  JButton w5; // Reset`
-	-	-	-	-	-	-	27	`.`
-	-	-	-	-	-	-	28	`  JCheckBox w6; // Log exit time`
-	-	-	-	-	-	-	29	`.`
-	-	-	-	-	-	-	30	`  JPanel contentPane;`
-	-	-	-	-	-	-	31	`.`
X	-	-	-	-	-	-	32	`  int exitCircle = 0, exitSquare = 0;`
-	-	-	-	-	-	-	33	`  Shape currentShape;`
X	-	-	-	-	-	-	34	`  Boolean created = false;`
-	-	-	-	-	-	-	35	`.`
X	-	X	-	X	-	-	36	`  enum Shape {`
X	-	X	-	X	-	-	37	`    CIRCLE, SQUARE`
-	-	-	-	-	-	-	38	`  }`
-	-	-	-	-	-	-	39	`.`
-	-	-	-	-	-	-	40	`  public RadioButtonDemo() {`
X	-	-	-	-	-	-	41	`    super("Radio Button Demo");`
X	-	-	-	-	-	-	42	`    contentPane = new JPanel(new BorderLayout());`
-	-	-	-	-	-	-	43	`.`

```
X - - - - - -  44      w1 = new JRadioButton("Circle");
X - - - - - -  45      w1.setSelected(false);
X - - - - - -  46      w1.addActionListener(new W1Listener());
- - - - - - -  47  .
X - - - - - -  48      w2 = new JRadioButton("Square");
X - - - - - -  49      w2.setSelected(false);
X - - - - - -  50      w2.addActionListener(new W2Listener());
- - - - - - -  51  .
X - - - - - -  52      ButtonGroup selectShapeGroup = new ButtonGroup();
X - - - - - -  53      selectShapeGroup.add(w1);
X - - - - - -  54      selectShapeGroup.add(w2);
- - - - - - -  55  .
X - - - - - -  56      Box selectShapePanel = new Box(BoxLayout.Y_AXIS);
X - - - - - -  57      selectShapePanel.add(w1);
X - - - - - -  58      selectShapePanel.add(w2);
- - - - - - -  59  .
X - - - - - -  60      selectShapePanel.setBorder(
- - - - - - -  61          BorderFactory.createTitledBorder(
- - - - - - -  62          BorderFactory.createLineBorder(Color.GRAY),
- - - - - - -  63                                  "Select"));
- - - - - - -  64  .
X - - - - - -  65      draw(new EmptyPanel());
- - - - - - -  66  .
X - - - - - -  67      w3 = new JButton("Create");
X - - - - - -  68      w3.addActionListener(new W3Listener());
- - - - - - -  69  .
X - - - - - -  70      w5 = new JButton("Reset");
X - - - - - -  71      w5.setEnabled(false);
X - - - - - -  72      w5.addActionListener(new W5Listener());
- - - - - - -  73  .
X - - - - - -  74      w0 = new JButton("Exit");
X - - - - - -  75      w0.addActionListener(new W0Listener());
- - - - - - -  76  .
X - - - - - -  77      JPanel buttonPanel = new JPanel();
X - - - - - -  78      buttonPanel.add(w3);
X - - - - - -  79      buttonPanel.add(w5);
X - - - - - -  80      buttonPanel.add(w0);
- - - - - - -  81  .
X - - - - - -  82      contentPane.add(selectShapePanel,
- - - - - - -  83                      BorderLayout.WEST);
X - - - - - -  84      contentPane.add(w4, BorderLayout.CENTER);
X - - - - - -  85      contentPane.add(buttonPanel, BorderLayout.SOUTH);
- - - - - - -  86  .
X - - - - - -  87      w6 = new JCheckBox("Log exit time.");
```

```
- - - - - - -  88   .
X - - - - - -  89       setContentPane(contentPane);
X - - - - - -  90       setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
X - - - - - -  91     }
- - - - - - -  92   .
- - - - - - -  93     private void draw(JPanel shape) {
X X X - - - -  94       if (w4 != null) {
X X X - - - -  95         contentPane.remove(w4);
- - - - - - -  96       }
X X X - - - -  97       w4 = shape;
X X X - - - -  98       w4.setBorder(BorderFactory.createTitledBorder(
- - - - - - -  99           BorderFactory.createLineBorder(Color.GRAY),
- - - - - - - 100           "Rendered Shape"));
X X X - - - - 101       contentPane.add(w4);
X X X - - - - 102       repaint();
X X X - - - - 103       pack();
X X X - - - - 104     }
- - - - - - - 105   .
X - - - - - - 106     class W1Listener implements ActionListener {
- - - - - - - 107       @Override
- - - - - - - 108       public void actionPerformed(ActionEvent arg0) {
- - - X X - - 109         currentShape = Shape.CIRCLE;
- - - X X - - 110         if (created) {
- - - - - - - 111           draw(new CirclePanel());
- - - - - - - 112         }
- - - X X - - 113       }
- - - - - - - 114     }
- - - - - - - 115   .
X - - - - - - 116     class W2Listener implements ActionListener {
- - - - - - - 117   .
- - - - - - - 118       @Override
- - - - - - - 119       public void actionPerformed(ActionEvent arg0) {
X - - - - - - 120         currentShape = Shape.SQUARE;
X - - - - - - 121         if (created) {
- - - - - - - 122           draw(new SquarePanel());
- - - - - - - 123         }
X - - - - - - 124       }
- - - - - - - 125     }
- - - - - - - 126   .
X - - - - - - 127     class W3Listener implements ActionListener {
- - - - - - - 128       @Override
- - - - - - - 129       public void actionPerformed(ActionEvent e) {
- - - - - - - 130         JPanel shape;
X X X - - - - 131         if (currentShape == Shape.CIRCLE ||
```

```
- - - - - - -  132            currentShape == Shape.SQUARE) {
X X - - - - -  133          w5.setEnabled(true);
- - - - - - -  134        }
X X X - - - -  135        created = true;
X X X - - - -  136        if (currentShape == Shape.CIRCLE)
- X - - - - -  137          shape = new CirclePanel();
X - X - - - -  138        else if (currentShape == Shape.SQUARE)
X - - - - - -  139          shape = new SquarePanel();
- - - - - - -  140        else
- - X - - - -  141          shape = new EmptyPanel();
X X X - - - -  142        draw(shape);
X X X - - - -  143      }
- - - - - - -  144    }
- - - - - - -  145  .
X - - - - - -  146    class W5Listener implements ActionListener {
- - - - - - -  147      @Override
- - - - - - -  148      public void actionPerformed(ActionEvent e) {
- - - - - - -  149        w5.setEnabled(false);
- - - - - - -  150        created = false;
- - - - - - -  151        draw(new EmptyPanel());
- - - - - - -  152      }
- - - - - - -  153    }
- - - - - - -  154  .
X - - - - - -  155    class W0Listener implements ActionListener {
- - - - - - -  156      @Override
- - - - - - -  157      public void actionPerformed(ActionEvent e) {
- - - - - - -  158  .
- - - - - X X  159        String message = "Are you sure?";
- - - - - - -  160  .
- - - - - X X  161        if (created == true &&
- - - - - - -  162            currentShape == Shape.CIRCLE) {
- - - - - X -  163          exitCircle++;
- - - - - - -  164        }
- - - - - X X  165        if (created == true &&
- - - - - - -  166            currentShape == Shape.SQUARE) {
- - - - - - -  167          exitSquare++;
- - - - - - -  168        }
- - - - - - -  169  .
- - - - - X X  170        Object[] params = { message, w6 };
- - - - - - -  171  .
- - - - - X X  172        if (exitCircle == 10 || exitSquare == 10) {
- - - - - - -  173          System.exit(0);
- - - - - - -  174        }
- - - - - X X  175        int exit =
```

```
- - - - - - -  176            JOptionPane.showConfirmDialog(null, params,
- - - - - - -  177            "Exit Confirmation",
- - - - - - -  178            JOptionPane.YES_NO_OPTION);
- - - - - - -  179          if (exit == 0) {
- - - - - - -  180            if (w6.isSelected()) {
- - - - - - -  181              writeTimeStamp();
- - - - - - -  182            }
- - - - - - -  183            System.exit(0);
- - - - - - -  184          }
- - - - - - -  185        }
- - - - - - -  186      }
- - - - - - -  187  .
- - - - - - -  188      private void writeTimeStamp() {
- - - - - - -  189      }
- - - - - - -  190  .
- - - - - - -  191      class CirclePanel extends JPanel {
- - - - - - -  192        private static final long serialVersionUID = 1L;
- - - - - - -  193  .
- X - - - - -  194        public CirclePanel() {
- X - - - - -  195          super();
- X - - - - -  196        }
- - - - - - -  197  .
- - - - - - -  198        @Override
- - - - - - -  199        public void paintComponent(Graphics g) {
- X - - - - -  200          g.drawOval(60, 30, 45, 45);
- X - - - - -  201        }
- - - - - - -  202      }
- - - - - - -  203  .
- - - - - - -  204      class SquarePanel extends JPanel {
- - - - - - -  205        private static final long serialVersionUID = 1L;
- - - - - - -  206  .
X - - - - - -  207        public SquarePanel() {
X - - - - - -  208          super();
X - - - - - -  209        }
- - - - - - -  210  .
- - - - - - -  211        @Override
- - - - - - -  212        public void paintComponent(Graphics g) {
X - - - - - -  213          g.drawRect(60, 30, 45, 45);
X - - - - - -  214        }
- - - - - - -  215      }
- - - - - - -  216  .
- - - - - - -  217      class EmptyPanel extends JPanel {
- - - - - - -  218        private static final long serialVersionUID = 1L;
- - - - - - -  219  .
```

A	B	C	D	E	F	G	L	Radio Button Demo
X	-	X	-	-	-	-	220	`    public EmptyPanel() {`
X	-	X	-	-	-	-	221	`      super();`
X	-	X	-	-	-	-	222	`    }`
-	-	-	-	-	-	-	223	`  }`
-	-	-	-	-	-	-	224	`.`
-	-	-	-	-	-	-	225	`  private static void createAndShowGUI() {`
X	-	-	-	-	-	-	226	`    RadioButtonDemo frame = new RadioButtonDemo();`
X	-	-	-	-	-	-	227	`    frame.setVisible(true);`
X	-	-	-	-	-	-	228	`    frame.pack();`
X	-	-	-	-	-	-	229	`  }`
-	-	-	-	-	-	-	230	`.`
-	-	-	-	-	-	-	231	`  public static void main(String[] args) {`
X	-	-	-	-	-	-	232	`    javax.swing.SwingUtilities.invokeLater(`
X	-	-	-	-	-	-	233	`      new Runnable() {`
-	-	-	-	-	-	-	234	`      public void run() {`
X	-	-	-	-	-	-	235	`        createAndShowGUI();`
X	-	-	-	-	-	-	236	`      }`
-	-	-	-	-	-	-	237	`    });`
X	-	-	-	-	-	-	238	`  }`
-	-	-	-	-	-	-	239	`}`

Bibliography

[1] AMALFITANO, D., FASOLINO, A. R., AND TRAMONTANA, P. Rich Internet Application Testing Using Execution Trace Data. In *Conference on Software Testing, Verification, and Validation Workshops* (2010), pp. 274–283.

[2] ARISS, O. E., XU, D., DANDEY, S., VENDER, B., MCCLEAN, P., AND SLATOR, B. A Systematic Capture and Replay Strategy for Testing Complex GUI Based Java Applications. In *Conference on Information Technology* (2010), pp. 1038–1043.

[3] ARLT, S., BANERJEE, I., BERTOLINI, C., MEMON, A. M., AND SCHAF, M. Grey-box GUI Testing: Efficient Generation of Event Sequences. *CoRR abs/1205.4928* (2012).

[4] ARLT, S., PODELSKI, A., BERTOLINI, C., SCHAF, M., BANERJEE, I., AND MEMON, A. Lightweight Static Analysis for GUI Testing. In *ISSRE'12 Proceedings of the 23rd IEEE International Symposium on Software Reliability Engineering* (Washington, DC, USA, 2012), IEEE Computer Society.

[5] BANERJEE, I., NGUYEN, B., GAROUSI, V., AND MEMON, A. Graphical User Interface (GUI) Testing: Systematic Mapping and Repository. *Information and Software Technology* (2013).

[6] BELLI, F. Finite-State Testing and Analysis of Graphical User Interfaces. In *Symposium on Software Reliability Engineering* (2001), p. 34.

[7] BERTOLINI, C., AND MOTA, A. A Framework for GUI Testing based on Use Case Design. In *Conference on Software Testing, Verification, and Validation Workshops* (2010), pp. 252–259.

[8] BERTOLINI, C., PERES, G., AMORIM, M., AND MOTA, A. An Empirical Evaluation of Automated Black-Box Testing Techniques for Crashing GUIs. In *Software Testing Verification and Validation* (2009), pp. 21–30.

[9] CAI, K.-Y., ZHAO, L., AND WANG, F. A Dynamic Partitioning Approach for GUI Testing. In *Computer Software and Applications* (2006), pp. 223–228.

[10] CHANG, T.-H., YEH, T., AND MILLER, R. C. GUI Testing Using Computer Vision. In *Conference on Human factors in computing systems* (2010), pp. 1535–1544.

[11] CHEN, J., AND SUBRAMANIAM, S. Specification-based Testing for GUI-based Applications. *Software Quality Journal 10*, 2 (2002), 205–224.

[12] CHEN, W.-K., TSAI, T.-H., AND CHAO, H.-H. Integration of Specification-Based and CR-Based Approaches for GUI Testing. In *Conference on Advanced Information Networking and Applications* (2005), pp. 967–972.

[13] DABOCZI, T., KOLLAR, I., SIMON, G., AND MEGYERI, T. Automatic Testing of Graphical User Interfaces. In *Instrumentation and Measurement Technology Conference* (2003), pp. 441–445.

[14] DEREZINSKA, A., AND MALEK, T. Experiences in Testing Automation of a Family of Functional-and GUI-similar Programs. *Journal of Computer Science & Applications 4*, 1 (2007), 13–26.

[15] ESMELIOGLU, S., AND APFELBAUM, L. Automated test generation, execution, and reporting. In *In Proceedings of Pacific Northwest Software Quality Conference* (Oct 1997), IEEE Press.

[16] GANOV, S., KILLMAR, C., KHURSHID, S., AND PERRY, D. E. Event Listener Analysis and Symbolic Execution for Testing GUI Applications. *Formal methods and software engineering 5885*, 1 (2009), 69–87.

[17] GARG, A., VIDYARAMAN, S., UPADHYAYA, S., AND KWIAT, K. USim: a User Behavior Simulation Framework for Training and Testing IDSes in GUI Based Systems. In *Symposium on Simulation* (2006), pp. 196–203.

[18] GRECHANIK, M., XIE, Q., AND FU, C. Creating GUI Testing Tools Using Accessibility Technologies. In *Conference on Software Testing, Verification, and Validation* (2009), pp. 243–250.

[19] HELLMANN, T. D., HOSSEINI-KHAYAT, A., AND MAURER, F. Supporting Test-Driven Development of Graphical User Interfaces Using Agile Interaction Design. In *Proceedings of the 2010 Third International Conference on Software Testing, Verification, and Validation Workshops* (2010), pp. 444–447.

[20] HICINBOTHOM, J. H., AND ZACHARY, W. W. *A Tool for Automatically Generating Transcripts of Human-Computer Interaction*, vol. 2. 1993, p. 1042.

[21] HU, C., AND NEAMTIU, I. A GUI bug finding framework for Android applications. In *Proceedings of the 2011 ACM Symposium on Applied Computing* (2011), SAC '11, ACM, pp. 1490–1491.

[22] HU, C., AND NEAMTIU, I. Automating GUI testing for Android applications. In *Proceedings of the 6th International Workshop on Automation of Software Test* (2011), pp. 77–83.

[23] JAASKELAINEN, A., KATARA, M., KERVINEN, A., MAUNUMAA, M., PAAKKONEN, T., TAKALA, T., AND VIRTANEN, H. Automatic GUI test generation for smartphone applications - an evaluation. In *Conference on Software Engineering* (2009), pp. 112–122.

[24] KASIK, D. J., AND GEORGE, H. G. Toward automatic generation of novice user test scripts. In *Proceedings of the SIGCHI conference on Human factors in computing systems* (1996), ACM, pp. 244–251.

[25] MEMON, A., BANERJEE, I., HASHMI, N., AND NAGARAJAN, A. DART: A Framework for Regression Testing. In *Conference on Software Maintenance* (2003), pp. 410–419.

[26] MEMON, A., BANERJEE, I., AND NAGARAJAN, A. GUI ripping: Reverse engineering of graphical user interfaces for testing. In *Conference on Reverse Engineering* (2003), pp. 260–269.

[27] MEMON, A., BANERJEE, I., NGUYEN, B., AND ROBBINS, B. The First Decade of GUI Ripping: Extensions, Applications, and Broader Impacts. In *Proceedings of the 20th Working Conference on Reverse Engineering (WCRE)* (2013), IEEE Press.

[28] MEMON, A. M. Developing testing techniques for event-driven pervasive computing applications. In *Proceedings of The OOPSLA 2004 workshop on Building Software for Pervasive Computing (BSPC 2004)* (2004).

[29] MEMON, A. M. Employing User Profiles to Test a New Version of a GUI Component in its Context of Use. *Software Quality Control 14*, 4 (2006), 359–377.

[30] MEMON, A. M. An Event-flow Model of GUI-based Applications for Testing. *Software Testing, Verification & Reliability 17*, 3 (2007), 137–157.

[31] MEMON, A. M., BANERJEE, I., AND NAGARAJAN, A. Automatically testing "nightly/daily builds" of GUI applications. In *Proceedings of the 2003 International Conference on Dependable Systems and Networks* (June 2003).

[32] MEMON, A. M., BANERJEE, I., AND NAGARAJAN, A. What Test Oracle Should I Use for Effective GUI Testing? In *Conference on Automated Software Engineering* (2003), pp. 164–173.

[33] MEMON, A. M., POLLACK, M. E., AND SOFFA, M. L. Hierarchical GUI Test Case Generation Using Automated Planning. *IEEE Transactions on Software Engineering 27*, 2 (2001), 144–155.

[34] MEMON, A. M., AND XIE, Q. Studying the Fault-Detection Effectiveness of GUI Test Cases for Rapidly Evolving Software. *IEEE Trans. Softw. Eng.* (2005), 884–896.

[35] NGUYEN, B. N., ROBBINS, B., BANERJEE, I., AND MEMON, A. GUITAR: An innovative tool for automated testing of GUI-driven software. *Automated Software Engineering* (2013), 1–41.

[36] OSTRAND, T., ANODIDE, A., FOSTER, H., AND GORADIA, T. A visual test development environment for GUI systems. *ACM SIGSOFT Software Engineering Notes 23*, 2 (1998), 82–92.

[37] PAIVAA, A. C., FARIAA, J. C., AND VIDAL, R. F. Towards the Integration of Visual and Formal Models for GUI Testing. *Electronic Notes in Theoretical Computer Science 190*, 2 (2007), 99–111.

[38] RISOLDI, M., AND BUCHS, D. *A domain specific language and methodology for control systems GUI specification, verification and prototyping*. 2007 IEEE Symposium on Visual Languages and Human-Centric Computing. IEEE, 2007, pp. 179–182.

[39] SHEHADY, R. K., AND SIEWIOREK, D. P. A Method to Automate User Interface Testing Using Variable Finite State Machines. In *Symposium on Fault-Tolerant Computing* (1997), p. 80.

[40] SUN, Y., AND JONES, E. L. Specification-driven Automated Testing of GUI-based Java Programs. In *ACM Southeast Regional Conference* (2004), pp. 140–145.

[41] TAKALA, T., KATARA, M., AND HARTY, J. Experiences of system-level model-based GUI testing of an Android application. In *Software Testing, Verification and Validation (ICST), 2011 IEEE Fourth International Conference on* (2011), IEEE, pp. 377–386.

[42] WHITE, L., AND ALMEZEN, H. Generating Test Cases for GUI Responsibilities Using Complete Interaction Sequences. In *Symposium on Software Reliability Engineering* (2000), p. 110.

[43] XIE, Q., AND MEMON, A. M. Designing and Comparing Automated Test Oracles for GUI-based Software Applications. *ACM Transactions on Software Engineering and Methodology 16*, 1 (2007), 1–36.

[44] XIE, Q., AND MEMON, A. M. Using a Pilot Study to Derive a GUI Model for Automated Testing. *ACM Transactions on Software Engineering and Methodology 18*, 2 (2008), 1–33.

[45] YUAN, X., COHEN, M. B., AND MEMON, A. M. Towards Dynamic Adaptive Automated Test Generation for Graphical User Interfaces. In *Conference on Software Testing, Verification, and Validation Workshops* (2009), pp. 263–266.

[46] YUAN, X., COHEN, M. B., AND MEMON, A. M. GUI Interaction Testing: Incorporating Event Context. In *IEEE Transactions on Software Engineering* (2010).

[47] YUAN, X., AND MEMON, A. M. Generating Event Sequence-Based Test Cases Using GUI Runtime State Feedback. *IEEE Trans. Softw. Eng. 36* (January 2010), 81–95.