ABSTRACT

Title of dissertation: Testing GUI-based Software with Undetermined Input Spaces

Bao N. Nguyen, Doctor of Philosophy, 2013

Dissertation directed by: Professor Atif M. Memon
Department of Computer Science
University of Maryland, College Park

Most software applications feature a Graphical User Interface (GUI) front-end as the main, and often the only, method for the user to interact with the software. System-testing a software application requires it to be tested as a whole through the GUI. Testers need to generate sequences of GUI events (e.g., mouse clicks and menu selections) to exercise various behaviors of the application. Because the input space of a typical GUI (i.e., the space of all possible GUI events and their interactions) is often enormous, manual GUI testing is impractical. Model-based testing is a new approach that *automatically* and *systematically* generates a large number of test cases by leveraging a formal model representing the GUI input space. Unfortunately, modern applications often have a "context-sensitive reachability GUI," in which the GUI components are only reachable with some particular state or environment constraints. Thus, it is challenging to determine the GUI input space and and obtain a GUI model for automated GUI testing.

This research proposes new testing techniques to tackle the challenges in

model-based GUI testing. The central thesis is this: *GUI-based applications can be effectively and efficiently tested by systematically and incrementally leveraging the application runtime execution observations.*

To explore the thesis, a novel model-based testing paradigm called Observer-Model-Exercise* (OME*) is developed. This paradigm relies on the opportunistic observations obtained during test case execution to incrementally explore the GUI input space and construct a GUI model for test case generation.

To evaluate OME*, an open-source automated model-based GUI testing framework called GUITAR is developed. An empirical study with 8 widely-used open-source applications demonstrated that the OME* approach is feasible. Compared to previous model-based testing approaches, OME* was able to increase the GUI input space discovered by as much as 1,044%. As a result, 34 new faults were detected in the subject applications.

Testing GUI-based Software With Undetermined Input Spaces

by

Bao N. Nguyen

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
2013

Advisory Committee:
Professor Atif M. Memon, Chair and Advisor
Professor Ashok K. Agrawala
Professor Pete Keleher
Professor Alan Sussman
Professor Gang Qu

© Copyright by
Bao N. Nguyen
2013

Acknowledgments

This dissertation would not have been possible without the help, encouragement, and support of many other people.

I would like to thank my advisor, Atif Memon, for his continuous advice, support, and encouragement since I first came to UMD. I was very fortunate to have Atif as my advisor. I would also like to thank my other thesis committee members, Ashok Agrawala, Pete Keleher, Alan Sussman and Gang Qu, for giving me valuable feedback on the thesis, and selfless support in the process.

Many of my colleagues commented on my ideas, critiqued my drafts, directed me to related work, or provided bug patches for my programs. I am grateful to Penelope Brooks, Jaymie Strecker, Scott McMaster, Xun Yuan, Mike Lam, Ishan Banerjee, Ethar Elsaka, Leslie Milton, Bryan Robbins, and Bryan Ta.

Special thanks to the students in two classes CMSC 435 and CMSC 737 for their contributions to the GUITAR project. I would also like to acknowledge the open-source community for making their software publicly available. Without their products, I would not be able to conduct my experiments to validate my research.

Part of my research was funded by the National Science Foundation and the Vietnam Education Foundation. I am grateful to all the folks in these organizations. Their financial support allowed me to concentrate on research.

Finally, I own my deepest thanks to my family, especially my parents and my loving wife, Trang. Their exceptional support, love, and understanding have been a constant source of encouragement for me during my PhD.

Table of Contents

List of Tables v

List of Figures vi

1 Introduction 1
- 1.1 Thesis Statement 7
- 1.2 Approach 7
- 1.3 Related Work 9
- 1.4 Research Scope 12
- 1.5 Dissertation Outline 12

2 Background 14
- 2.1 Running Example 15
- 2.2 Test Case Generation Techniques 17
  - 2.2.1 State Machines 17
    - 2.2.1.1 Finite State Machines 18
    - 2.2.1.2 Variable Finite State Machines 21
    - 2.2.1.3 Complete Interaction Sequences 23
    - 2.2.1.4 Off-nominal Finite State Machines 27
  - 2.2.2 Workflows 29
    - 2.2.2.1 Event Flow Graph 30
    - 2.2.2.2 Event Interaction Graph 32
    - 2.2.2.3 Event Semantic Interaction Graph 35
    - 2.2.2.4 Off-nominal Event Graph 41
  - 2.2.3 Pre- Post-Condition Models 44
  - 2.2.4 Event Sequence-Based Models 47
  - 2.2.5 Probabilistic Models 50
  - 2.2.6 Combinatorial Interaction Models 54
    - 2.2.6.1 Latin Squares 54
    - 2.2.6.2 Covering Arrays 56
  - 2.2.7 Hierarchical Models 57
    - 2.2.7.1 Keyword-driven Models 58
    - 2.2.7.2 Hierarchical Finite State Machines 62
    - 2.2.7.3 UML Diagram-based 64
- 2.3 Summary 67

3 Observe-Model-Exercise* Testing Paradigm 69
- 3.1 Overview 70
- 3.2 Realizing the OME* Paradigm 76
  - 3.2.1 Contribution 1: Context-Aware Mapping 76
  - 3.2.2 Contribution 2: Simultaneously Extracting New Model Elements During Test Execution 79
  - 3.2.3 Contribution 3: Unique Widget Signatures 82

3.2.4 Contribution 4: Incremental EFG$^+$ Enhancements 85
3.2.5 Contribution 5: Incremental Test-Case Generation 87
3.3 Summary 88

4 GUITAR: A Generic Model-based GUI Testing Framework 89
4.1 Overall Architecture 89
4.1.1 Model Core 90
4.1.2 Platform-independent Components 92
4.1.3 Platform-specific Components 94
4.1.3.1 The Ripper 96
4.1.3.2 The Replayer 97
4.1.3.3 Using Executors 98
4.2 Creating Testing Workflow 99
4.3 Extending GUITAR 101
4.3.1 Within-platform Extension 101
4.3.1.1 Custom GUI Components 102
4.3.1.2 Custom Event Types 103
4.3.2 Cross-platform Extension 104
4.4 GUITAR in Practice 107
4.5 Summary 111

5 Empirical Evaluation 113
5.1 Research Questions and Metrics 113
5.2 Selecting & Setting Up Software Subjects 114
5.3 Defining Functions for Unique Signatures 117
5.3.1 Sandbox and Text Parameters 118
5.4 Running the Experiment 119
5.5 Threats to Validity 122
5.6 Results 123
5.7 Discussion 132
5.8 Benchmark 135
5.9 Summary 135

6 Conclusions 136
6.1 Discussion 137
6.1.1 Effectiveness 137
6.1.2 Efficiency 138
6.1.3 Limitations 139
6.2 Future Work 141

Bibliography 146

List of Tables

2.1 GUI Testing Techniques Discussed. 14
2.2 Technique Taxonomy. 15
2.3 Events available on each widget. 16

3.1 Partial Mapping. 77

4.1 Accessing GUI component information 106
4.2 Mapping the internal GUI objects 107
4.3 Performing GUI events 107
4.4 GUI platforms supported by GUITAR 108

5.1 Subject applications 116
5.2 Automated Widget/Window Identification vs. Manual Oracle. 119
5.3 Test Cases Generated and Execution Time 122
5.4 Data for RQ1 and RQ2. 124
5.5 Summary of faults detected in the iterative phases of OME* 130

List of Figures

1.1 MS Word 2010 motivating example. 4

2.1 The Radio Button Demo Application. 16
2.2 The Finite State Machine for Radio Button Demo. 19
2.3 Variable Finite State Machine. 22
2.4 FSM for the *create a new shape* responsibility. 27
2.5 Faulty Complete Interaction Sequence - dotted edges are transitions in the CIS. 29
2.6 Event Flow Graph. 31
2.7 Event Interaction Graph. 34
2.8 Event semantic interaction example. 36
2.9 Event Semantic Interaction Graph. 41
2.10 Inverse Event Sequence Graph. 43
2.11 A Task Specification. 46
2.12 AI Planning. 47
2.13 Example test cases. 51
2.14 Probabilistic Event Flow Graph with history $H = 2$. 53
2.15 2-way Covering and Covering Array. 57
2.16 Label Transition Systems. 59
2.17 Hierarchical Finite State Machine (self-loops are omitted for readability). 65
2.18 Use case diagram. 67

3.1 Running example. 71
3.2 Available Events Observed During Execution. 78
3.3 Code to collect GUI widget states. 81
3.4 A partial window hierarchy of MS Word. 86

4.1 GUITAR component architectures. 91
4.2 The *Executor* component. 95
4.3 A simple GUITAR-based testing workflow. 100
4.4 Customized component in JabRef Preferences window. 103

5.1 Continuous Integration Testing System. 121
5.2 OME* sees more of the EFG with each iteration. 129

6.1 Reaching the “Project Properties” window in DrJava. 144
6.2 Reaching the “New To Do Item” window in ArgoUML. 145
6.3 The user inactivity warning window in Rachota. 145

Chapter 1

Introduction

Graphical User Interface (GUI) is an integral component of contemporary software applications. To the end-users, the GUI is the only part of the software that they can actually "see" and interact with. A GUI abstracts away the complexity of the back-end components such as databases, communication systems and hardware. Because the only way to access the functionalities of GUI-based software is through its GUI, system testing a software often means ensuring it works properly and reliably as a whole at the GUI level. *GUI testing* is the process of executing sequences of GUI events[1] (i.e., GUI test cases) on the software's GUI front-end to reveal any possible defect.

The most prominent features of a GUI are the ease-of-use and the flexibility that it offers to both software users and software developers. Unfortunately, these features also create many challenges for software testing. Testing an application with a GUI front-end requires testers to handle (1) the enormous number of GUI components, (2) the complexity and diversity of GUI behaviors, and (3) the susceptible-to-change nature of GUI designs. Over the last decades, there have been many advances in automation for "behind-the-scenes" software testing activi-

[1]A GUI event (e.g., *click-on-Cancel-button*, *select-Radio-button*) is the action that a user performs on a widget (e.g., *Cancel button*, *Radio button*). Whenever the context is clear, we use the terms "event" and "widget" interchangeably; e.g., when we say "the user performs the *Cancel* event" we actually means that "the user performs the action *click-on* the *Cancel* button widget."

ties such as unit testing [1, 2, 3], concurrency testing [4, 5], security testing [6, 7], and performance testing [8, 9]. However, GUI testing still remains a largely *manual* and *ad hoc* process [10, 11, 12, 13]. This research aims to address the challenges in *GUI testing automation*.

Why is the automation of GUI testing so difficult? Consider how GUI-based software applications are tested today. Most often, a tester is given a set of tasks with the goal of verifying that these tasks can be performed using the software; and that the software does not "behave badly." Sometimes, the tester is also given a set of use cases with *high-level* descriptions of their steps (e.g., *"save the file"*, *"load the document"*). The tester executes these high-level steps by using GUI widgets on which events can be performed. The choice of which events to execute is most often left to the tester. For example, the tester may perform *"save the file"* in one of three ways: (1) click on the *Save* icon in the toolbar, (2) use menu items *File*→*Save*, or (3) use alternative menu items *File*→*Save_As* followed by a file name in the text-box. During this process, the tester may discover new ways to combine certain events to perform a task.

The tester does not always have a complete picture of the GUI's input space (i.e., the space of all possible GUI events and their interactions). In many software development environments, the tester is not usually supplied a *blueprint* of the GUI or its set of allowable workflows. In principle, the tester has no idea of what event sequences were missed during the testing process. End-users may execute untested event sequences and encounter failures. Moreover, the implemented GUI may allow event sequences that the designers never wanted to allow. But there is no way for a

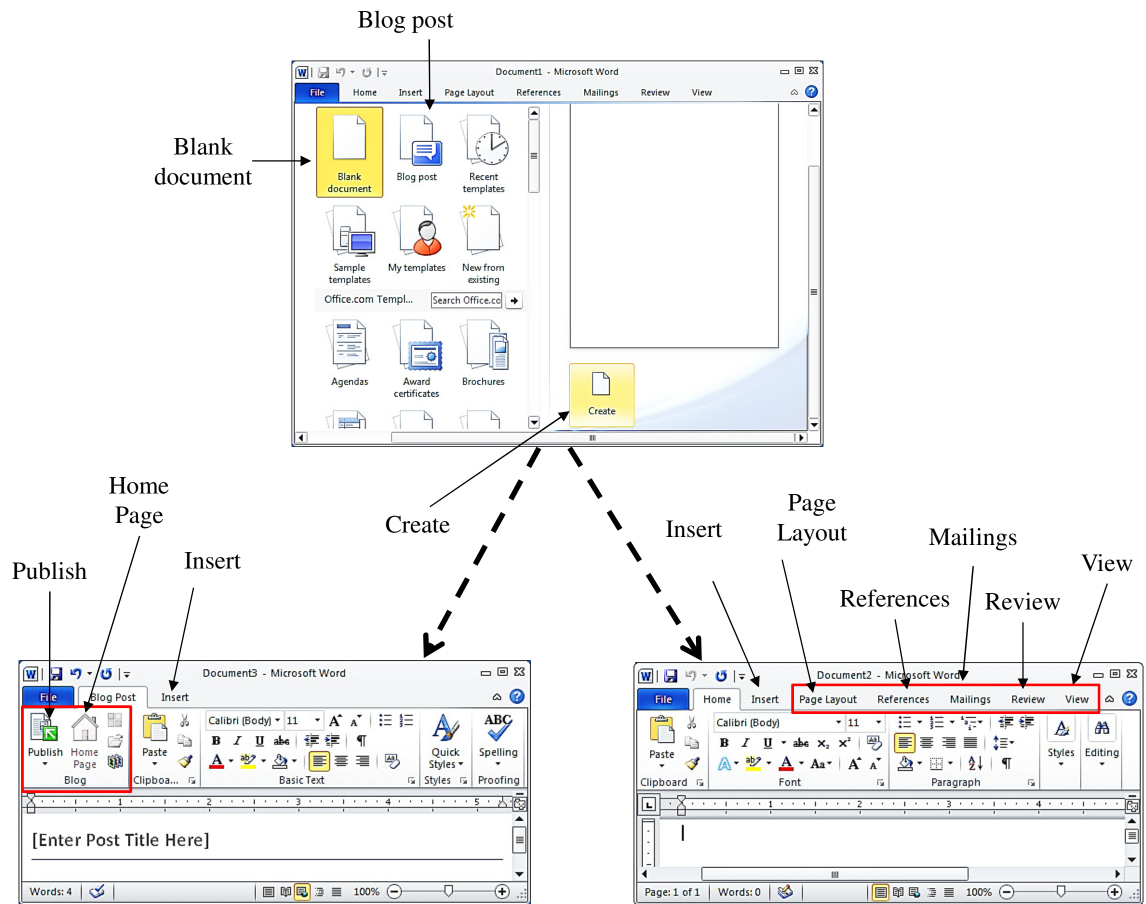
tester to determine which sequences are missed and which should not be allowed.

Human testers have the experience and domain knowledge to navigate through and verify the correctness of such systems with unknown or partially known input spaces without the aforementioned *blueprint* [10]. For example, a human tester who clicks on a button labeled *Close* expects that something (most often the current window) in the GUI to close, and all of its constituent events to become subsequently unavailable. Similarly, in MS Word 2010, Figure 1.1(a), a human tester *creating a new document* will expect the menu structure (and corresponding set of available events) to be different for a *blog* type of document versus a conventional *blank* document.

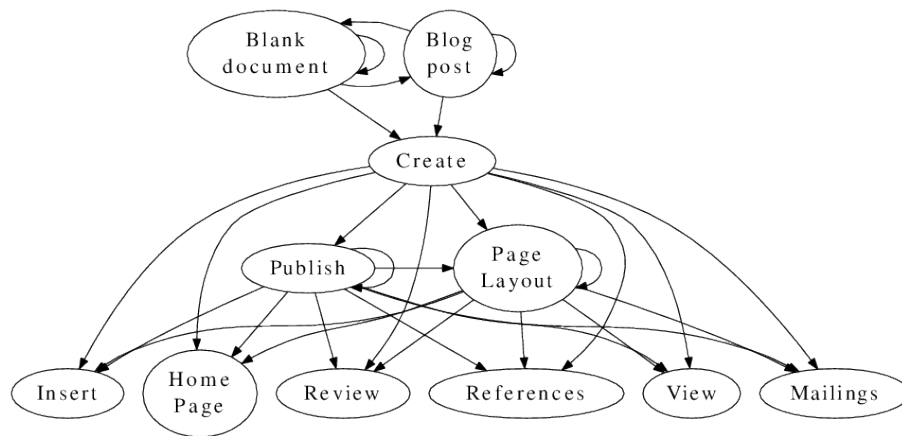
However, automated test harnesses and tools lack the experience and domain knowledge of humans. Without a representation of allowed and disallowed workflows, they are unable to reason their way out of an unexpected situation. This causes many GUI testing tools to either (1) rely on a human tester or (2) perform very limited automated testing tasks. An example of the former is the capture/replay (record/playback) tools [14] to recreate manually pre-recorded (or programatically coded) event sequences. An example of the latter is the random testing tools such as the Monkey[2] and Eclipse-based GUIdancer[3] to perform simple random walks of the user interface, execute events as encountered and detect crashes.

Recently, a new GUI testing approach called *model-based testing* [15, 16, 17] has been developed. This approach creates a graphical model of the GUI input

[2] `http://developer.android.com/guide/developing/tools/monkey.html`
[3] `http://www.guidancer.com`



(a) The *Create* button (top) is context-driven; the selected document type, either *Blank document* or *Blog post*, creates different events (bottom-left for Blog; bottom-right for regular document).



(b) The partial event-flow graph.

Figure 1.1: MS Word 2010 motivating example.

space and obtains test cases by *automatically* and *systematically* traversing the model. The most popular GUI model used is the event-flow graph (EFG) [18].

An EFG is essentially a type of GUI blueprint containing all workflows allowed by the GUI's structure and events. More specifically, it is a directed graph structure with nodes that represent events, and edges that represent the `follows` relationship. The relationship "e_x `follows` e_y" means that event e_x may be executed *immediately* after event e_y along some execution path in the GUI. This is represented as an edge from node n_x to n_y, where n_x represents e_x and n_y represents e_y. Part of the EFG for the GUI of Figure 1.1(a) is shown in Figure 1.1(b). The EFG contains edges from *Create* to *Publish*, *Home Page*, and *Insert* (in *Blog post* window) ; and to *Page Layout*, *References*, *Mailings*, *Review*, *Insert*, and *View* (in *Blank Document* window). Each of these events may be executed immediately after *Create* in the GUI.

It is not straightforward to create an EFG for a GUI. In previous work, a reverse engineering technique called *GUI Ripping* was developed to automatically create an approximation of the EFG [19]. The *GUI Ripper* dynamically traverses the GUI, opening windows, performing events, keeping track of all windows seen, and using algorithms to construct an EFG. The goal of *Ripping* is not to test the GUI's events; rather, it attempts to open as many windows as possible, extracting events from each, and computing the `follows` relationship. Because the *Ripper* performs a generalized, fully automatic traversal, it may miss application-specific parts of the GUI that are "guarded" with very specific inputs (such as a password) or behavior that requires very specific combinations of inputs, such as context-sensitive menu items. In our example of Figure 1.1(a), starting with the top-most window, the *Ripper* would select *Blank document* (because it is the first icon in the list) and then

click *Create*. This opens a new window. After the *Ripper* has finished interacting with this window and of its all sub-windows, it returns to the top window and performs all remaining events. Unfortunately, without any human intervention, the *Ripper* does not know that it needs to perform *Create* again, and in a specific context where the *Blog post* icon is selected, to reveal the bottom-right state of the main window. Hence, this window part will never be reached and events *Page Layout*, *References*, *Mailings*, *Review*, and *View* will be missed.

Existing testing approaches assume that it is straightforward to *fully* determine the GUI input space and to create a GUI model for testing. Unfortunately, this assumption does not hold true for most modern software. Today, software often has a "context-sensitive reachability GUI," in which GUI components are only reachable under some certain state and environment conditions. In our example, depending on the document options selected, clicking on the *Create* button will lead to different sets of GUI components. Another example is on the GUI of mobile devices where GUI components are often hidden and only pop up in specific senarios. With such complex and dynamic GUIs, it is not easy to determine which GUI events are available and how they are related. The current GUI testing approaches rely on an incomplete GUI input space; thus, they are unable to adequately test the GUI. As a result, GUI defects are still very common [20, 21].

1.1 Thesis Statement

This work aims to overcome the limitations in the current model creation approaches in model-based testing, specifically the GUI Ripping technique. The key idea is that the runtime information available during software execution can be valuable in understanding the structure of the GUI input space. By systematically leveraging this information, the GUI input space can be incrementally determined as the software runs.

This leads to the following thesis statement:

GUI-based applications can be effectively and efficiently tested by systematically and incrementally leveraging the application runtime execution observations.

1.2 Approach

To prove the above thesis, we develop a new paradigm *Observe-Model-Exercise** (OME*) for GUI testing. Starting with an incomplete model of the GUI's input space, a set of coverage elements to test, and test cases, OME* iteratively (1) *observes* the existence of new events during execution of the test cases, (2) expands the *model* of the GUI's input space, computing new coverage elements, and (3) obtains new test cases to *exercise* the new elements. The star in the paradigm's name represents the iterative nature of the testing process.

This work makes 5 main contributions, corresponding to 5 challenges for testing applications with a context-sensitive GUI:

Challenge 1: It is challenging to generate specific event sequences to repli-

cate context-sensitive behaviors. Because events and event sequences are context sensitive, they may have been observed due to the execution of particular prior event sequences.

Contribution 1: We make use of a new context-aware mapping that maintains information about the event sequences that were used to reach model elements.

Challenge 2: It is hard to devise new event sequences that reveal new parts of the input space and help to enhance the model without incurring significant additional cost.

Contribution 2: We simultaneously extract new GUI model elements–events and `follows` relationships–during test execution.

Challenge 3: It is not straightforward to identify new events, i.e., to determine in multiple contexts whether an event has already been seen before.

Contribution 3: We develop a unique signature for each widget and matching heuristics to help detect new widgets.

Challenge 4: It is complex to incrementally make changes to the model to add new elements.

Contribution 4: We develop new operations on the EFG$^+$ to incrementally enhance the model as new information becomes available.

Challenge 5: It is challenging to incrementally generate new test cases.

Contribution 5: We develop an algorithm to compute new test requirements from recent model enhancements and generate test cases to satisfy the requirements.

Our final contribution is our experiment, involving 8 open-source applications on which we executed over 400,000 test cases and consumed almost 1000 machine

days. We compared OME* with the current state-of-the-art. We saw significant improvements, both in terms of new areas of the input space that we explored, and fault detection – we fully automatically detected 34 new faults in these applications.

1.3 Related Work

The *GUI Ripping* technique proposed in 2003 [19] set the stage for using the executing GUI software itself to *automatically* model its own input space. In summary, the *Ripper* starts at the *main* window of a software application under test, automatically detects all 'clickable' GUI widgets and exercises the application by systematically executing these elements. The GUI structure obtained is then converted to an EFG, which is subsequently used to systematically generate test cases. Since then, several techniques have been developed to augment the EFG model using annotations. For example, Yuan et al. [22] annotated the EFG with *semantic* information derived from the runtime state of the GUI. However, these approaches were unable to discover unexplored parts of the input space.

A technique called *exploratory testing* [5, 10] is closest to our current approach. In exploratory testing, human testers manually explore the system under test without fully knowing its input space. As the system is being tested, they learn the system's behaviors and manually decide what to do next. There is no predetermined test script or test input. This technique takes advantage of the testers' experiences and provides rapid feedback to the developers. However, because it relies heavily on human skills, the results are often *subjective*, *hard to replicate* [23], and *difficult*

to apply to large systems [13].

Extending the work on GUI ripping, Mesbah et al. [24] leverage a *crawl-based* technique to reverse engineer the structure of a website under test. A tool called Crawljax automatically detects all 'clickable' web elements and crawls the website by exercising these elements. The website structure is then analyzed to construct an intermediate abstract state machine model, which is used as a skeleton to systematically generate test cases. Saxena et al. [6] extend this technique by adding a string constraint solver to the crawler to better explore the event space.

The remainder of the related work we discuss here shares a common theme. During test execution, these approaches keep track of all new event handlers, object states and web services observed during execution and try to generate additional test cases to exercise these new elements.

In *web application testing*, Artzi et al. [25] use execution states to generate additional test inputs. Due to the nature of the web applications, the event handlers can be dynamically registered to and removed from the client at runtime. An execution unit dynamically monitors the set of events registered at a particular time and attempts to exercise them.

In *object-oriented unit testing*, Dallmeier et al. [26] dynamically synthesize a state-machine model by monitoring the object states in different executions. As test cases are executed, new object states are observed and incrementally incorporated into the original model. The extended model is then used to generate additional test cases until some stopping criterion is met. To further enhance the model, a source code scanning technique is used to extract all available method calls. The

methods are invoked from all obtained states in a trial-and-error process, to reveal possibly unobserved class behaviors. Zhang et al. [9] present a similar approach but use advanced static analysis techniques to infer the constraints between method calls and their arguments. The constraints help to avoid generating illegal test cases as well as to direct test case generation toward unexplored program behaviors to achieve higher code coverage. At the system level, Walkinshaw et al. [27] propose a similar approach but for *embedded system testing*.

In *service oriented application (SOA)* testing, Bartolini et al. [28] introduce an approach to "whiten" the binary services from the external, third-party libraries to support testing the service consumers. An intermediate agent is added to the services through instrumentation to expose their coverage level as the test cases are executed. Based on the data collected, the test case generator can infer the internal behaviors of the services and decide how to generate test cases for the service consumers.

Our work differs from the above approaches in several ways. First, our target domain is that of GUIs, which have enormous, possibly infinite, input spaces. GUI applications increasingly integrate multiple source code languages and object code formats, along with virtual function calls, reflection, multi-threading, and event-handler callbacks. These features severely impair the applicability of techniques that rely on static analysis or the availability of platform-specific and format-specific instrumentation tools. Second, we have a fully automated and scalable technique. Our underlying model is an EFG rather than a state machine, which, for reasons discussed in prior work [29] are more appropriate for this domain. Third, our model

enhancement is based on new events, not states. In other words, we are extending the input alphabet which is often considered unchanged in the state machine models. Finally, most of the above approaches rely on code instrumentation. Our approach, in contrast, does not require any knowledge of intermediate binary code or source code.

1.4 Research Scope

GUIs may be used as front-ends to any different types of software applications. Thus, the number of all possible GUIs is enormous. It would be extremely difficult to create a universal testing approach to work with all possible types of GUIs.

In this dissertation, to provide focus, we only consider a sub-class of GUIs. In particular, the GUIs of our interest react to events performed only by a single user; the events are deterministic, i.e., their outcomes are completely predictable. Testing GUIs that react to temporal and non-deterministic events is beyond the scope of this research.

1.5 Dissertation Outline

The remainder of this dissertation is structured as follows. Chapter 2 provides a background review and discusses the existing related work in more detail. Then we present the formal models and algorithms to realize the OME* testing paradigm (Chapter 3). We describe our experimentation tool implementation (Chapter 4) and an empirical evaluation for OME* (Chapter 5). Finally, we conclude with a discus-

sion of our testing approach and provide directions for future work (Chapter 6).

Chapter 2

Background

The testing approaches presented in this work belongs to the family of *model-based* techniques for testing GUI-based applications. This chapter presents a survey on existing model-based GUI testing techniques to provide a research context for this work. The techniques surveyed are summarized in Table 2.1.

In summary, all techniques require the creation of a model of the software or its GUI, and algorithms to use the model to generate test cases. The techniques of interest to us employ 6 distinct models, shown in Column 1 of Table 2.1; the "hierarchical" model uses a combination of these models organized in a hierarchy.

Model	**Technique**	**Abbreviation**	**Section**
State machine	Finite State Machine	FSM	2.2.1.1
	Variable Finite State Machine	VFSM	2.2.1.2
	Complete Interaction Sequence	CIS	2.2.1.3
	Faulty Complete Interaction Sequence	FCIS	2.2.1.4
Workflow	Event Flow Graph	EFG	2.2.2.1
	Event Interaction Graph	EIG	2.2.2.2
	Feedback based	ESIG	2.2.2.3
	Faulty Event Sequence Graph	FESG	2.2.2.4
Pre- Post-condition	AI Planning	AI	2.2.3
Event sequence	Genetic Models	GA	2.2.4
Probabilistic	Probabilistic Event Flow Graph	PEFG	2.2.5
Combinatorial	Latin Squares	LS	2.2.6.1
	Coverage Arrays	CA	2.2.6.2
Hierarchical	Keyword-driven Model	KW	2.2.7.1
	Hierarchical Finite State Machines	HFSM	2.2.7.2
	UML-Diagram Based	UML	2.2.7.3

Table 2.1: GUI Testing Techniques Discussed.

There are two important aspects of each technique that we discuss. First is the model that it employs. In some cases, the models are created manually; in

others, they are derived in an automated manner. The second important aspect is the test-case generation approach, which, for some techniques is manual; but for most is automated. Table 2.2 shows the set of techniques discussed in this chapter, partitioned along two dimensions: (model creation {*manual, automated*}, test generation {*manual, automated*}).

Model creation / **Test generation**	Manual	Automated
Manual	FSM, VFSM, CIS, FCIS	—
Automated	KW, FESG, AI, GA, PEFG, LS, CA, HFSM, UML	EFG, EIG, ESIG

Table 2.2: Technique Taxonomy.

The remainder of this chapter presents these techniques. But first, we present a small GUI application, that we use as a running example. This running example is used to illustrate the important aspects of each technique, and its relative strengths and weaknesses.

2.1 Running Example

The simple running example application called "`Radio Button Demo`" is seen in Figure 2.1. The GUI contains 9 widgets labeled w_0 through w_8. A user can perform events on almost all the widgets (there is no event available on w_4). Table 2.3 shows the events associated with each widget. We note that in this simple example, each widget has at most one associated event. In a more complex GUI, a widget may have multiple associated events.

The application's functionality is very straightforward – the *initial state* has

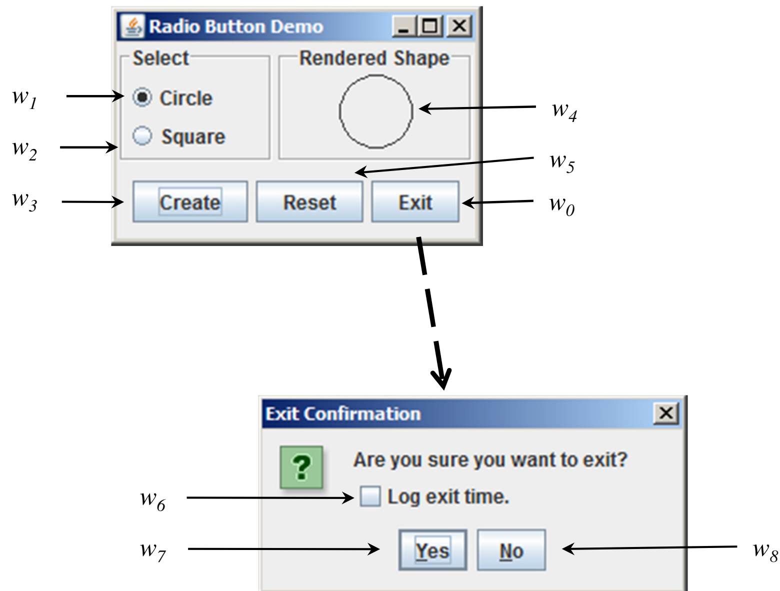


Figure 2.1: The Radio Button Demo Application.

Widget	Event name
w_1	*circle*
w_2	*square*
w_3	*create*
w_5	*reset*
w_0	*exit*
w_6	*(un)check*
w_7	*yes*
w_8	*no*

Table 2.3: Events available on each widget.

`Circle` (corresponding to w_1) selected, the `Rendered Shape` area (widget w_4) is empty and the `Reset` button is disabled. Events are used to change the state of the GUI. Event *circle* sets the radio button setting to circle; if there is already a square in the `Rendered Shape` area, then the shape is immediately changed to a circle. Event *square* is similar to *circle*, except that it changes the shape to a square. Event *create* creates a shape in the `Rendered Shape` area according to current settings of w_1 and w_2. Event *reset* resets the entire software to its initial state. This event is only available when there is an existing shape. Event *exit* opens a modal "`Exit`

`Confirmation`" window that contains widgets w_6, w_7, and w_8. This window blocks all widgets in the main window when open. Event *(un)check* changes the status of the check-box w_6 (originally unchecked) so that when it is checked the exiting time will be logged to a file before the application is terminated. Event *no* closes the window and moves focus back to the main window; and event *yes* closes the entire application.

The GUI of this application is simple, yet quite flexible. The numbers of 1-, 2-, 3-, 4-, and 5-way unique event sequences (and hence possible test cases) that may be executed in the initial state of the GUI are 4 (remember that the `Exit Confirmation` window is initially closed and w_5 is disabled), 17, 66, 253, and 798 respectively.

2.2 Test Case Generation Techniques

This section presents an overview of all the techniques listed in Table 2.1. The techniques are classified according to the underlying model used.

2.2.1 State Machines

Because GUIs are composed of objects (i.e., the widgets) that maintain state, in terms of widget-properties (e.g., *Enabled*, *Caption*, *Width*) and their values (e.g., *TRUE*, "*Cancel*", 20), many researchers have found it natural to model GUIs using state machines [30, 31, 32, 33]. For example, the GUI of Figure 2.1 starts in an "initial state" in which, among other widgets, widget w_3 is not selected and

w_5 is disabled. If one were to model the state of the GUI as a set of triples $(widget, property, value)$, the initial state could be represented as $\{\ldots,(w_3, Selected, FALSE), (w_3, Enabled, TRUE), (w_5, Enabled, FALSE), \ldots\}$. As can be imagined, depending on how one models the state, such machines can get extremely large for non-trivial GUIs. In this section, we present several techniques that researchers have employed to control this state space explosion. Esmelioglu et al. [30] use constraints, Shehady et al. [31] use global variables, White et al. [32] focus on a part of the state machine, and Belli et al. [33] develop off-nominal test cases. We present these techniques next.

2.2.1.1 Finite State Machines

In this section, we present details of the approach taken by Esmelioglu et al. [30], who model the GUI as a finite state machine (FSM). Formally, a FSM can be represented as a quintuple $FSM = (S, I, O, T, \Phi)$, where S is the finite set of GUI states, I is the set of inputs, i.e., events that may be performed on the GUI, O is the finite set of outputs, T is the transition function $S \times I \rightarrow S$ that specifies the next state as a function of the current state and input event, Φ is the output function $S \times I \rightarrow O$ that specifies the resulting output from a transition.

For GUI testing, a tester is free to select certain aspects of the software to model in the state. For example, we choose to represent the state of the GUI using 4 of its elements: (1) *log*, which is 1 if w_6 is checked, 0 otherwise; (2) $exitWinOpen$, which is 1 if the `Exit Confirmation` window is open, 0 otherwise; (3) *created* which

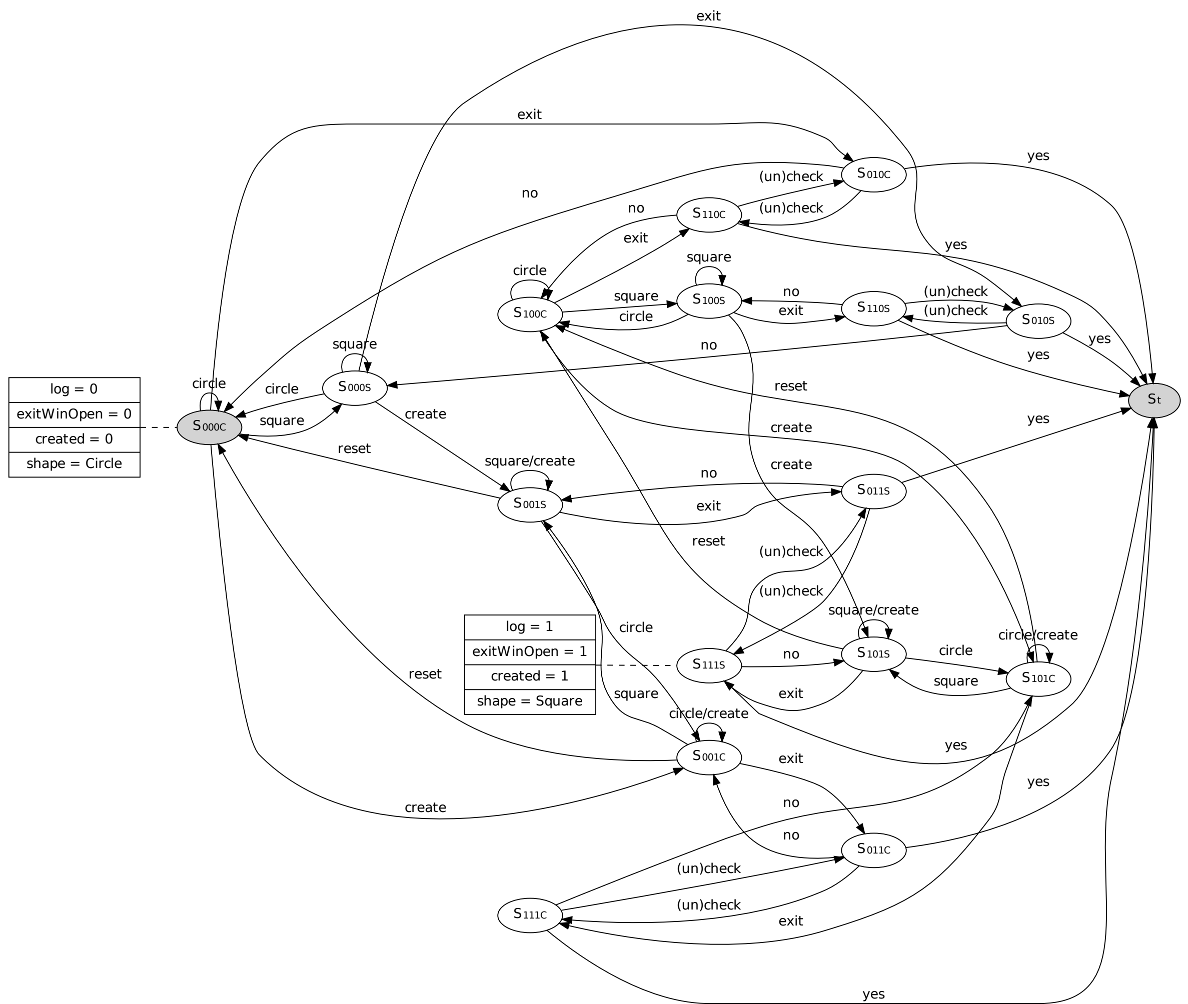


Figure 2.2: The Finite State Machine for Radio Button Demo.

is 1 if a shape is created, 0 otherwise; (4) *shape* which is either *Circle* or *Square*.

We can then represent the state of the GUI using a length 4 vector consisting of the above 4 elements in the order specified above. For example, state S_{000C} is the initial state in which w_6 is unchecked, the exit confirmation window is closed, no shape is created at w_4, and the shape radio button for circle is set. Similarly, S_{111S} is the state in which w_6 is checked, the exit confirmation window is open, a shape is visible at w_4, and the shape radio button for square is set.

We use the above definition of GUI state to create an FSM. Figure 2.2 shows the FSM of the GUI of Figure 2.1. Nodes in the graph represent states and edges represent transitions; there are two special states (shaded nodes) in the FSM: the initial state right after the software starts (S_{000C}), and the terminal state when the software has been terminated (S_t). Some of the state transitions are as follows: If the `Create` button has never been clicked, then the user can transit between S_{000x} states by selecting different radio button options (x represents any value of the corresponding state element, in this case x is either C or S). Once `Create` has been clicked, the GUI transits to a new state where the third state element turns from 0 is 1 (i.e., a new shape has been created). The user can transit back and forth between S_{x0xC} and S_{x0xS} by selecting different radio button options (x represents any value of the corresponding state element). However, the user cannot do the same for the pair (S_{01xC}, S_{01xS}) or S_{11xC} states because the `Exit Confirmation` window blocks all widgets in the main window.

Once the FSM has been created, test case generation from an FSM is very intuitive. The test designer may start at the initial state, traverse edges of the FSM as desired and record the transitions as events. For example, in Figure 2.2, a test case could be: $\langle$*square*, *circle*, *create*, *exit*, *(un)check*, *yes*$\rangle$ which takes the software through states S_{000S}, S_{000C}, S_{001C}, S_{011C}, S_{111C}, and S_t.

Although FSMs are easy to create, they suffer from some major problems. First, they do not scale for large GUIs. Moreover, the states may not have any relationship to the structure of the GUI. Hence they can be difficult to maintain. A new model called variable finite state machines (VFSMs), developed by Shehady et

al. [31], presented next, attempts to rectify some of these problems.

2.2.1.2 Variable Finite State Machines

Shehady et al. [31] use Variable Finite State Machines (VFSMs) for testing GUIs. The key difference between VFSMs and FSMs is that VFSMs allow a number of global variables, each of which takes values from a finite domain. The values of the variables are used to compute the next state and the output in response to an input. Transitions may also modify the values of these variables. In principle, the space of GUIs that can be modeled using VFSMs is the same as those that can be modeled using FSMs.

Formally, a VFSM is represented as a 7-tuple $VFSM = (S, I, O, T, \Phi, V, \zeta)$, where S, I, O are similar to their counterparts in FSMs, $V = \{V_1, V_2, V_3, \ldots, V_n\}$ (each V_i is the set of values that the ith variable may assume) and n is the total number of variables in the VFSM. Let $D = S \times I \times V_1 \times V_2 \times \ldots \times V_n$ and $D_T \subseteq D$; T is the transition function $D_T \rightarrow S$ and Φ is a function $D_T \rightarrow O$. Hence the current state of each of the variables affects both the next state and the output of the VFSM. ζ is the set of variable transition functions. At each transition, ζ is used to determine whether any of the variables' values have been modified. Each variable has an initial state at startup.

Figure 2.3 shows an VFSM of the `Radio Button Demo`'s GUI. The VFSM is much smaller than the corresponding FSM (Figure 2.2) because the states have been simplified. Each state is simply represented by a length 3 vector, i.e., that specifies

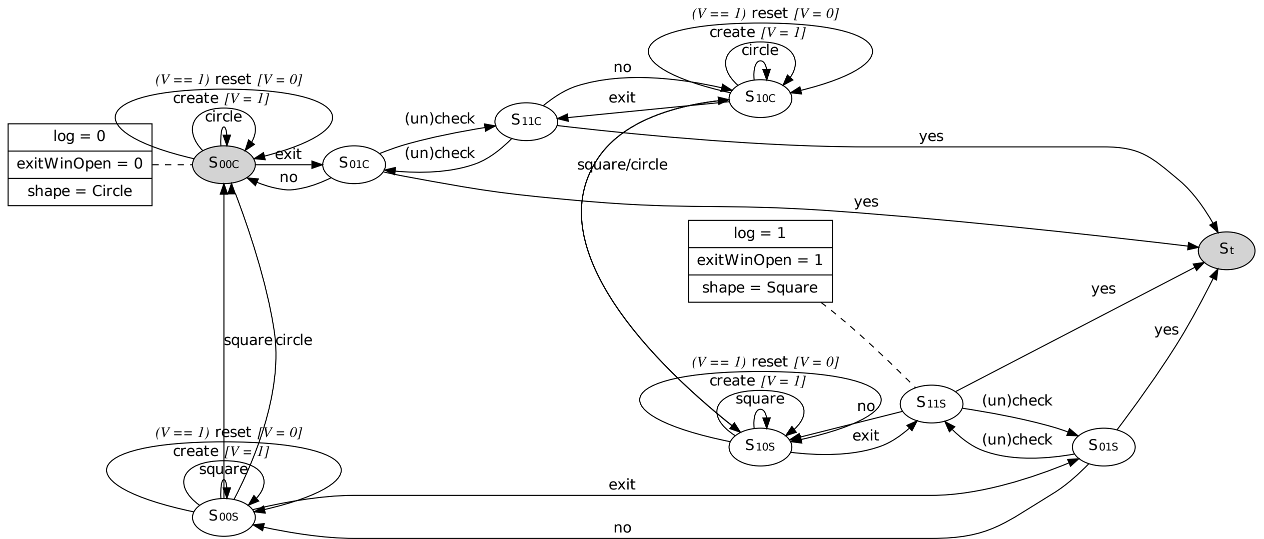


Figure 2.3: Variable Finite State Machine.

whether *log* needs to be maintained, the `Exit Confirmation` window is open, and the type of shape that has been selected.

The states have been simplified because the element *created* has been removed from the state. This information is now maintained in a variable V that can take values 0 and 1. Edges of the VFSM are annotated with predicates (shown in parenthesis placed before the edge label) and assignments to the variables (shown in square brackets placed after the edge label). Initially, V is set to 0. Transitions are taken depending on the outcome of the predicates. For example, the *reset* transition is taken from S_{00C} only if $V == 1$; once taken, it changes V to 0. Similarly, *create* changes V to 1.

The VFSM created is much more concise (it has 9 states) than the original FSM in Figure 2.2 (which has 17 states). This is because several states in the FSM are grouped and represented by a single state in the VFSM. VFSMs can be converted into their equivalent FSMs for test case generation. The key idea is to fold the

information of V and ζ into S and T. Given a VFSM's S and $V = \{V_1, V_2, \ldots, V_n\}$, the new FSM's set of states S_{eq} is obtained as $S_{eq} = \{S_i | S_i \in S \times V_1 \times V_2 \times V_3 \times \ldots \times V_n\}$, i.e., this creates a set of states that combines the information of the states and the variables into one state. Similarly, the new FSM's transition function $T_{eq} : S_{eq} \times I \rightarrow S_{eq}$ may be created by combining the T and ζ functions of the VFSM. Since the range of T is S and the range of ζ is $V = \{V_1, V_2, \ldots, V_n\}$, S_{eq} is the Cartesian product of the two ranges; also T and S have the same domain.

2.2.1.3 Complete Interaction Sequences

Another approach to restrict the state space of a state machine is by employing software usage information. The method proposed by White et al. [32] solves the FSM's state explosion problem by focusing on a subset of interactions performed on the GUI. They key idea is to identify *responsibilities* for a GUI; a responsibility is a GUI activity that involves one or more GUI objects and has an observable effect on the surrounding environment of the GUI, which includes memory, peripheral devices, underlying software, and application software. For each responsibility, a *complete interaction sequence (CIS)*, which is a sequence of GUI objects and selections that will invoke the given responsibility, is identified. Parts of the CIS are then used for testing the GUI.

The GUI testing steps for CIS are as follows.

1. Manually identify responsibilities in the GUI.

2. For each responsibility, identify its corresponding CIS.

3. Create an FSM for each CIS.

4. Apply transformations to the FSM to obtain a *reduced FSM*. These transformations include the following.

    (a) Abstracting strongly connected components into a *superstate*.

    (b) Merging CIS states that have structural symmetry.

5. Use the reduced FSM to test the CIS for correctness.

The two abstractions mentioned above (Steps 4a and 4b) are interesting from a modeling point of view. They are described in more detail next.

Definition: A part of a FSM, called a *subFSM*, is a *strongly connected component* if for every pair (S_1, S_2), $S_1, S_2 \in S$, there exists a directed path from S_1 to S_2. Each such component is then replaced by a *superstate* and tested in isolation.

A subFSM has structural symmetry if the following conditions hold.

1. it contains states S_1 and S_2 such that S_1 has one incoming transition, S_2 has one outgoing transition, and a number of paths reach S_2 from S_1;

2. for each path in the subFSM, context (the path taken to get to S_1 from outside the subFSM) has no effect on the states/transitions or output;

3. no transition or state encountered after S_2, is affected by paths taken inside the subFSM.

Such a subFSM can be reduced into a superstate and tested in isolation.

Given a GUI, the test designer first reduces the FSM after applying the above transformations, thereby reducing the total number of states in the FSM. This results in smaller number of paths in the FSM, hence reducing the number of test cases. Without any loss of generality, each FSM is assumed to have a distinct start state and distinct terminating state.

As was the case before, a test is a path through the FSM. The test designer creates two types of tests: *design tests* that assume that the FSM is a faithful representation of the GUI's specifications, and *implementation tests* that for each CIS, consider the possibility that potential transitions not described in the design may occur in the implementation.

For design tests, the test designer creates sufficient number of tests starting at the initial state and ending at the termination state so that the following conditions hold:

- all distinct paths in the reduced FSM are executed; each time a path enters a superstate corresponding to a component, an appropriate test path of the component is inserted into the test case at that point,
- all the design subtests of each component are included in at least one test, which may require additional tests of the reduced FSM to satisfy this constraint.

The key idea of conducting implementation testing is to check all GUI events in the CIS to determine whether they invoke any new transitions in the reduced FSM. To implement test the reduced FSM, the test designer must construct sufficient test

sequences at the initial state and stopping at the terminal state so that the following conditions hold:

- all the paths of the reduced FSM are executed, and
- all the implementation tests for each remaining component are included at least once.

By using the CIS concept, the test designer can test a GUI from various perspectives, each defined by the CIS. These CIS can also be maintained in a library to be reused across various GUIs.

For example, in the `Radio Button Demo` application, the tester may design a *"create a new shape"* responsibility that involves 4 objects w_1, w_2, w_3, and w_5 (assuming that the `Rendered Shape` area is empty and the `Exit Confirmation` window is not opened). Figure 2.4 shows an FSM for this responsibility where each node represents a GUI state and each edge represents a state transition. Note that the states in this FSM are abstract states representing several states in the FSM in Section 2.2.1.1. For example, S_{x00C} is an abstraction of all states where the `Circle` radio button is selected (x can be replaced by any value of the corresponding state element).

The subFSM consisting of the two states S_{x00C} and S_{x00S} is a strongly connected component. Thus, this subFSM can be tested in isolation and then replaced it by a superstate S_{x00x} (i.e., a shape is selected). To test the internal behaviors of the subFSM, the state sequence needed to be covered is $\langle S_{x00C}, S_{x01S} \rangle$; which is obtained by the event sequence $\langle$*square*, *circle*$\rangle$. With an assumption that the sub-

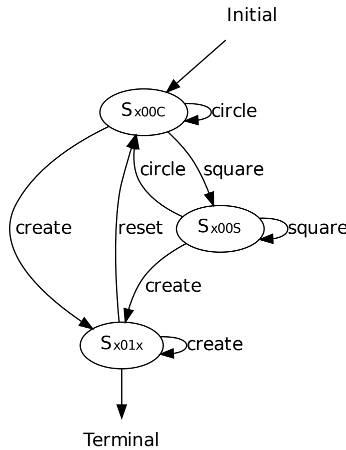


Figure 2.4: FSM for the *create a new shape* responsibility.

FSM is well tested, state sequence needed to test the reduced FSM is ⟨*Initial*, S_{x00x}, S_{x01x}, S_{x00x}, S_{x01x}, *Terminal*⟩. This sequence is then translated to an executable test case taking the GUI from the initial state to the terminal state: ⟨*create*, *reset*, *create*⟩.

2.2.1.4 Off-nominal Finite State Machines

The three approaches discussed thus far generate test cases to test the GUI for legal event sequences specified in the state machine model. However, the GUI might have been coded incorrectly to allow other sequences left unspecified in the state machine. For example, in our `Radio Button Demo` GUI, does the GUI allow the user to click on the *reset* button when the application is launched, or after *reset* has been executed once? For example, is the sequence $\langle reset, reset, reset\rangle$ allowed?

The implicit assumption is that such off-nominal sequences are illegal and should not be allowed by the GUI. Belli et al. [33] argue that these sequences should also be tested in addition to the legal sequences. They augment the *Complete In-*

teraction Sequence approach to test the GUI system's robustness by generating such off-nominal test cases. The augmented model is called *Faulty Complete Interaction Sequence (FCIS)*.

As was the case for the CIS, each FCIS can be specified by an FSM. This FSM is constructed by the following steps:

1. Build the CIS and the corresponding FSM consisting of all legal sequences of user-system interactions. Each edge of the FSM is called an *Interaction Pair (IP)*.

2. Identify Faulty Interaction Pairs (FIPs) consisting of inputs that are not legal. These are all the "missing" IPs in the original FSM. Note that FIPs and IPs together define a complete FSM called the *Complete Finite State Automata* (CFSA).

Figure 2.5 shows an FSM of the FCIS corresponding to the CIS in Figure 2.4. The solid lines in the graph represent the FIPs and the dotted lines represent the edges in the CIS's FSM.

Test case generation for a FCIS is straightforward. The tester can systematically design test cases for various undesired system behaviors by covering all possible FIPs. One way to do this is to select an untested FIP, i.e., an edge in the FCIS, generate a sequence of events from the FSM's start state to the first event in the selected edge, and prepend this sequence to the edge, creating a test case that will test the selected FIP. Once this is done for each FIP, all of them would be tested and covered.

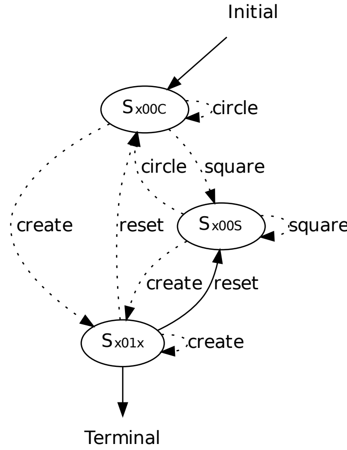


Figure 2.5: Faulty Complete Interaction Sequence - dotted edges are transitions in the CIS.

As we can see in Figure 2.5, there is one FIP in the FSM: $\langle S_{x01x}, S_{x00S} \rangle$. By prefixing this FIP with the state sequence $\langle \mathit{Initial}, S_{x00C}, S_{x01x} \rangle$, we get a complete sequence in the CFSA to examine the illegal behavior: $\langle \mathit{Initial}, S_{x00C}, S_{x01x}, S_{x00S} \rangle$. The sequence can be translated to a test case which is a sequence of events starting at the initial state: $\langle \mathit{create}, \mathit{reset} \rangle$.

2.2.2 Workflows

Some researchers have used the GUI's business *workflow*, i.e., a sequence of connected steps, for test case generation. A typical GUI workflow is represented by a set of *events* (the steps) and some type of sequencing relationship between the events. In this section, we describe the Event Flow Graph model [34], a seminal work in this category. Then, we present two variants of this model: the Event Interaction Graph [35] and the Event Semantic Interaction Graph [22]. Finally, we discuss the Faulty Event Sequence Graph [33], an off-nominal model for the workflow-based

approach.

2.2.2.1 Event Flow Graph

Intuitively, an Event Flow Graph (EFG) represents all possible event sequences that may be executed on a GUI [34]. The graph nodes represent events in the GUI and the graph edges represent a sequencing relationship that shows the set of events events that may be performed immediately after a given event. The concept of the EFG is similar to that of a control-flow or program-flow graph [36] that capture the flow of all possible executions of program statements, except that an EFG represents the flow of events, not code, in a GUI.

Definition: An EFG for a GUI G is formally defined as a triple <**V**, **E**, **B**> where:

1. **V** is a set of vertices representing all the events in G. Each $v \in \mathbf{V}$ represents an event in G.
2. $\mathbf{E} \subseteq \mathbf{V} \times \mathbf{V}$ is a set of directed edges between vertices. Event e_j `follows` e_i (or equivalently $e_j = \texttt{follows}(e_i)$) iff e_j may be performed immediately after e_i. An edge $(v_x, v_y) \in \mathbf{E}$ iff the event represented by v_y `follows` the event represented by v_x.
3. $\mathbf{B} \subseteq \mathbf{V}$ is a set of vertices representing *initial events* of G that are available to the user when the GUI is first invoked.

The EFG for the `Radio Button Demo` application is shown in Figure 2.6. The events are shown as oval nodes. The shaded nodes are *initial events*, i.e., they are available to the user when the GUI is first launched. The directed edges show the `follows` relationship between events. For example, a user can click on the `Yes` button in the `Exit Confirmation` window either immediately after clicking on the `Exit` button or immediately after clicking on w_6; hence there is an edge from *exit* to *yes*, and from (*un*)*check* to *yes*. The user cannot click on the `Yes` button after the `No` button because *no* closes the dialog; there is no edge from *no* to *yes*. Similarly, there is no edge from *no* to *no*; nor is there an edge from *yes* to *yes*.

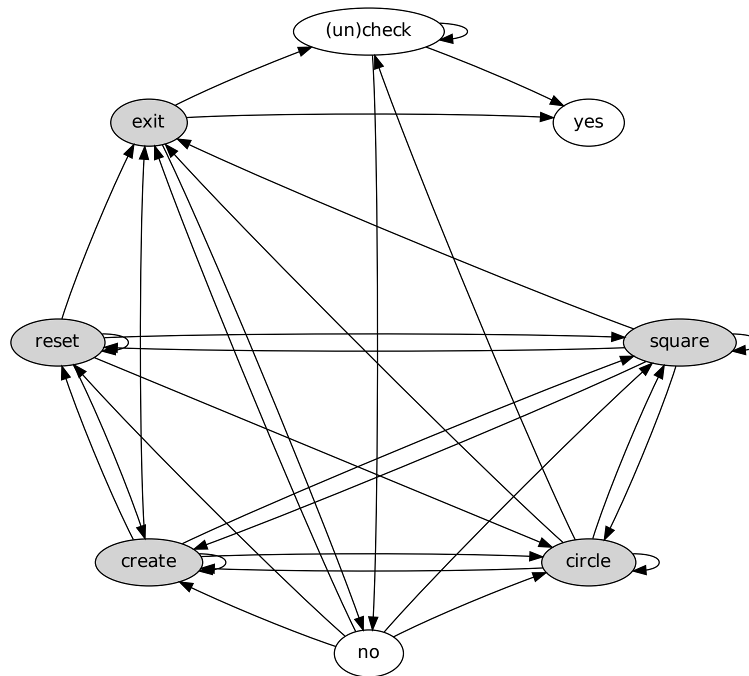


Figure 2.6: Event Flow Graph.

An approximation of the EFG for a GUI can be automatically obtained by a reverse engineering process call GUI ripping [19]. All events available in the GUI are automatically performed to open the hidden widgets and windows in a depth-

first manner. During the GUI ripping process, the key attributes of each widget are captured (e.g., whether it opens a modal/modeless window, it opens a menu, it closes a window). These attributes are then used to automatically construct the EFG. Because such a process is unable to infer complex state-based relationships between events, e.g., one enables/disables the other, a tester has to manually check and edit it to obtain the final EFG.

Because the EFG captures all possible sequences of events that may be executed by a user on the GUI, any path in the EFG is a valid user-executed event sequence, and hence, a potential test case. Moreover, any graph traversal technique on the EFG can be used to yield test cases. Examples of some techniques that have been used in the past are *goal-directed search* [37], *random-walk* [35], and *bounded breadth-first search* [38]. For example, a random walk of the EFG of Figure 2.6 may yield the test case $\langle square, square, circle, square, create, reset, exit, yes \rangle$.

2.2.2.2 Event Interaction Graph

Because the EFG captures all possible event sequences that may be executed on the GUI, the number of event sequences that may be generated from an EFG becomes extremely large. In fact, the number grows exponentially with sequence length [39, 34]. It is important to reduce this number for practical reasons. To address this issue, Xie et al. [38] conducted several empirical studies on the characteristics of test cases derived from the EFG model. The experiments showed that a large number of faults were detected by the test cases that tested interactions between certain type

of events which (1) close a modal window (*termination events*) or (2) interact with the underlying code (*system-interaction events*). Other events used to manipulate the GUI structure such as open or close menu/modeless windows, called *structural events*, are unlikely to reveal faults. One possible explanation for these results was that the code for structural events is usually simple and generated automatically by visual GUI-building tools; therefore it is less likely to be faulty. Based on these results, a new model called the Event Interaction Graph (EIG) was developed.

Intuitively, an EIG contains only *termination* and *system-interaction* events; an edge between two nodes in the EIG shows that one event might be executed after (not necessarily immediately after) the other along some execution path. Formally, EIG edges are defined by an `interacts-with` relation through the following definitions:

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

Definition: An event-flow-path $< n_1; n_2; \ldots; n_k >$ is *interaction-free* iff none of $n_2, \ldots, n_{k-1}$ represent termination or system-interaction events.

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

The EIG edges actually represent the above *interacts-with* relationship between

events. An EFG can be automatically transformed into an EIG by using graph-rewriting rules (details are presented in [40]). The EIG for the `Radio Button Demo` application is shown in Figure 2.7. Note that the EIG does not contain the window-opening *exit* event. The graph-rewriting rule used to obtain this EIG was to (1) delete *exit* because it is a window-open event, (2) for all remaining events e_x replace each edge (e_x,*exit*) with edge (e_x, e_y) for each occurrence of edge (*exit*, e_y), and (3) for all e_y, delete all edges (*exit*, e_y).

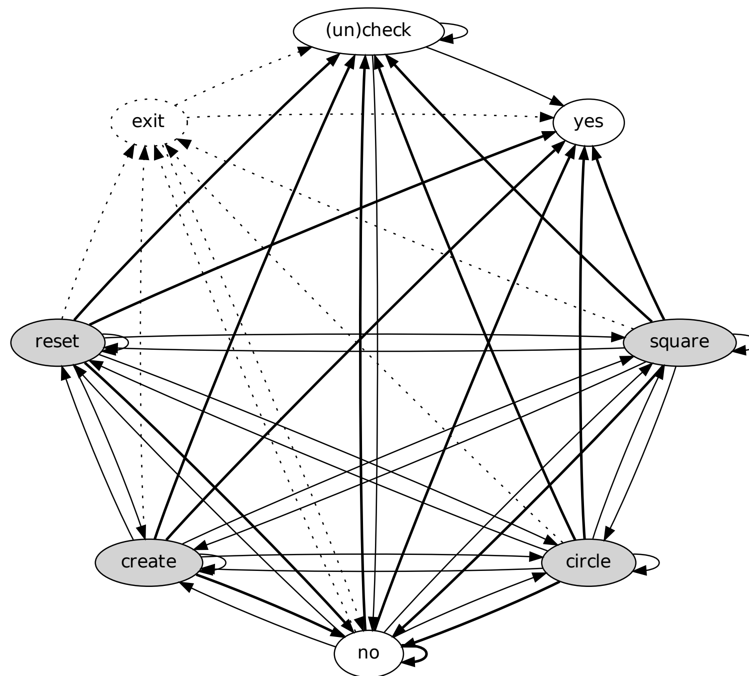


Figure 2.7: Event Interaction Graph.

As was the case with EFGs, a test case in the EIG model is also a path in the EIG, starting with an initial event. One possible test case might be $\langle square, square, circle, yes\rangle$. Because EIG nodes do not represent events to open or close menus/windows, the sequences obtained from the EIG may not be executable. For example, the test case $\langle square, square, circle, yes\rangle$ will not execute because *yes* is not available for execu-

tion after *circle*. For that reason, at execution time, other events needed to reach the EIG events are automatically inserted using the original EFG. During the test-case execution, the EIG test case above will be expanded to $\langle square, square, circle, exit, yes \rangle$

2.2.2.3 Event Semantic Interaction Graph

Although the EIG model is smaller than the EFG, it is still a dense graph and suffers from the same problems as does the EFG – the number of generated event sequences grows exponentially with length. In more recent work, Yuan et al. [22] create a sparse graph, where events are connected by edges only if they were shown to influence each other's execution behavior. Consider the `Radio Button Demo` example. The top-left GUI in Figure 2.8 shows the *initial state* (S_0) of the application. After an event *square* is executed, the GUI changes its state to the one shown in the top-right ($square(S_0)$). In this state, the `Square` radio button is selected. Starting from S_0, one can execute another event (*create*) and obtain the state shown in the bottom-left ($create(S_0)$); a circle is created by clicking the `Create` button. If, however, the sequence $\langle square; create \rangle$ is executed in S_0, a new state ($create(square(S_0))$), shown in the bottom-right is obtained; a square has been created. This execution is equivalent to the execution of event *create* in the state $square(S_0)$. The event *square* clearly influences the event *create*. We say that event *square* "interacts with" event *create*, and should be tested together to check for interaction problems.

The main idea behind observing GUI run-time states and using them to determine which events to test together can also be justified by examining the code

of event handlers. For example, the event handlers for *square* and *create* share two variables `created`, which indicates if a shape is created, and `currentShape`, which specifies the current selected shape; *create* sets `created` to `TRUE` and influences *square*'s flow of control; *square* sets `currentShape` to a square, which *create* uses as a parameter to create a shape; hence it's not surprising that they influence each other's execution.

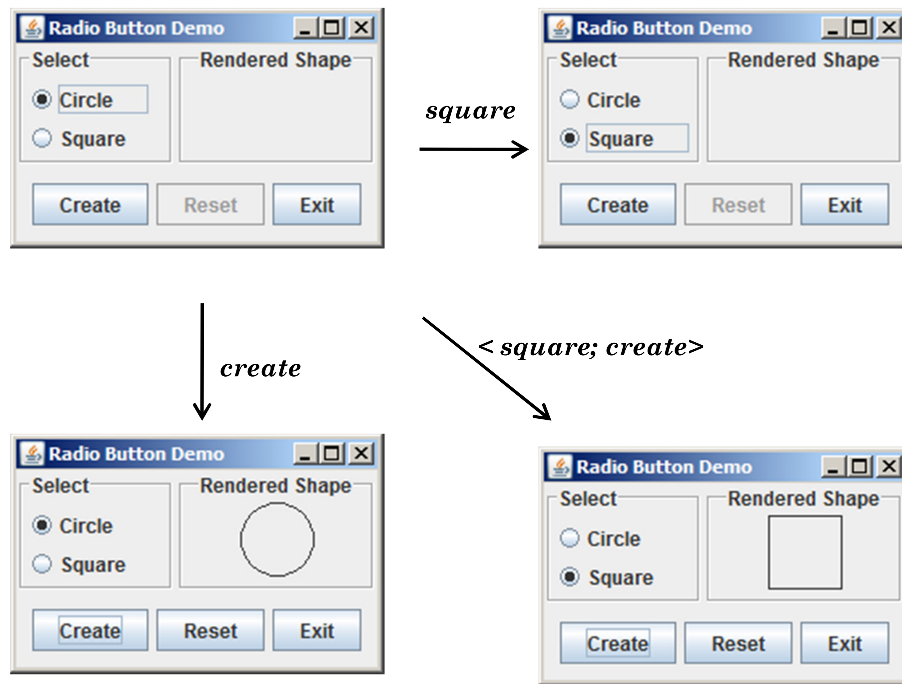


Figure 2.8: Event semantic interaction example.

The example illustrated in Figure 2.8 is just one case of how the GUI state may be used to pinpoint interactions between event handlers. Yuan et al. formally define six cases that describe (as evaluative predicates) situations in which two events, called e_1 and e_2, interact, i.e., e_1 influences e_2. In these six cases, e_1 and e_2 are system-interaction events in modeless windows; this situation is referred as *Context 1*.

Case 1: $\mathcal{P}_{1(1)}(e_1, e_2) = \exists w \in W, p \in P_w, v \in V_p, v' \in V_p, s.t.\ ((v \neq v') \wedge ((w, p, v) \in \{S_0 \cap e_1(S_0) \cap e_2(S_0)\}) \wedge ((w, p, v') \in e_2(e_1(S_0))))$; there is at least one widget w with property p with initial value v (hence the triple (w, p, v) is in S_0), which is not affected by the individual events e_1 or e_2 (the triple is also in $e_1(S_0)$ and $e_2(S_0)$); however, it is modified when the sequence $\langle e_1; e_2 \rangle$ is executed, i.e., the value of w's property p changes from v to v'.

Case 2: $\mathcal{P}_{2(1)}(e_1, e_2) = \exists w \in W, p \in P_w, v \in V_p, v' \in V_p, v'' \in V_p, s.t.\ ((v \neq v') \wedge (v' \neq v'') \wedge ((w, p, v) \in \{S_0 \cap e_2(S_0)\}) \wedge ((w, p, v') \in e_1(S_0)) \wedge ((w, p, v'') \in e_2(e_1(S_0))))$ there is at least one widget w with property p that has an initial value v, which is not modified by the event e_2; it is modified by e_1; however, it is modified differently by the sequence $\langle e_1; e_2 \rangle$.

In our running example, widget w_4, in the GUI's initial state, is not modified by event *square*, i.e., it remains empty; it is modified by event *create*, i.e., a circle is shown; however, w_4 is modified differently by the sequence $\langle create; square \rangle$. Hence, Case 2 applies to *create* and *square*.

Case 3: $\mathcal{P}_{3(1)}(e_1, e_2) = \exists w \in W, p \in P_w, v \in V_p, v' \in V_p, v'' \in V_p, s.t.\ ((v \neq v') \wedge (v' \neq v'') \wedge ((w, p, v) \in \{S_0 \cap e_1(S_0)\}) \wedge ((w, p, v') \in e_2(S_0)) \wedge ((w, p, v'') \in e_2(e_1(S_0))))$ there is at least one widget w with property p that has an initial value v, which is not modified by the event e_1; it is modified by e_2; however, it is modified differently by the sequence $\langle e_1; e_2 \rangle$. Note that this case is different from Case 2 because the event sequence remains the same, i.e.e_1 is executed before e_2.

In our running example, widget w_4, in the GUI's initial state, is not modified by event *square*, i.e., it remains empty; it is modified by event *create*, i.e., a circle is

shown; however, w_4 is modified differently by the sequence $\langle square; create\rangle$. Hence, Case 3 applies to *square* and *create*.

Case 4: $\mathcal{P}_{4(1)}(e_1, e_2) = \exists w \in W, p \in P_w, v \in V_p, v' \in V_p, v'' \in V_p, \bar{v} \in V_p, s.t.\ ((v \neq v') \wedge (v \neq v'') \wedge (v'' \neq \bar{v}) \wedge ((w, p, v) \in S_0) \wedge ((w, p, v') \in e_1(S_0)) \wedge ((w, p, v'') \in e_2(S_0)) \wedge ((w, p, \bar{v}) \in e_2(e_1(S_0))))$; there is at least one widget w with property p that has an initial value v, which is modified by individual events e_1 and e_2; however, it is modified differently by the sequence $\langle e_1; e_2\rangle$.

The above four cases all handle widgets that persist across the four states being considered, i.e.,, S_0, $e_1(S_0)$, $e_2(S_0)$, and $e_2(e_1(S_0))$. In many cases, event execution "creates" new widgets, e.g., by opening menus; the next case handles newly created widgets.

Case 5: $\mathcal{P}_{5(1)}(e_1, e_2) = \exists w \in W, p \in P_w, v \in V_p, v' \in V_p, s.t.\ ((v \neq v') \wedge ((w, p, v) \in e_x(S_0)) \wedge ((w, p, v) \notin S_0) \wedge ((w, p, v') \in e_2(e_1(S_0))))$; there is at least one *new* widget w with property p and value v in $e_x(S_0)$, i.e.,, it was created by event e_x (either e_1 or e_2) but did not exist in state S_0; it was created by the sequence $\langle e_1; e_2\rangle$ but with a different value for some property.

A common occurrence of event interaction in GUIs is enabling/disabling widgets, which may be modeled as the widget's `ENABLED` property being set to `TRUE` or `FALSE`.

Case 6: $\mathcal{P}_{6(1)}(e_1, e_2) = \exists w \in W, \texttt{ENABLED} \in P_w, \texttt{TRUE} \in V_{\texttt{ENABLED}}, \texttt{FALSE} \in V_{\texttt{ENABLED}}, s.t.\ (((w, \texttt{ENABLED}, \texttt{FALSE}) \in S_0) \wedge ((w, \texttt{ENABLED}, \texttt{TRUE}) \in e_1(S_0)) \wedge \texttt{EXEC}(e_2, w))$; there exists at least one widget w that was disabled in S_0 but enabled by e_1. Event e_2 is performed on w, represented by a predicate $\texttt{EXEC}(e_2, w)$.

In our running example, *create* enables *reset*; hence Case 6 applies.

Modal windows create special situations for Cases 1 through 6 due to the presence of termination events. User actions in these windows do not cause immediate state changes; they typically take effect after a termination event has been executed, leading to *contexts 2* and *context 3*.

Context 2: If both e_1 and e_2 are associated with widgets that are contained in one modal window with termination event TERM, then the definitions of $e_1(S_0)$, $e_2(S_0)$, and $e_2(e_1(S_0))$ are modified as follows: $e_1(S_0)$ is the state of the GUI after the execution of the event sequence $\langle e_1; \texttt{TERM} \rangle$, $e_2(S_0)$ is the state of the GUI after the execution of the event sequence $\langle e_2; \texttt{TERM} \rangle$, and $e_2(e_1(S_0))$ is the state of the GUI after the execution of the event sequence $\langle e_1; e_2; \texttt{TERM} \rangle$. All the predicates defined in Cases 1 through 6 apply, using these modified definitions, for e_1 and e_2 in the same modal window. The notation used for these predicates when applied in Context 2 is $\mathcal{P}_{n(2)}(e_1, e_2)$, where n is the case number.

Context 3: If e_1 is associated with a widget contained in a modal window with termination event TERM, and e_2 is associated with a widget contained in the modal window's *parent* window (i.e.,, the window that was used to open the modal window) then $e_1(S_0)$ is the state of the GUI after the execution of the event sequence $\langle e_1; \texttt{TERM} \rangle$, $e_2(S_0)$ is the state of the GUI after the execution of the event e_2, and $e_2(e_1(S_0))$ is the state of the GUI after the execution of the event sequence $\langle e_1; \texttt{TERM}; e_2 \rangle$. All the predicates defined in Cases 1 through 6 apply. The notation used for these predicates when applied in Context 3 is $\mathcal{P}_{n(3)}(e_1, e_2)$, where n is the case number.

There is an *Event Semantic Interaction* relationship between two events e_1 and e_2 at least one of the predicates in Cases 1 through 6 evaluates to `TRUE` in at least one context. If multiple cases apply, then one of the case numbers is used. Due to the specific ordering of the events in the sequence $\langle e_1; e_2 \rangle$, the ESI relationship is not symmetric. As demonstrated earlier, for our `Radio Button Demo` application: $square \rightarrow create$, $create \rightarrow square$, and $create \rightarrow reset$.

Once all of the cases have been implemented, the feedback-based process execution is straightforward. The steps of the execution are as follows.

1. The seed suite consisting of all 2-way interactions $\langle e_x; e_y \rangle$ between GUI events is executed on the software in state S_0; these test cases are simple enumerations of all EIG edges. All events e_y are also executed in S_0. The state information $e_x(S_0)$, $e_y(S_0)$, $e_y(e_x(S_0))$ is collected and stored.
2. The above predicates are evaluated for each pair of system-interaction events in the EIG that are either (1) directly connected by an edge (Context 1) or (2) connected by a path that does not contain any intermediate system-interaction events (contexts 2 and 3), i.e., there is at least one termination event that closes a modal window on this path. If one of the predicates evaluates to `TRUE`, the two events are ESI-related.

Once all the ESIs in a GUI have been identified, a graph model called the ESI graph (ESIG) is created. The ESIG contains nodes that represent events; a directed edge from node n_x to n_y shows that there is an ESI relationship from the event represented by n_x to the event represented by n_y. Figure 2.9 shows the ESIG of the

Radio Button Demo GUI. The solid lines are ESIG edges; for comparison, we also show the EFG edges (dotted lines) and EIG edges (dashed lines).

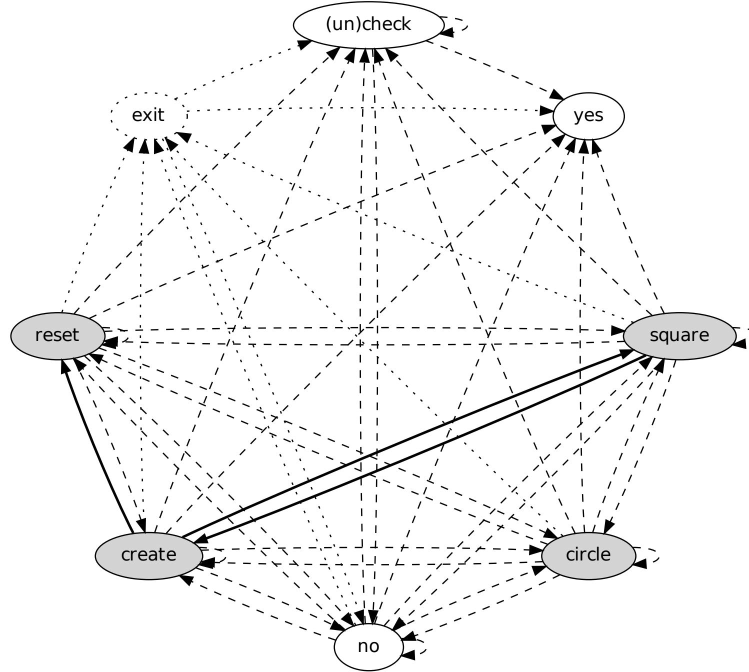


Figure 2.9: Event Semantic Interaction Graph.

As was the case for EFGs and EIGs, the ESIG may be traversed using different graph traversal algorithms to generate test cases. For our example ESIG in Figure 2.9, two test cases are $\langle create; reset \rangle$ and $\langle square; create; square; create; reset \rangle$.

2.2.2.4 Off-nominal Event Graph

Belli et al. develop a technique to generate off-nominal test cases using the GUI's workflow [33]. They define the workflow as an Event Sequence Graph (ESG).

Definition: An *event sequence graph* $ESG = (V, E)$ is a directed graph where $V \neq \emptyset$ is a finite set of vertices (nodes), $E \subseteq V \times V$ is a finite set of arcs (edges), and $\Xi, \Gamma \subseteq V$ are finite sets of distinguished vertices with $\xi \in \Xi$ and $\gamma \in \Gamma$ called entry

nodes and exit nodes, respectively, wherein $\forall v \in V$ there is at least one sequence of vertices $< \xi, v_0, ..., v_k >$ from each $\xi \in \Xi$ to $v_k = v$ and one sequence of vertices $< v_0, \ldots, v_k >$ from $v_0 = v$ to each $\gamma \in \Gamma$ with $(v_i, v_{i+1}) \in E)$, for $i = 0, ..., k-1$ and $v \neq \xi, \gamma$.

Intuitively, the ESG is similar to the event-flow graph, except that there is a notion of exit nodes in an ESG. Such a workflow allows the definition of an event sequence (ES).

Definition: Let V, E be as defined above. Then any sequence of vertices $< v_0, \ldots, v_k >$ is called an *event sequence* ES if $(v_i, v_{i+1}) \in E$, for $i = 0, \ldots, k$.

This definition is used to define a complete event sequence (CES) in the ESG.

Definition: An ES is a complete ES (or, it is called a *complete event sequence*, CES), if $\alpha(ES) = \xi \in \Xi$ is an entry and $\beta(ES) = \gamma \in \Gamma$ is an exit.

where α and β are the manually defined functions used to determine the entry and exit vertex of an ES.

These above definitions allow the formal definition of an off-nominal test case (or faulty event sequence) based on the ESG.

Definition: For an $ESG = (V, E)$, its completion is defined as $\widehat{ESG} = (V, \widehat{E})$ with $\widehat{E} = V \times V$.

Definition: The inverse (or complementary) ESG is then defined as $\overline{ESG} = (V, \bar{E})$ with $\bar{E} = \hat{E} \setminus E$.

Figure 2.10 shows the inverse ESG of the `Radio Button Demo` GUI. The dotted edges are ESG (EFG) edges. The oval shaded nodes represent initial events while the octagon nodes represent exit events.

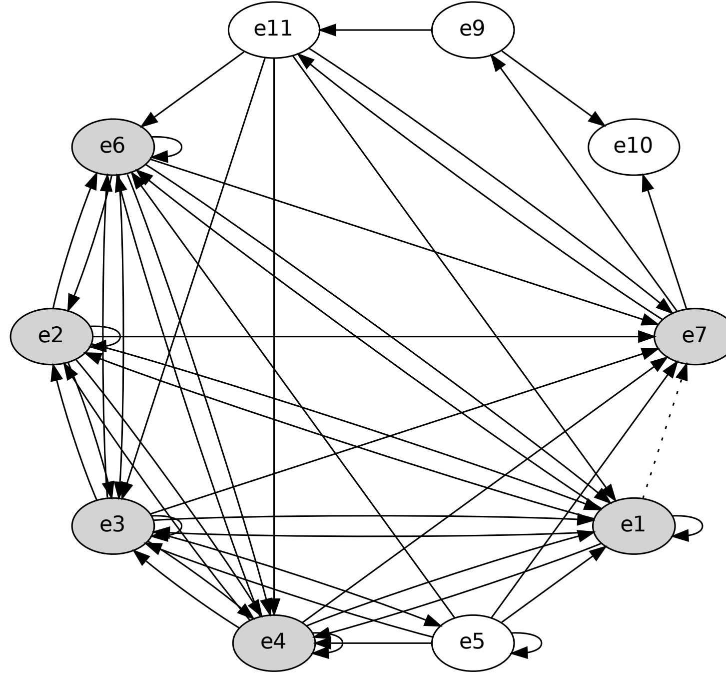


Figure 2.10: Inverse Event Sequence Graph.

The solid edges in Figure 2.10 are the ones that are absent from the ESG. More formally, they represent *faulty event pairs*.

Definition: Any edge of the $\overline{ESG}$ is a faulty event pair (FEP) for the ESG.

Definition: Let $ES = v_0, \ldots, v_k$ be an event sequence of length $k + 1$ of an ESG and $FEP =< v_k, v_m >$ a faulty event pair of the corresponding ESG. The concatenation of the ES and FEP then forms a faulty event sequence $FES =< v_0, ..., v_k, v_m >$.

Such an FES can be used as an off-nominal test case. An example of such a test case for our running example is $\langle square, circle, reset, no \rangle$. The pair $(reset, no)$ should not be executable because of the `Exit Confirmantion` modal dialog.

2.2.3 Pre- Post-Condition Models

In an approach presented by Memon et al. [34], the test designer models the GUI in terms of pre- and post-conditions for each event. The test designer then identifies commonly used tasks for the GUI; these are then input to the test case generator. The generator employs the pre- and post-conditions and specifications to generate event sequences to achieve the tasks.

The motivating idea behind this approach is that GUI test designers will often find it easier to specify typical user goals than to specify sequences of GUI events that users might perform to achieve those goals. The software underlying any GUI is designed with certain intended uses in mind; thus the test designer can describe those intended uses. However, it is difficult to manually obtain different ways in which a user might interact with the GUI to achieve typical goals. Users may interact in idiosyncratic ways, which the test designer might not anticipate. Additionally, there can be a large number of ways to achieve any given goal, and it would be very tedious for the GUI tester to specify even those event sequences that s/he can anticipate. The test case generator described in this section uses AI planning to generate GUI test cases for commonly used tasks using a GUI model based on pre- and post-conditions of all GUI events.

The test case generation process is partitioned into two phases, the *setup* phase and *plan-generation* phase. In the first step of the setup phase, the GUI representation is employed to identify planning operators, which are used by the planner to generate test cases. By using knowledge of the GUI, the test designer

defines the preconditions and effects of these operators. During the second or plan-generation phase, the test designer describes scenarios (tasks) by defining a set of initial and goal states for test case generation. Finally, the AI plannig system generates a test suite for the tasks using the plans. The test designer can iterate through the plan-generation phase any number of times, defining more scenarios and generating more test cases.

Formally, a planning problem $P(\Lambda, D, I, G)$ is a 4-tuple, where Λ is the set of operators, D is a finite set of objects, I is the initial state, and G is the goal state. Note that an operator definition may contain variables as parameters; typically an operator does not correspond to a single executable action but rather to a family of actions, one for each different instantiation of the variables. The solution to a planning problem is a plan: a tuple $< S, O, L, B >$ where S is a set of plan steps (instances of operators, typically defined with sets of preconditions and effects), O is a set of ordering constraints on the elements of S, L is a set of causal links representing the causal structure of the plan, and B is a set of binding constraints on the variables of the operator instances in S. Each ordering constraint is of the form $S_i < S_j$ (read as "S_i before S_j") meaning that step S_i must occur sometime before step S_j (but not necessarily immediately before). Typically, the ordering constraints induce only a partial ordering on the steps in S. Causal links are triples $< S_i, c, S_j >$, where S_i and S_j are elements of S and c represents a proposition that is the unification of an effect of S_i and a precondition of S_j. Note that corresponding to this causal link is an ordering constraint, i.e., $S_i < S_j$. The reason for tracking a causal link $< S_i, c, S_j >$ is to ensure that no step "threatens" a required link, i.e.,

no step S_k that results in $\neg c$ can temporally intervene between steps S_i and S_j.

For the `Radio Button Demo` application, one possible *task* may be to create a square shape for w_4. This task is shown in Figure 2.11. Even with this simple application, there are several ways to perform this task. In fact, there are an infinite number of ways—in principle, a user can click on the `Square` radio button an arbitrary number of times. This task is input to the planner by describing the state of all the widgets in the initial and goal states.

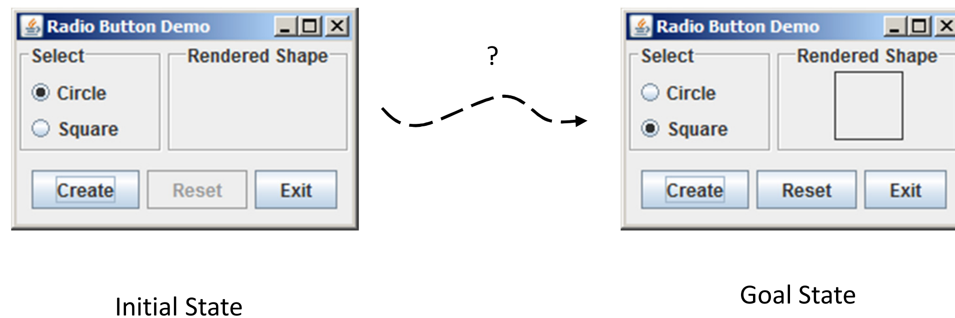
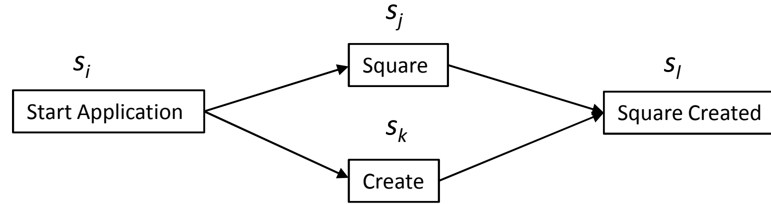


Figure 2.11: A Task Specification.

Together with a specification of all pre- and postconditions of the events, the task is used by the planner to output the plan shown in Figure 2.12(a). As mentioned above, most AI planners produce *partially-ordered* plans, in which only some steps are ordered with respect to one another. The plan in Figure 2.12(a) is one such plan. The ordering constraints are shown as edges and also explicitly stated in Figure 2.12(b).

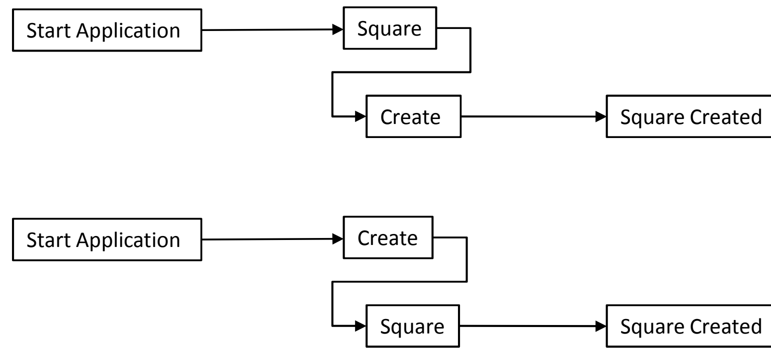
A total-order plan can be derived from a partial-order plan by adding ordering constraints, induced by removing threats. Each total-order plan obtained in such a way is called a linearizion of the partial-order plan. A partial-order plan is a solution



(a) A Partial-order Plan.

$$S_i < S_j; S_i < S_k; S_k < S_l; S_j < S_l$$

(b) The Ordering Constraints in the Plan.



(c) the Two Linearizations.

Figure 2.12: AI Planning.

to a planning problem if and only if every consistent linearizion of the partial-order plan meets the solution conditions. Figure 2.12(c) shows the two linearizion of the plan; each of these linearizion can be used as a test case.

2.2.4 Event Sequence-Based Models

Because GUI test cases are sequences of events, Kasik et al. [41] manipulate such sequences of events to obtain new test cases. Their approach is based on genetic algorithms. The key motivation behind using genetic algorithms is that there is a need to test the GUI from the perspective of different groups of users, e.g., experts and novice users. Unsophisticated and novice users often exercise GUI applications in ways that the designer, the developer, and the tester did not anticipate. An

expert user or tester usually follows a predictable path through an application to accomplish a familiar task. The developer knows where to probe, to find the potentially problematic parts of an application. Consequently, applications are well tested for state transitions that work well for predicted usage patterns but become unstable when given to novice users. Novice users follow unexpected paths in the application, causing program failures. Such failures are difficult to predict at design and testing time.

One approach to test the GUI for novice interactions is to release the software to a small community for beta testing. However, this approach is expensive and time-consuming. Kasik et al.'s approach generates test cases that mimic a novice user. The key idea behind this approach is that expert users take short paths through an application's GUI, using short-cuts when available and perform their tasks quickly. Novice users, on the other hand, take longer, exploratory paths to complete a task and gradually build better ways as they learn more about the application. It is challenging to automatically generate these paths for GUI testing.

In its simplest form, a genetic algorithm manipulates a table of random numbers; each row of the table represents a gene. The individual elements of a row (gene) contain a numeric genetic code and are called *alleles*. Allele values start as numbers that define the initial genetic code. The genetic algorithm lets genes that contain "better" alleles survive to compete against new genes in subsequent generations.

The basic genetic algorithm is as follows:

- Initialize the alleles with valid numbers.
- Repeat the following until the desired goal is reached:
  - Generate a *score* for each gene in the table.
  - Reward the genes that produce the best results by replicating them and allowing them to live in a new generation. All others are discarded using a *death rate*.
  - Apply two operators, mutation and crossover, to create new genes.

For GUIs testing, the event sequence is represented by a gene, each element being an event. The primary task of setting up the genetic algorithm is to set the death rates, crossover styles, and mutation rates so that novice behavior is generated. Also, to use genetic algorithms to generate meaningful interactions mimicking novice users, a clear and accurate specification of both the user interface dialog and the program state information is needed. The state information controls the legality of specific dialog components and the names of a legal command during an interaction. Without access to the state information, the generator may produce many meaningless input events.

For our running example, the `Radio Button Demo` GUI, an expert might use $\langle square, create \rangle$ to create a square. The genetic algorithm may convert this sequence into the longer sequence $\langle circle, create, square, create \rangle$, thereby mimicking a novice user.

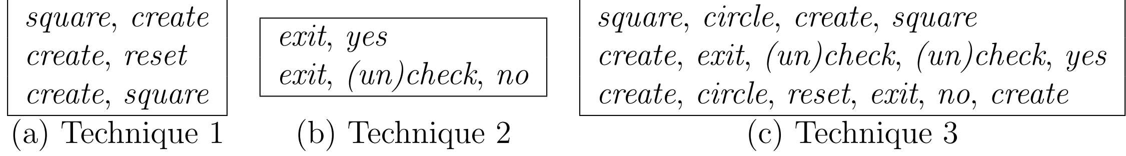
2.2.5 Probabilistic Models

As seen in this chapter, there are several techniques to generate GUI test cases based on a model of the GUI. In practice, a GUI test designer may use a mix of these techniques to obtain several test suites. The test designer is faced with two significant challenges:

- *Overlaps in test suites:* As can be imagined, many of these techniques often overlap in what they test. A test designer who uses two or more GUI testing techniques may waste valuable resources testing and retesting the same parts of the GUI. Ideally, the test designer would like to consolidate all the test suites and obtain one suite that minimizes overlaps.
- *Large number of short tests and few long tests:* The sheer size of the individual suites presents practical problems for test execution. Because each test case requires significant overhead in terms of *setup* and *teardown*, having a large number of short tests is inefficient. Ideally, the test designer would like to obtain longer sequences that combine the strengths of individual short-sequence suites.

Consider for example, the three test suites shown in Figure 2.13, each generated using a different technique. It may be expensive to execute and maintain all these test cases. Brooks et al. [42] employ a probabilistic model of the GUI to combine these suites.

The probabilistic model is based on the event-flow graph model. The model contains a collection of R paths through the EFG called $r_1, r_2, \ldots, r_R$. Each path



(a) Technique 1 (b) Technique 2 (c) Technique 3

Figure 2.13: Example test cases.

r_i where $1 \leq i \leq R$, consists of a sequence of n events in addition to $INIT$ and $FINAL$:

$$r_i = INIT, x_1, x_2, \ldots, x_n, FINAL;$$

$$\forall j \; e_j \in \{e_1, e_2, \ldots, e_{n-1}\} \wedge$$

$$follows(e_{j+1}, e_j)$$

where $x_1, x_2, \ldots, x_n$ and $e_1, e_2, \ldots, e_{n-1}$ are events in the EFG, r_i denotes a path, and each path r_i contains only events with a *follows* relationship between them. Valid paths can also be formed by the concatenation of two paths, *e.g.*, r_a and r_b, provided the first event of r_b $follows$ the last event of r_a in the EFG.

Let $count(e_i)$ return the number of times event e_i occurs in the paths $r_1, r_2, \ldots, r_R$. The prior probability that a randomly selected event from any of $r_1, r_2, \ldots, r_R$ is e_i is:

$$P(e_i) = \frac{count(e_i)}{\Sigma_{j=1}^{E} count(e_j)}.$$

Now, $count(e_i)$ and the prior probability calculation are extended from single events to sequences of events. Let s be a length-S subsequence of some path through

the EFG (not necessarily in $r_1, r_2, \ldots, r_R$):

$$s_i = x_1, x_2, \ldots, x_S$$

$$\forall j \; e_j \in \{INIT, e_1, e_2, \ldots, e_{n-1}, FINAL\} \wedge$$

$$follows(e_{j+1}, e_j).$$

The prior probability that a randomly selected, length-S subsequence from any of $r_1, r_2, \ldots, r_R$ turns out to be s is

$$P(s) = \frac{count(s)}{\Sigma_{s_i \in subs(S)} count(s_i)},$$

where $count(s)$ returns the number of times s occurs as a subsequence of $r_1, r_2, \ldots, r_R$ and $subs(S)$ is the set of all length-S subsequences in $r_1, r_2, \ldots, r_R$.

Given that s immediately precedes e_i, the conditional probability of e_i is

$$P(e_i|s) = \frac{P(s_1, s_2, \ldots, s_S, e_i)}{\Sigma_{j=1}^{E} P(s_1, s_2, \ldots, s_S, e_j)}.$$

Note that $P(e_i|s)$ can be thought of as $P(e_i)$ when s has length 0. This is not the same as $P(e_i|INIT)$, which is the probability that event e_i is the first event in the sequence, occurring immediately after $INIT$. Rather, $P(e_i|s)$ is the probability of e_i given *no information* about the events that precede it.

A *probabilistic EFG* (PEFG) is created by annotating each event (node) in the EFG with a table containing the event's prior probability and its probability con-

ditioned on each subsequence in $\{r_1, r_2, \ldots, r_R\}$ up to some maximum subsequence length, or history, H.

Figure 2.14 shows the PEFG obtained for the test suites of Figure 2.13. Column 2 of each table associated with every node shows the probability of executing the event associated with the node after the length 2 sequence shown in Column 1 of the table. For example, the entry for node *(un)check* corresponding to row *exit*, *(un)check* is 0.5. This is because the subsequence *exit*, *(un)check* appears twice in the original test suites. Once *exit*, *(un)check* has been executed, there is a 0.5 probability that the next event will be *(un)check*. These probabilities can be used to generate event sequences. One example sequence is $INIT, exit, (un)check, FINAL$. The resulting test case is $\langle exit, (un)check \rangle$.

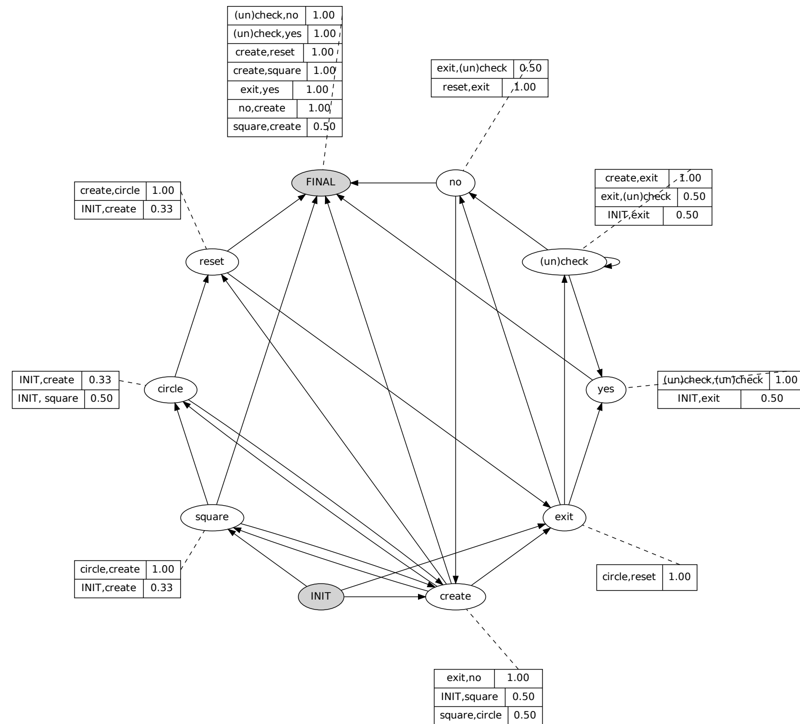


Figure 2.14: Probabilistic Event Flow Graph with history $H = 2$.

2.2.6 Combinatorial Interaction Models

Software system faults are not only caused by individual components working in isolation but also caused by the interactions between them [43, 44]. In its basic form, GUI interaction testing consists of testing for interactions between all GUI components and their selections. However, since the number of GUI components is often huge, the number of tests required to cover the combinational interactions grows large very quickly [22]. Several combinational interaction models have been proposed to model GUI component interactions and reduce the number of test cases. This section presents two combinatorial models used for test case generation – a *Latin square* to cover *pair-wise* interactions [39] and a *Covering Array* to cover *multi-way* interactions with an arbitrary coverage strength [45].

2.2.6.1 Latin Squares

White [39] proposes the use of Latin squares to model the GUI inputs and generate test cases. He identifies two ways in which GUI interactions can arise: statically and dynamically (or a combination of both). Static interactions are restricted to one screen whereas dynamic interactions move from one screen to another to perform events on GUI objects. White makes the assumption that it is enough to test pair-wise interactions of GUI events. Similar assumptions have led to success in finding errors efficiently for conventional software [46].

The concept of Latin square is used to maintain the pairwise interaction coverage while keeping the number of test cases minimized.

Definition: A *Latin square*, of order n, is a matrix of n symbols in a $n \times n$ cells, arranged in n rows and n columns, such that every symbol exactly once in each row and once in each column.

Definition: A pair of Latin squares $A = (a_{ij})$ and $B = (b_{ij})$ are *orthogonal* iff the ordered pairs (a_{ij}, b_{ij}) are distinct for all i and j. In other words, when superimposed on each other, the ordered elements pairs of two orthogonal squares created in each cell cover all n^2 pairs.

Given k factors F_1, F_2, ..., F_k, where each factor is a GUI component from which selections are made. The GUI inputs are modeled as follow:

- Reorder k factors by cardinality: $|F_1| \geq |F_2| \geq \cdots \geq |F_k|$.
- Construct $k-2$ orthogonal Latin squares with size n, where n is the cardinality of $|F_1|$.

To test k GUI components with maximum n level, we need $k - 2$ orthogonal Latin squares. The cell entries of the superimposed square represent $k - 2$ components in the test and the row and column indices represent the additional 2 components. Since the generated triples *(row index, column index, cell entry)* are unique, the pairwise coverage requirement is guaranteed.

The original model proposed by White only considered menu items. Because our running example does not have menus, we cannot use this approach to test our example GUI.

2.2.6.2 Covering Arrays

Yuan et al. [45] use *covering arrays* [34] to generate test cases. The key motivation behind using covering arrays is to generate longer sequences that are systematically sampled at a particular coverage strength. This approach is a generalization of the Latin square discussed in the previous section; a fundamental difference is that in covering arrays, the coverage strength is not limited to 2-way interactions. Furthermore, the use of covering arrays allows fine control over the location of each event in the test case.

Definition: A *covering array* $CA(N; t, k, v)$ is an $N \times k$ array on v symbols with the property that every $N \times t$ sub-array contains all ordered subsets of size t of the v symbols *at least* once. In other words, any subset of t-columns of this array will contain all t-combinations of the symbols.

Constructing a covering array with a minimal number of rows is an optimization problem. There are both mathematical algorithms as well as computational techniques such as greedy and meta-heuristic search [44]for this problem.

This test case generation technique leverages covering arrays to keep the number of test cases minimized while maintaining a required t-way coverage is between GUI events. A GUI is taken as input and first partitioned into different parts. Then, for each GUI part, a covering array is constructed to cover all events inside it. The output of this process is a set of covering arrays for all GUI partitions. Each array row becomes a GUI test case.

For our example `Radio Button Demo` application, we first partition the events

into different groups. For example, the three events *(un)check*, *yes* and *no* in the `Exit Confirmation` window can form the '`Exit`' group.

Suppose we are interested in 2-way coverage (i.e., test all possible 2-way interactions shown in Figure 2.15(a)) such that each event occupies all four positions in a length 4 sequence. If we used exhaustive enumeration, we need $3 \times 3 \times 3 \times 3 = 81$ test cases. Formulating the problem as a covering arrays $CA(N; 2, 4, 3)$, Figure 2.15(b), the number of test cases is only nine, each of which becomes a test case.

1. ⟨*yes, yes*⟩
2. ⟨*yes, no*⟩
3. ⟨*yes, (un)check*⟩
4. ⟨*no, no*⟩
5. ⟨*no, (un)check*⟩
6. ⟨*no, yes*⟩
7. ⟨*(un)check, (un)check*⟩
8. ⟨*(un)check, yes*⟩
9. ⟨*(un)check, no*⟩

(a) 2-way covering.

yes	*yes*	*yes*	*yes*
yes	*(un)check*	*(un)check*	*no*
yes	*no*	*no*	*(un)check*
no	*yes*	*(un)check*	*(un)check*
no	*(un)check*	*no*	*yes*
no	*no*	*yes*	*no*
(un)check	*yes*	*no*	*no*
(un)check	*(un)check*	*yes*	*(un)check*
(un)check	*no*	*(un)check*	*yes*

(b) Covering Array: $CA(9; 2, 4, 3)$

Figure 2.15: 2-way Covering and Covering Array.

2.2.7 Hierarchical Models

All of the testing techniques discussed thus far use a single model of the GUI. However, using only one model may be impractical for a large GUI. Several researchers have addressed this problem by modeling the GUI at multiple levels of abstraction. The GUI is broken down into different components and modeled hierarchically.

We now discuss three such hierarchies, namely Keyword-driven hierarchy [47], Hierarchical finite state machines [48], and UML-diagram based hierarchy [49].

2.2.7.1 Keyword-driven Models

Keyword-driven testing [50] is a script-based testing technique widely used in Industry. This technique divides the test case generation process into two phases: test plan and test implementation. In the test plan phase, the test designers design test cases using high-level activities called *action words*. In the test implementation phase, the test engineers transform the action words into executable events called *keywords*. To avoid ambiguities, the selected keywords are unique.

The idea behind using abstract test cases, i.e., those that contain high-level action words, is that domain experts, without any implementation skills, can easily design test cases using only the action words. This step can be done early, even before the system implementation has been started. The abstract test cases are also easier to comprehend; test maintenance is also more efficient.

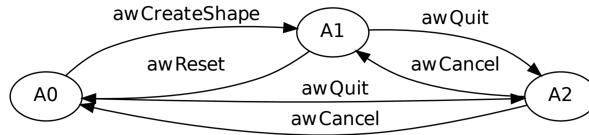
Inspired by the keyword-driven testing technique, Antti et al. [47] propose a GUI testing model using Label Transition Systems (LTS). A LTS is a state machine whose transition names are taken from an alphabet. Formally, a LTS is defined as:

Definition: A *labeled transition system* (LTS) is a quadruple $(S, \Sigma, \Delta, \hat{s})$ where S is a set of states, Σ is a set of actions (alphabet), $\Delta \subseteq S \times \Sigma \times S$ is a set of transitions and $\hat{s} \in S$ is an initial state.

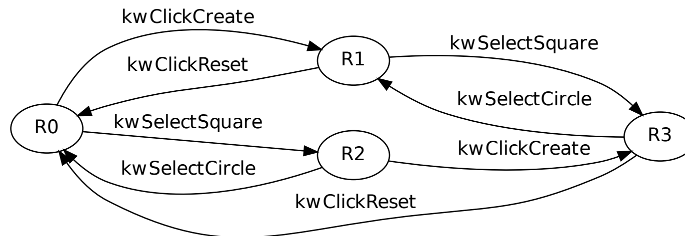
A GUI is modeled using two sets of LTSs corresponding to the two levels levels of abstraction in the keyword driven approach. The LTSs for the action word level are called *action machines* and the LTSs for the keyword level are called *refinement machines*. The action machines provide an overview of the system while

each refinement machines describes GUI navigation for certain parts of the GUI.

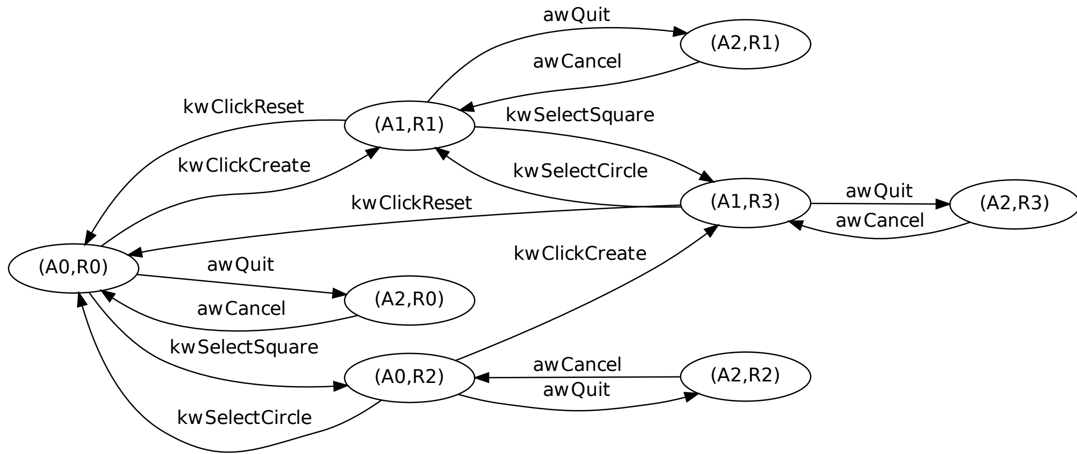
Figure 2.16(a) presents an action machine $\mathcal{A}$ for the `Radio Button Demo` application GUI. The labels in this machine represent the action words. Figure 2.16(b) is a refinement machine for the main window. The labels in this machine are keywords describing the actual GUI events.



(a) Action machine $\mathcal{A}$.



(b) Refinement machine $\mathcal{R}$.



(c) Parallel composition $\mathcal{C}$.

Figure 2.16: Label Transition Systems.

These machines are automatically composed to an executable LTS by a parallel composition operator defined as follows.

Definition: $||_R(\mathcal{L}_1, \ldots, \mathcal{L}_n)$ is the *parallel composition* of n LTSs according to

rules R where LTS $\mathcal{L}_i = (S_i \Sigma_i, \Delta_i, \hat{s_i})$ if let Σ_R be a set of resulting actions and $\surd$ be a "pass" symbol such that $\forall i : \surd \notin \Sigma_i$. The rule set $R \subseteq (\Sigma_1 \cup \{\surd\}) \times \cdots \times (\Sigma_n \cup \{\surd\}) \times \Sigma_R$. Now $||_R(\mathcal{L}_1, \ldots, \mathcal{L}_n) = (S, \Sigma, \Delta, \hat{s})$ where:

- $S = S_1 \times \cdots \times S)n$
- $\Sigma = \{a \in \Sigma_R | \exists a_1, \ldots, a_n : (a_1, \ldots, a_n, a) \in R\}$
- $((s_1, \ldots, s_n), a, (s'_1, \ldots, s'_n)) \in \Delta$ if and only if there is $(a_1, \ldots, a_n, a) \in R$ such that for every i $(1 < i < n)$
  - $(s_i, a_i, s'_i) \in \Delta$ or
  - $a_i = \surd$ and $s_i = s'_i$
- $\hat{s} = \langle \hat{s_1}, \ldots, \hat{s_n} \rangle$

A rule in a parallel composition associates an array of actions (or "pass" symbol $\surd$) of input LTSs to an action in the resulting LTS. The action is the result of the synchronous execution of the actions in the array. If there is a $\surd$ instead of an action, the corresponding LTS will not participate in the synchronous execution described by the rule.

Assuming that we have the following composition rules:

$$
\begin{aligned}
R \;=\{ \quad & (1)\langle awCreateShape, kwClickCreate, kwClickCreate\rangle \\
& (2)\langle awCreateShape, kwSelectCircle, kwClickCircle\rangle \\
& (3)\langle awCreateShape, kwSelectSquare, kwClickSquare\rangle \\
& (4)\langle awReset, kwClickReset, kwClickReset\rangle \\
& (5)\langle awCancel, \surd, awCancel\rangle \\
& (6)\langle awQuit, \surd, awQuit\rangle\}
\end{aligned}
$$

Figure 2.16(c) shows the composition machine $\mathcal{C}$ synthesized using the above rules. As we can see, the states in $\mathcal{C}$ are a combination (product) of $\mathcal{A}$'s states and $\mathcal{R}$'s states. By applying rules (1)-(4), two action words *awCreateShape* and *awReset* are refined to the corresponding keywords in $\mathcal{C}$. However, the action words *awCancel* and *awQuit* still remain unchanged. The rules (5) and (6) only copy them from $\mathcal{A}$ to $\mathcal{C}$. To refine those action words we need other refinement machines and composition rules.

After the composition machine is created, the test case generation is straightforward. Each path in the composition machine will become a GUI test case, which is a sequence of keywords. For our example, one possible test case might be ⟨*kwClickCreate*, *kwSelectSquare*, *kwSelectCircle*, *kwClickReset*⟩, which translates to $\langle create, square, circle, reset\rangle$.

2.2.7.2 Hierarchical Finite State Machines

Paiva et al. [48] use the hierarchy of GUI dialogs to create a hierarchical state-machine model for testing. In particular, the GUI is modeled as a hierarchy of FSMs whose vertices can either represent single states or groups of states in the original FSM. The model consisting of these FSMs is called a Hierarchical Finite State Machine (HFSM).

The hierarchy is based on GUI dialogs. Consider a GUI represented by k dialogs D_1, D_2, $\ldots$, D_k which manipulate a set of variable V: $V = \{v_1, \ldots, v_{|V|}\}$. From the complete FSM of the application, the tester manually specifies the state machine F_i for each dialog D_i. Given the FSM_{D_i} for a dialog D_i, it is possible to deduce the variables manipulated that dialog. A variable v_i is *written* by (or is affected by) a dialog D if there is a transition in FSM_D that changes the value of v_i. A variable v_i is *read* by (or influences the behavior of) a dialog D if at least one of the following conditions holds:

1. there are two transitions T and T' in FSM_D and a variable v_k in V (not necessarily $i \neq k$) such that: (i) the source states of T and T' are different only in the value of v_i; (ii) T and T' have the same triggering action (name and arguments); (iii) the destination states of T and T' have different values of v_k; and (iv) at least one of the transitions (say T) changes the value of v_k;

2. there are two states S and S' and a transition T with source S in FSM_D such that: (i) S and S' are different only in the value of v_i; (ii) there is no transition T' with source S' and the same action as T .

Let $PFSM_{Di}$ be the projection of FSM_{Di} onto the variables manipulated by dialog D_i then we can use $PFSM_{D_i}$ to describe the internal behaviors of D_i. Also from $PFSM_{D_i}$, it is possible to reconstruct FSM_{D_i} by taking the union of the instances of $PFSM_{D_i}$ for all possible combinations of variable values that are not manipulated by it.

Using the notation of PFSMs, the original state machine can be organized into a 3-level HFSM:

1. The *top level* is an abstract FSM representing the relationships between independent dialogs.
2. The *intermediate level* is a set of projected FSMs representing internal behaviors for each dialog.
3. The *bottom level* is a complete FSM representing the behaviors of the entire GUI.

Considering the `Radio Button Demo` application, and its GUI states represented by a length 4 vector {*log, exitWinOpen, created, shape*} as done in Section 2.2.1.1, a tester may specify a subFSM for the main window (dialog D_{Main}) to include all states where *exitWinOpen* is set to 0 and the transitions between them. The other states make up the subFSM for the `Exit Confirmation` window (dialog D_{Exit}). Figure 2.17(c) shows the complete FSM (*bottom level*) for the application. The states are organized into two regions (enclosed by dashed lines) corresponding to two subFSMs. Note that the same full FSM was previous shown in Section 2.2.1.1, except that its layout has changed.

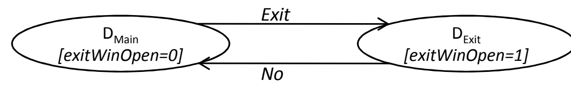
We can infer that that *created* and *shape* are two variables manipulated by D_{Main} while *log* is the only variable manipulated by D_{Exit}. Neither D_{Main} nor D_{Exit} manipulates $exitWinOpen$. Using this analysis, the *top level* and *the intermediate level* of the HFSM can be constructed as shown in Figure 2.17(a) and Figure 2.17(b).

Two dialogs are *independent* if the set of variables written by one dialog is disjoint from the set of variables manipulated (read or written) by the other. In this case, instead of testing the complete FSM we only need to consider their PFSMs individually. In other words, those dialogs do not need to be tested very time there is a change on variables they do not depend on. To test a dialog D, the variables not manipulated by D are fixed to a particular value and the test cases are generated using the $PFSM$ of D.

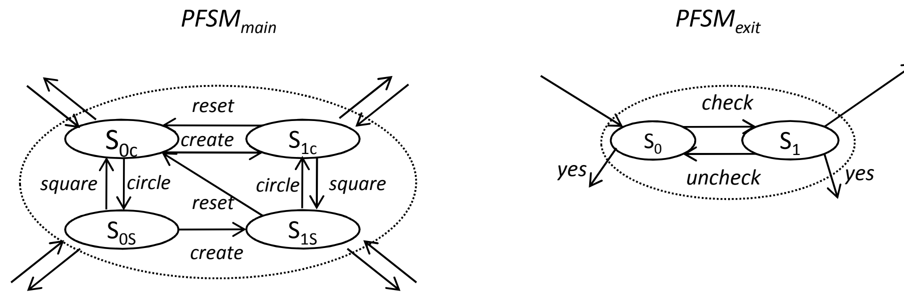
Applying this strategy to test the `Radio Button Demo`'s GUI, we first realize that D_{Main} and D_{Exit} are two independent dialogs. So we can test D_{Main} by fixing $log = 0$ ($exitWinOpen$ is already fixed) and generate test case in the $PFSM_{Main}$. Similarly, to test D_{Exit} we fix $created = 0$ and $shape = C$. Two transiting actions *exit* and *no* also need to be tested once by fixing $created = 0$, $shape = C$ and $log = 0$. Instead of testing all possible paths of the FSM in Figure 2.17(c), we now only need to examine those in bold.

2.2.7.3 UML Diagram-based

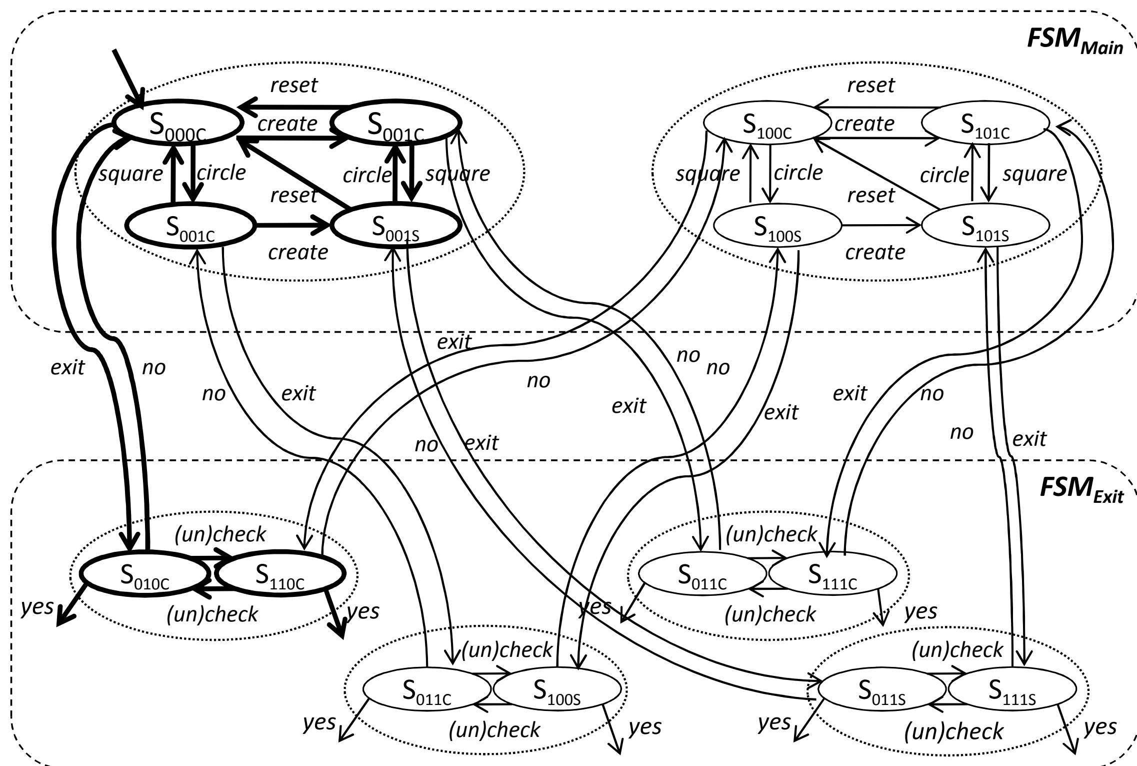
As seen in previous sections, using formal models to represent GUIs makes it possible to systematically generate and analyze test cases. However, these models are often



(a) Level 1.



(b) Level 2.



(c) Level 3.

Figure 2.17: Hierarchical Finite State Machine (self-loops are omitted for readability).

not intuitive, causing difficulties for test designers who are not familiar with formal Computer Science concepts. Paiva et al. [49] builds another visual layer on top of formal models to assist testers. The GUI is modeled using familiar UML notations

and then automatically translated to the underlying formal model by tools. More specifically, the formal model is a set of FSMs which are encoded in a specification language called Spec# (an extension of the C# programming language) [51].

The GUI behaviors are specified by four UML diagrams: *use case diagrams*, *activity diagrams*, *class diagrams* and *state machine diagrams*. These diagrams are enriched with additional stereotypes to enable automatic transformation from the visual forms to Spec# code.

Use case diagrams provides an overview of the main functionalities and features of the GUI application. They describe the scenarios in which the GUI is used. The use case diagrams are used to support other UML diagrams. However, there is no formal Spec# code directly generated from these diagrams. Figure 2.18 shows a use case diagram one might design for the `Radio Button Demo` example. The diagram consists of three main use case *Edit shape*, *Reset*, and *Exit* corresponding to three main scenarios the user may interact with the GUI.

Activity diagrams describes the business logic of use cases. The conditions and steps in the diagrams are directly encoded in Spec# syntax. Besides the user steps, they may have parameters that correspond to user inputs, pre/post-conditions (describing use case intent) and assertions. *Class diagrams* describes the static structure of the GUI. Each top-level window is modeled as an object. The state variables are represented by class variables, while the interactive controls are *State machine diagrams* describe the dynamic reactive behaviors of the GUI. The diagrams show GUI states at different levels of abstraction, the user actions available at each state, their effects on the GUI states, and the sequences of user actions. Each state

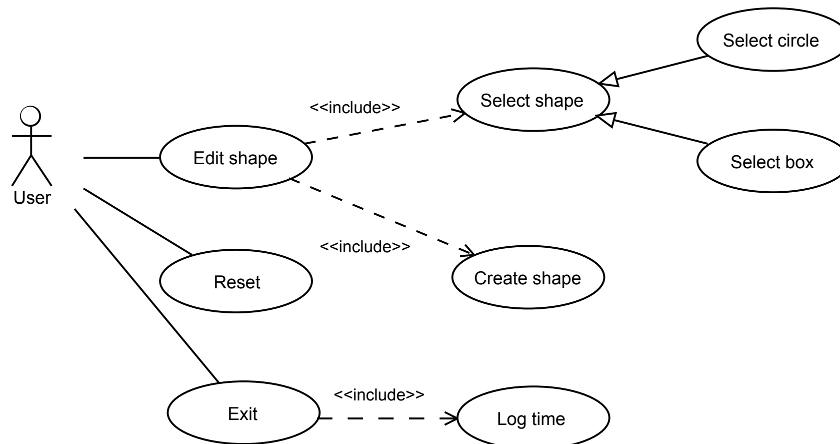


Figure 2.18: Use case diagram.

of the state machine can be formalized by a Boolean condition on the state variables. Each transition has a triggering event that is the call of a method representing a user action. The transitions may additionally have pre- and post-conditions on state variables and method parameters. A set of rules are developed to translate the state machine diagrams in to the Spec # code. After the formal specifications (e.g., Spec# code) are generated for all UML diagrams, an analyzer tool (e.g., Spec Explorer) is used to analyze the formal models and generate test cases for each diagram accordingly.

2.3 Summary

This chapter presented some of the recent advances in automated model-based GUI testing to motivate the need of the proposed work. Graphical user interfaces are by far the most popular means used to interact with software today. Unfortunately, the

state-of-the-practice in GUI testing has not kept pace with the rapidly evolving GUI technology. In practice, GUI testing is largely manual, often resulting in inadequate testing.

In its very fundamental form, the goal of GUI testing is to determine whether the GUI executes as expected, as documented in the specifications, or as required by the intended user. This definition is very broad and may encompass factors such as testing the GUI's usability, correctness, and performance. Since GUI testing is a multifaceted problem, no one technique can be used for GUI testing; in fact, in practice, a collection of techniques are almost always used. Model-based testing can be considered a promising approach to handle the complexity of the GUI-based software.

Finally, in all of the proposed model-based testing techniques, the GUI input space is assumed determined. However, with the context-sensitive nature of the modern GUIs, this assumption is no longer true. More comprehensive approaches to explore the GUI's input space and construct an adequate testing model are needed. The field of model-based GUI testing remains ripe for the application of upcoming areas of research.

Chapter 3

Observe-Model-Exercise* Testing Paradigm

System testing of software applications with a graphical-user interface (GUI) front-end requires that sequences of GUI events—that sample the application's input space—be generated and executed as test cases on the GUI. However, the context-sensitive behavior of the GUI of most of today's non-trivial software applications makes it practically impossible to fully determine the software's input space. Consequently, GUI testers—both automated and manual—working with undetermined input spaces are, in some sense, blindly navigating the GUI, unknowingly missing allowable event sequences, and failing to realize that the GUI implementation may allow the execution of some disallowed sequences.

This chapter presents a new paradigm for GUI testing called *Observe-Model-Exercise** (OME*) to tackle the emerging challenges in GUI testing. The key feature of OME* is its opportunistic use of *test case execution* for model enhancement. More specifically, we now *observe* the existence of new events either during *Ripping* or test execution to create or enhance an EFG$^+$ *model* – an extension of the EFG model – and *exercise* the newly observed GUI events in test cases using test adequacy criteria. The "*" in OME is due to the iterative nature of the entire approach. As new test cases are generated and executed, their executions are used to observe new events, which are added to the model and used to compute new test requirements,

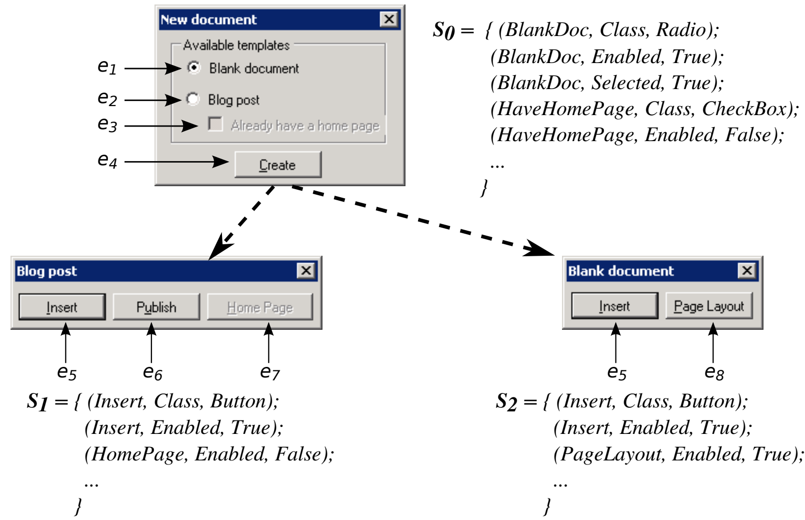
and subsequently to obtain additional test cases. The iteration ends when no new enhancements can be made to the model.

In next section, we present a step-by-step overview of OME* via an example. Then, we conceptually discuss the new models and algorithms to realize OME*. More detail on the emprical evaluation of OME* will be provided in the following chapters.

3.1 Overview

Because this work leverages several of previously reported techniques [19, 38, 22, 45] we feel that it is appropriate to present an overview, with a running example, to demonstrate the prior work as well as the new OME* paradigm. Figure 3.1(a) presents the GUI of our running example, motivated by the MS Word example that we showed in the previous section. It consists of four events in the *New document* window. Events e_1, e_2, and e_3 are *non-structural events*—they do not open/close windows/menus—that manipulate radio buttons and checkbox states. Selecting the *Blog post* radio button enables e_3. Event e_4 opens a new *modal window*[1] entitled either *Blog post* (with non-structural events e_5, e_6, and e_7) or *Blank document* (with non-structural events e_5 and e_8) depending on the states of the radio buttons in the *New document* window. Checking the *Already have a home page* check box enables e_7.

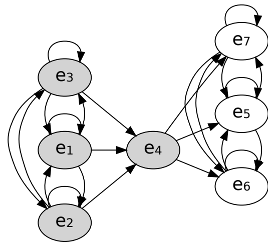
[1]A modal window, once invoked, restricts the focus of the user to the events within the window, until explicitly closed.



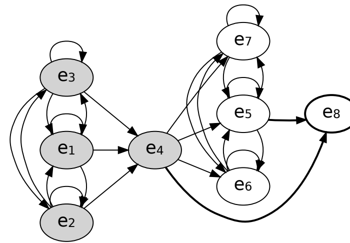
(a) GUI of running example

Edge	Path to edge
...	...
(e_1, e_2)	$NONE$
(e_2, e_3)	$\langle e_1 \rangle$
(e_3, e_4)	$\langle e_1, e_2 \rangle$
(e_4, e_5)	$\langle e_1, e_2, e_3 \rangle$
(e_5, e_7)	$\langle e_1, e_2, e_3, e_4 \rangle$
...	...

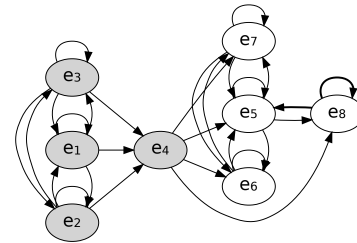
(b) Context-Aware Mapping



(c) EFG after *Ripping*



(d) EFG after e_8 is observed



(e) EFG after e_8 is executed

Figure 3.1: Running example.

Our overall goal is to test this running example. We summarize our process using the following steps:

Step 1: *Running the Ripper:* We start by running our *Ripper* on the application to obtain its EFG. Events e_1, e_2, e_3, and e_4 are all available in the main window; their states are as shown in Figure 3.1(a). Because of their availability in the GUI's start state, these events form the *initial nodes* set, I. The *Ripper* incorporates these nodes into the EFG; they are shown as shaded ovals in Figure 3.1. The Ripper then starts executing the encountered events one by one: e_1 followed by e_2, then e_3, and e_4. After events e_1, e_2, and e_3, the Ripper determines that they are non-structural events because no window is opened or closed; the `follows` relationships are then computed according to the algorithms presented in earlier work [19] and added to the EFG. Event e_4 opens a new window; because of the selected state of the *Blog post* radio button and checked state of e_3, the new window is titled *Blog post* with three events e_5, e_6, and e_7, all enabled. They are all executed but no new window opens. Their `follows` relationships are then computed and added to the EFG. The final EFG after the *Ripping* phase is shown in Figure 3.1(c).

Step 2: *Generating and executing test cases:* In this example, we will assume that we want to cover all EFG edges as our test criterion; we have used this criterion in earlier work (e.g., [18, 38, 22]). There are 24 edges in the EFG of Figure 3.1(c), yielding 24 test cases. The process for test case generation has been explained in earlier reported work [19]. Edges are selected one by one; for each edge (e_x, e_y), a path is computed—using a method called *prepend_context()*—from one of the *initial nodes* to (e_x, e_y), yielding a test case.

In previous work, because we lacked specific information about the events, our *prepend_context()* method could only rely on the EFG's topology to obtain a path from one of the nodes in the initial nodes set to the edge in question. For efficiency reasons, we used the shortest path. For example, if we select the edge (e_5, e_7), the shortest path to its first event is $\langle e_4 \rangle$, yielding a test case $\langle e_4, e_5, e_7 \rangle$. However, execution of this test case stops at e_5 because e_7 is disabled. This presents us with Challenge 1 mentioned in Section 1.2, Chapter 1: *it is challenging to generate particular event sequences to replicate context-sensitive behavior of events.*

In our work presented in this research, we now maintain a context-aware mapping between edges and paths to edges that have previously been seen to be executable. This mapping, together with our previous EFG model forms our new EFG+ model. Using the mapping, partly seen in Figure 3.1(b), the entry for edge (e_5, e_7) is $\langle e_1, e_2, e_3, e_4 \rangle$ because this was the executable path seen during *Ripping*. Hence, we will get $\langle e_1, e_2, e_3, e_4, e_5, e_7 \rangle$ as our test case. All 24 test cases are generated in this fashion, guaranteeing that all 24 EFG edges will be covered. These 24 test cases are then executed.

From our knowledge of the GUI, we know that we have yet to test event e_8. However, our *Ripper* does not even know of the existence of e_8. We need ways to drive the GUI into such a state that e_8 is exposed, tested, and added to our EFG model. To do so would, in principle, require that we traverse all possible paths in the GUI. This presents us with Challenge 2: *it is challenging to devise new event sequences that reveal new parts of the input space and help to enhance the model without incurring significant additional cost.*

In our work presented in this research, our approach to handle this challenge is to *simultaneously* use test execution for model enhancement. For example, one of our 24 test cases is $\langle e_1, e_4 \rangle$, whose execution will open the *Blank document* window with events e_5 and e_8. If at this time, we can recognize e_8 as a new yet-to-cover event, we can devise ways to cover it. We have developed mechanisms to add newly discovered events during test execution to our EFG. This presents us with Challenge 3: *it is challenging to identify new events/widgets, i.e., to determine whether an event/ widget has already been seen.* For example, we know that e_5 is the *Insert* button that we have seen earlier. On the other hand, we have never before seen e_8, the *Page Layout* button. Do we make a determination based solely on the "text labels" of these widgets? This would cause problems as many widgets in the GUI have the same text label (e.g., *OK*, *Cancel*). We have developed mechanisms to assign unique *signatures* to each widget; and heuristics to determine the uniqueness of the signatures.

Step 3: *Iteratively enhancing the EFG model, and generating and executing new test cases:* Having developed the ability to identify newly encountered widgets during test execution, we face Challenge 4: *it is challenging to incrementally make changes to the model to add new elements.* To date, we have developed algorithms to create the EFG in one pass. In our work presented in this dissertation, we develop techniques to incrementally enhance the EFG. The new EFG after the addition of e_8 is shown in Figure 3.1(d). Because we observed e_8 after the execution of e_4, we know that "e_8 `follows` e_4" which is why we have a new edge from e_4 to e_8. Moreover, because we know that e_5 is not a structural event, i.e., it does not open a new

window nor does it close the current window, e_8 could potentially `follow` e_5; hence, we also add the edge (e_5, e_8) to the EFG.

Now that we have two new not-yet-covered edges, (e_4, e_8) and (e_5, e_8), we need to generate test cases to cover them so that we can satisfy our test criteria. This presents us with Challenge 5: *it is challenging to incrementally generate new test cases.* In our work presented in this research, we have developed an algorithm to compute new test requirements from changes to the EFG+ model and generate test cases to satisfy the requirements. Using that algorithm, assume that we get test cases $\langle e_1, e_4, e_8 \rangle$ and $\langle e_1, e_4, e_5, e_8 \rangle$, to cover (e_4, e_8) and (e_5, e_8), respectively. These test cases are executed; e_8 is determined to be a non-structural event; two new `follows` relationships are added; these are new EFG edges (e_8, e_8) and (e_8, e_5) (new EFG shown in Figure 3.1(e)). As before, we now need to cover these new edges via new test cases. No changes are made to the EFG model during the execution of these test cases, and so the test process is complete, having satisfied the test criterion of covering all edges.

Even though we used a small example, we were able to show how OME* is used to discover new parts of the input space and exercise them. However, as we will see in our evaluation in Chapter 5, it is possible that we may not be able to automatically exercise all model elements that we observe; in such cases, manual intervention is needed.

3.2 Realizing the OME* Paradigm

We now discuss the new models, algorithms, and techniques that we developed to realize the new OME* paradigm. We structure our discussion around the contributions listed in Chapter 1 (Section 1.2).

3.2.1 Contribution 1: Context-Aware Mapping

In our past work, we relied on the "shortest-path algorithm" to obtain a sequence of events starting with a node in I, the initial nodes set, to the model element (e.g., EFG nodes, edges) that we are trying to exercise. As demonstrated by the example of edge (e_5, e_7) in the previous section, this does not always yield an executable event sequence, especially when GUI behavior is extremely context sensitive. To address this problem, we now maintain a new context-aware mapping between model elements and executable event sequences that have previously been successfully used to exercise these elements. Intuitively, during *Ripping* and test case execution, if we observe a certain model element is available after the execution of a particular event sequence, we create a new mapping to use later to reach the element.

Consider, for example, the execution of event sequence $\langle e_2, e_3, e_4 \rangle$ on the GUI of Figure 3.1(a). Recall that our coverage elements are EFG edges; hence our mapping will be between EFG edges and event sequences used to reach them. We start with the execution of e_2, after which the events e_1, e_2, e_3, and e_4 are available for execution. We execute e_3, which does not change the set of available events. We execute e_4, after which events e_5, e_6, and e_7 are available. The same information,

put in terms of the model elements, EFG edges, can be thought of as: *"edges* (e_4, e_5)*,* (e_4, e_6)*, and* (e_4, e_7) are reachable via the event sequence $\langle e_2, e_3 \rangle$." If, in the future, we want to cover these edges, we can use this information. This is precisely what we record in our mapping. Hence we see entries for the edges (e_4, e_5), (e_4, e_6), and (e_4, e_7) in our partial mapping shown in Table 3.1; there are several more, e.g., (e_3, e_4), which needs e_2. There are also several *NONE* entries, which means that the first element in the edge is in I, and it is enabled, making it trivial to reach this edge from the initial state.

Table 3.1: Partial Mapping.

Edge	**Path to edge**
(e_4, e_5)	$\langle e_2, e_3 \rangle$
(e_4, e_6)	$\langle e_2, e_3 \rangle$
(e_4, e_7)	$\langle e_2, e_3 \rangle$
(e_2, e_1)	*NONE*
(e_3, e_4)	$\langle e_2 \rangle$
(e_3, e_1)	$\langle e_2 \rangle$
(e_2, e_2)	*NONE*
(e_2, e_4)	*NONE*
...	...

We now describe the mapping formally and present an algorithm for its construction.

Definition: A **Context-aware Mapping** *CM* is a table of key-value pairs $\{me; \langle e_i, \ldots, e_j \rangle\}$; where *me* is a model element and $\langle e_i, \ldots, e_j \rangle$ is an event sequence after which *me* was previously observed to be available for execution, where event $e_i \in I$, the initial event set for the GUI. The entry is *NONE* if the first event in *me* is in I, i.e., no sequence is required to reach *me*.

As alluded to previously, the context-aware mapping is constructed from event

sequence execution. During the execution of each sequence, we maintain an explicit structure to compute the context-aware mapping. Figure 3.2 shows the structure for the example discussed above. At the very top is the executing event sequence $\langle e_2, e_3, e_4 \rangle$. The set of enabled events after each executed event is enclosed in a dotted oval. The shaded nodes are events in I. Solid arrows show the sequence executed; a dashed arrow from event e_x to e_y shows that e_y was available and enabled after the execution of e_x. To obtain the context-aware mapping, one needs only to trace each edge back to the starting event. For example, the edges (e_4, e_5), (e_4, e_6), and (e_4, e_7) have a path $\langle e_2, e_3 \rangle$ from the left-most node.

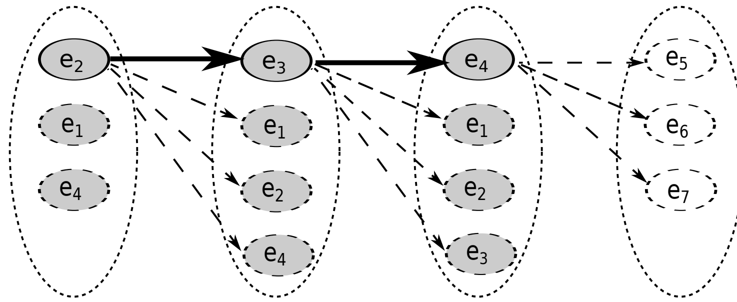


Figure 3.2: Available Events Observed During Execution.

Algorithm 1 shows how this structure, T, is constructed and used to create/update the mapping, *CM*. The algorithm takes three inputs: (1) a sequence of executed events, each paired with a set of events available and enabled after its execution, (2) the context-aware mapping available thus far (from previously processed event sequences), and (3) the set of events enabled in the initial state I. Lines 1–6 create the structure T. Edges are added from each executed event e_i to all events e_j that are available and enabled after e_i. Lines 7–23 use the structure to create the mapping. First, all the model elements *ME* are obtained from T (Line 7); for

our example, this is the set of edges. Then each element *me* is processed using one of two cases: (1) if the first event of *me* is enabled in the initial state, as is the case for edge (e_2, e_2), the mapping entry is set to *NONE* (Line 10), (2) otherwise a *searchPath()* function is used to find a sequence from the left-most element of T to *me* (Line 12). For example, for the edge (e_4, e_7), the path returned is $\langle e_2, e_3 \rangle$.

Because event sequences may be long, it is possible that the GUI is driven back to its initial state multiple times during execution. In such cases, the path may become unnecessarily long, which is why we use *truncate*() to remove leading events (Line 14). Lines 15–22 update the mapping *CM*. If an entry for *me* does not already exist in *CM*, the key-value pair $\{em, contextSeq\}$ is simply added; otherwise, the shorter of existing and new path is retained.

3.2.2 Contribution 2: Simultaneously Extracting New Model Elements During Test Execution

We define a GUI test case as a pair $(S_0, \langle e_1; e_2; e_3; \ldots; e_n \rangle)$, where S_0 is a designated start state of the GUI for this test case; and each $e_i \in E$, the set of events in the GUI. Our test executor (or *Replayer*) starts executing the test case by launching the GUI under test in start state S_0, and executes each event one by one. It determines the correctness of the GUI by using a test oracle [52]. Consider the GUI of our running example shown in Figure 3.1(a). The start state is marked S_0. All test cases start in this state. During test execution, the GUI transitions through a sequence of states where each state is obtained after the execution of an event. In our work,

Algorithm 1 Construct Mapping

Require: $\langle (e_1, \alpha(S_1)) \ldots, (e_n, \alpha(S_n) \rangle$: executed sequence
Require: *CM*: Context-aware mapping
Require: $\alpha(I)$: Events enabled in initial state

```
1:  T = ∅
2:  for i = 1 → n do
3:      for all e_j ∈ α(S_i) do
4:          T.addEdge(e_i, e_j)
5:      end for
6:  end for
7:  ME ← getModelElements(T)
8:  for all me ∈ ME do
9:      if firstEvent(me) ∈ α(I) then
10:         contextSeq = NONE
11:     else
12:         contextSeq = searchPath(me, T)
13:     end if
14:     truncate(contextSeq)
15:     if me ∉ CM then
16:         CM.addEntry(me, contextSeq)
17:     else
18:         contextSeq_old ← lookUp(CM, me)
19:         if |contextSeq_old| > |contextSeq| then
20:             CM.updateEntry(me, contextSeq)
21:         end if
22:     end if
23: end for
24: return CM: Updated context-aware mapping
```

we assume that the outcome of an event in a given state is deterministic. In our running example, once e_4 is executed, the GUI changes to state S_1 or S_2 based on the states of widgets corresponding to events e_1 and e_2.

We define a *GUI state* as the full set of all triples (w_i, p_j, v_k), where w_i is a widget currently extant in the GUI, p_j is a *property* of w_i, taken from a designated set of properties, and v_k is a *value* for p_j, taken from a set of possible values. We see some such triples in Figure 3.1(a) for our running example. The GUI states S_0, S_1, and S_2 would need to contain such triples for all widgets, all their properties,

and values.

We augmented our test executor to collect the state of the GUI after the execution of each event. Using reflection, we obtain the object class for each widget as well as the set of methods associated with the class. If the method name starts with the *get*, (e.g., *getLabel()*, *getX()*, *getY()*), we invoke it to *dynamically* obtain the value of the property. The part of the method name immediately following `get` becomes the name of the property. This approach is useful because it is impossible to predict the list of all properties of all possible widget types. For example, the `label` property is available for a *JButton* but not for a *JTextField*. Similarly, if the method name starts with the *is*, (e.g., *isEnabled()*, *isVisible()*), we assume that it returns a boolean value that is also added to our properties. Figure 3.3 shows part of our Java code used to collect states for GUI widgets.

```
Method[] methods = widget.getClass().getMethods();
for (Method m : methods) {
  String methodName = m.getName();

  if (methodName.startsWith("get")) {
    property = methodName.substring(3);
    value = m.invoke(widget, new Object[0]);
  }

  if (methodName.startsWith("is")) {
    property = methodName.substring(2);
    value = m.invoke(widget, new Object[0]);
  }
  ...
}
```

Figure 3.3: Code to collect GUI widget states.

Once we have the sequence of states, one state after each event, we developed a post-processing step to pass it for addition to the EFG model, which we discuss

in Section 3.2.4.

3.2.3 Contribution 3: Unique Widget Signatures

So far, we have conveniently referred to individual widgets by their *text labels*, e.g., *Insert*. Although this is fine for informal discussion in this chapter's text so long as the context is clear, use of a text label to identify a widget is insufficient for our tools such as the *Ripper* or *Replayer*. One cannot expect to perform an event on a widget, for example, using a method *invoke("Insert")*, and expect it to work correctly in all contexts; for instance, there might be two widgets, a button and a pull-down menu, in the current window with text label "*Insert*"; an automated tool does not know which one to execute. In such a situation, one might disambiguate by adding the "widget type" to the call, e.g., *invoke("Insert", Button)*. But this too would not work if both widgets were buttons. One may specify additional widget attributes, e.g., widget coordinates to the invocation to further disambiguate.

The above discussion is moot if each widget in the GUI had a unique identifier, perhaps assigned when programming the GUI, that remains unchanged across application runs. Such identifiers may be used by testers/tools to identify a widget, e.g., during the *ripping* and test generation phases, and then again later during test execution. Several researchers and practitioners have advocated the need for such identifiers for good *testability* of GUI software [53, 54]. However, in practice, such identifiers are rarely used [55]. In all fairness, there are situations in which it becomes difficult to use identifiers for widgets. For example, widgets may be *dy-*

namically generated based on some underlying data, e.g., one widget for each item available in an online store's database.

Whatever the reasons for not having widget identifiers in practice, the problem of not being able to uniquely identify widgets severely complicates our new work. Consider the *Insert* button in our running example. Our tools (*Ripper* and *Replayer*) may encounter it in two different contexts: first in the modal window entitled *Blog post* and second in the window entitled *Blank document*. These tools need to determine whether both these encounters were for the same widget or two different widgets; the determination will result in either one or two nodes in the EFG. Because we created this running example, we know that it is the same *Insert* in both instances, which is why we gave it the unique identifier e_5. In fact, we know that *Blog post* and *Blank document* are two instances of the same modal window. However, an automated tool has no way of knowing this information.

Admittedly, it is impossible to devise a general unique widget identification scheme that works for all possible GUIs. Any solution will have to be application-specific. In this section, we describe a general mechanism that must be manually fine-tuned on a per-GUI basis. Our mechanism is based on using a combination of certain parts of the state of the widget and its container (e.g., window). We cannot use the entire state for identification because it will contain some property values that change during the GUI's execution but do not play any role in identifying that widget. For example, the value of the *text* property for a *JTextField* object will change when the text changes; the *enabled* property changes when the object is enabled/disabled. Such properties cannot be used for our signature because any

change to their values will indicate a new widget, which would be incorrect.

More formally, we define the *signature*, C_{sig}, for a container C as follows:

$$
\begin{aligned}
C_{state} &\leftarrow \langle (p_1, v_1), (p_2, v_2), \ldots, (p_n, v_n) \rangle && (3.1)\\
\langle v_i, \ldots, v_k \rangle &\leftarrow \mathit{select}(\mathit{filter}_p, C_{state}) && (3.2)\\
C_{sig} &\leftarrow \Phi(\phi_i(v_i), \ldots, \phi_k(v_k)) && (3.3)
\end{aligned}
$$

where the user defines, per GUI, filter_p, a specification of a subset of the container's properties and transformations $\phi_i \ldots \phi_k$ on the values of the properties. The function *select* returns the values of the properties specified by filter_p and function Φ is a hash function on the transformed values.

Along similar lines, we define the *signature*, w_{sig}, for a widget w in a container with signature C_{sig}, as follows:

$$
\begin{aligned}
w_{state} &\leftarrow \langle (p_1, v_1), (p_2, v_2), \ldots, (p_n, v_n) \rangle && (3.4)\\
\langle v_i, \ldots, v_k \rangle &\leftarrow \mathit{select}(\mathit{filter}_p, w_{state}) && (3.5)\\
w_{sig} &\leftarrow \Gamma(C_{sig}, \gamma_i(v_i), \ldots, \gamma_k(v_k)) && (3.6)
\end{aligned}
$$

where filter_p and $\gamma_i \ldots \gamma_k$ are user-defined; and function Γ is a hash function on the transformed values and the container's signature.

In Section 5.3 (Chapter 5), we give examples of these user-defined functions and transformations, and empirically show, for our subject GUI applications, that they help to uniquely identify widgets.

3.2.4 Contribution 4: Incremental EFG$^+$ Enhancements

Once a new widget/event is identified, it is used to enhance the EFG+ model. We have already discussed, in Section 3.2.1, how to incrementally update the context-aware mapping, which is an important part of the EFG+ model. We now discuss how to incrementally enhance the EFG.

We have already informally discussed EFG enhancement in Section 3.1 and illustrated it in Figures 3.1(d) and 3.1(e). These figures actually show the three important steps for incremental EFG enhancement: (1) add a node to represent the new event; (2) add edges *to* the new node and (3) add edges *from* the new node to other nodes.

To explain these steps, we revisit two important terms in GUIs: *modal* and *modeless* windows. At any time during GUI interaction, a user is allowed to execute events within a modal window and any modeless window that was opened from the modal window. At no time can the user jump between modal windows without explicitly terminating them. Moreover, the user cannot interleave events that belong to modeless windows associated with different modal windows. Again, the user must explicitly terminate the modal window that is associated with the modeless window, explicitly invoke the other modal window, open the modeless window, and invoke any of its constituent events. A part of MS Word's window hierarchy is shown in Figure 3.4. *Edit Picture* and *Edit Chart* are modal windows whereas *Format Picture*, *Help Picture*, *Manage Template*, and *Help Chart* are modeless. Consider events x, y, z, a, b, and c. A user may execute x, y, and z together because they

are all contained in *Edit Picture*'s window group; similarly, events a, b, and c may be executed together. However, these two sets of events cannot interleave without their modal windows being explicitly invoked and terminated. The above behavior

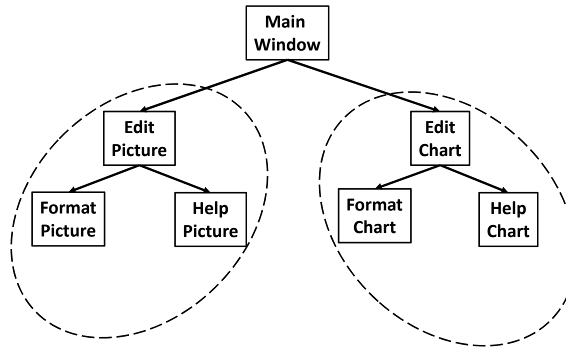


Figure 3.4: A partial window hierarchy of MS Word.

of GUI windows to restrict sets of events leads to the definition of a new term that we call the *scope* of an event. We define the *scope* of an event e as the set of events contained in the group of modal and modeless windows to which e belongs. We use scope in an algorithm to incrementally and efficiently enhance the EFG model.

More formally, we use Algorithm 2 to enhance our EFG. The algorithm is invoked after each event, e, is executed. It takes two parameters: (1) the EFG, and (2) the executed event. The set of all events available (enabled or disabled) is first obtained (Line 1). For each event, e_i, in this set, three steps are performed. First, if e_i has never been seen before (as was the case with e_8 in Figure 3.1(d)), then it is added to the set of nodes in the EFG (Line 4). Second, if the edge that was used to get to e_i was never seen before, then it is added as an edge (Line 7). This was the case for the edge (e_4, e_8) in Figure 3.1(d). Third, the set of events in e_i's scope are obtained (Line 9). Those that are not structural, i.e., do not open/close modal

windows, are used to add edges to the newly observed event e_i (Line 13). This is what we used for edge (e_5, e_8) in Figure 3.1(d).

Algorithm 2 Enhance EFG Model

Require: (N, E): EFG
Require: e: event executed
1: $AE \leftarrow getAllEventsAfter(e)$
2: **for all** $e_i \in AE$ **do**
3:   **if** $e_i \notin N$ **then**
4:     $N.addNode(e_i)$
5:   **end if**
6:   **if** $(e, e_i) \notin E$ **then**
7:     $E.addEdge(e, e_i)$
8:   **end if**
9:   $scope_i \leftarrow getScope(e_i)$
10:   **for all** $e_{ij} \in scope_i$ **do**
11:     **if not** (structural(e_{ij})) **then**
12:       **if** $(e_{ij}, e_i) \notin E$ **then**
13:         $E.addEdge(e_{ij}, e_i)$
14:       **end if**
15:     **end if**
16:   **end for**
17: **end for**
18: **return** (N, E): Updated EFG

The same algorithm is also used to add new edges *from* newly discovered events, as we saw in Figure 3.1(e) for e_8. However, this is done in a separate invocation of Algorithm 2, after the event is executed. Consider the invocation where the second parameter, e is the event e_8. The events available after e_8, Line 1, are $\{e_5, e_8\}$. Because there are no outgoing edges from e_8 in the EFG so far, Line 7 will add two new edges (e_8, e_5) and (e_8, e_8).

3.2.5 Contribution 5: Incremental Test-Case Generation

The new elements added to our model (e.g., EFG) may create new test requirements. For example, if a new edge has been added to the EFG and our test criterion is "cover all edges at least once," then we need to cover the new edge via a new test case.

Hence, we need new ways to incrementally generate test cases to cover new model elements. Note that not all changes to the model will create a need for new test cases. For example, if the criterion is "cover all nodes," then newly added edges in the model may not require additional test cases. The need for additional test cases is dictated by the test criteria, not new model elements.

To incrementally generate test cases, we maintain a set of model elements that have already been covered. Another set of model elements (the complete set – covered and not covered) is obtained from the latest EFG. These two sets give us the set of model elements that still need to be covered. For each not-yet-covered model element, we generate a test case to attempt to cover it. We first try to get a path from the initial state to the element using the context-aware mapping; this test case is guaranteed to be executable. If there is no mapping entry, then a path is generated using the shortest-path algorithm.

3.3 Summary

In this chapter, we presented a new testing paradigm that we call *Observe-Model-Exercise** (OME*) to address the challenges in model-based GUI testing. We described the key features of OME* and the algorithms used to realize it. In the next chapters, we will discuss more detail on our experimentation infrastructure (Chapter 4) and the empirical studies (Chapter 5) to evaluate OME*.

Chapter 4

GUITAR: A Generic Model-based GUI Testing Framework

In the previous chapters, we have introduced the ideas and models that make up the OME* testing paradigm. To evaluate the abstract concepts we have developed a framework for testing GUI-based application called GUITAR. The GUITAR framework provides tools to automate GUI testing activities including GUI model construction, test case generation, test case execution and test result analysis. It also supports multipe GUI platforms including Java Swing, Java SWT, UNO (Open Office), Android, iPhone and Web. The innovation of GUITAR lies in its architecture, which uses plug-ins to support flexibility and extensibility.

The framework is publicly released as an open-source project and available for download at `http://guitar.sourceforge.net`. Software developers and testers may use GUITAR to create new toolchains, new workflows based on the toolchains, and plug in a variety of measurement tools to conduct GUI testing. In this work, we will leverage GUITAR to conduct empirical studies to evaluate our OME* testing paradigm (Chapter 5).

4.1 Overall Architecture

We take the *component-based* approach [56] to design GUITAR. Figure 4.1 shows the UML2 component diagram [57] representing GUITAR's overall architecture. Each

component has a stereotype describing its role in the system: The *'core'* components provide global services in the system. The *'tool'* components provide blocks to build individual tools. The *'plugin'* components add additional, customized features to the tools.

Tools assemble components in different ways via their common interfaces. Testers can use tools independently or integrate into *toolchains* to support a specific workflow.

Each GUITAR component includes two separate layers to improve its flexibility and extensibility. The *abstract layer* defines an API to communicate with other components. This layer makes heavy use of abstract classes and interfaces to provide an abstract view of the component. The *implementation layer* provides low-level implementations for the component. This layer of separation makes components interchangeable, so that replacing one component does not interfere with other components of the framework. We describe each component in detail.

4.1.1 Model Core

The central component in GUITAR is the *Model core.* This component defines the conceptual data structures shared amongst other components, including three main structures:

- The *GUI Structure* represents the hierarchical view of the GUI. It consists of a set of the windows in the application. The windows initially available when the application starts are marked as *root windows.* All other windows are

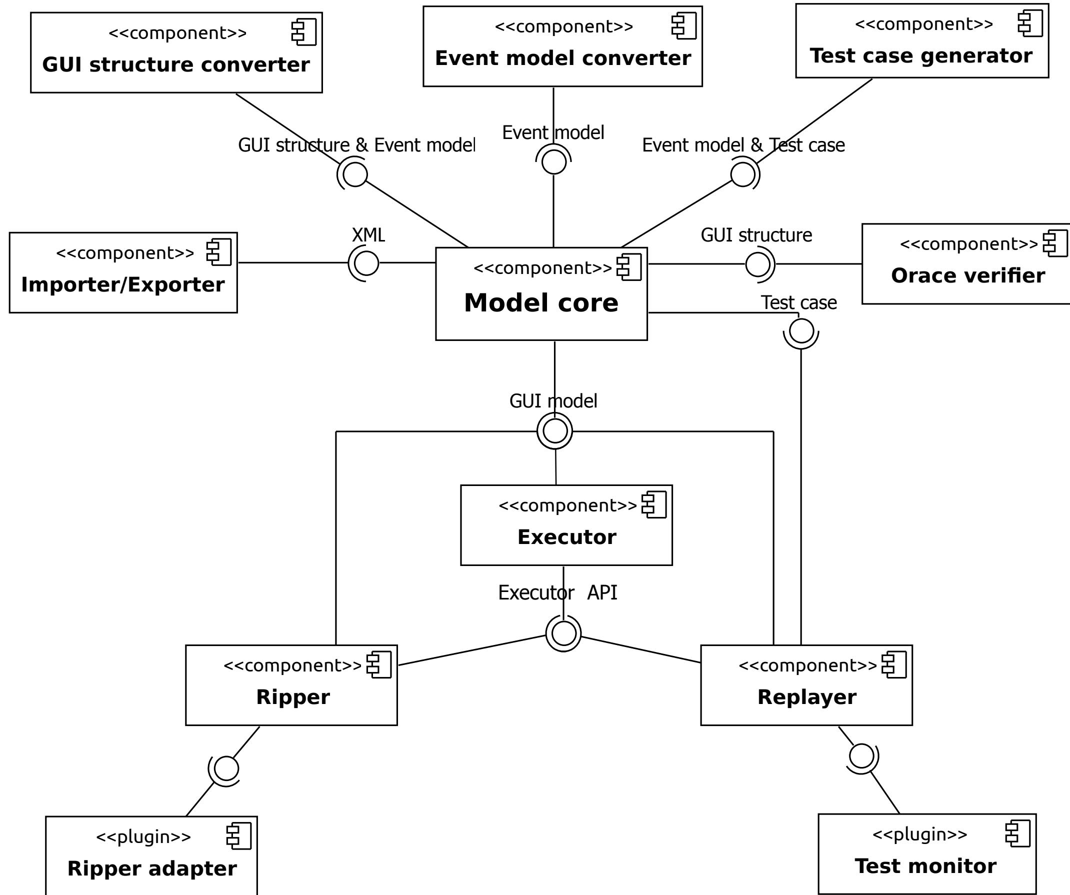


Figure 4.1: GUITAR component architectures.

invoked from root windows or their descendants. Each window contains GUI components with their properties and associated values. The GUI Structure organizes components in their natural, structural hierarchical layout (e.g., submenu is a child of top-level menu). In GUITAR, a GUI Structure can be used to represent either the static structure of the entire GUI or the dynamic GUI state as observed at a particular time. For example, the GUI Structure is used for both GUI trees output by the Ripper and GUI states output by the Replayer.

- An *Event Model* represents the relationships between events on GUI components, which we refer to as simply "events" for short. The Event Model consists of a directed graph with nodes representing events and edges representing relationships between events, e.g., the *follows* relationships in the EFG. Each event has an *event type* representing the class of action performed (e.g., left click, right click, text entry). Test Case Generators use an *Event Model* to systematically generate replayable test cases.

- The *Test Case* structure represents a sequence of GUI events, which can be performed one after another on the application from its initial state. A test case can optionally contain a sequence of *GUI Structure* objects representing the expected state of the GUI after each event as a form of assertion.

The remaining components interact with one another using the common data structures defined in the Model Core.

4.1.2 Platform-independent Components

As a feature of GUITAR, we want as many of the GUITAR components as possible to work independently of the GUI platform. Indeed, many components in the GUITAR architecture work at the abstract level of the GUI and therefore do not require any platform-specific details to provide important functionality.

The *GUI Structure Converter* converts from a GUI Structure to an Event Model. This tool analyzes the application GUI tree, extract all GUI events and constructs a graphical model representing the relationships between events. The

graph output by the GUI Structure Converter supports automatically generating test cases.

The *Event Model Converter* is similar to the GUI Structure Converter, except that it transforms from one event model to another. For example, probabilistic values can be added to the edges of an Event Model in support of probabilistic test case generation techniques. Users of GUITAR can extend model converters of both types to work with their own models and support tools based on these models.

The graph structure of the Event Model reduces test case generation to a graph traversal problem. The *Test Case Generator* takes an event model as input and performs specified graph traversal algorithms on the model to automatically generate test cases. Depending on the model exploration strategies, various test case generators can be built around an event model. The Test Case Generator also generates values for event parameters if necessary (e.g., reading text inputs from a data file to support text input events). Currently, GUITAR supports two types of test case generation strategies: *systematic* and *random* sampling on event models. The systematic sampling strategy generates test suites by covering all possible sequences of a given length from the event model. The random sampling strategy, on the other hand, generates test suites performing a random walk traversal on the model.

The *Oracle Verifier* provides mechanisms to automatically determine whether a GUI executed correctly for a test case. Since a test case for a GUI is a sequence of events, a test designer must decide both what to assert and when or how often to check an assertion, e.g., after each event in a test case or after the entire test

case completes execution. Variations of these two factors significantly impact the fault-detection ability and cost of a GUI test case. Currently, we have developed two Oracle Verifier implementations with GUITAR: the CrashVerifier for reporting crashes and the StateVerifier for matching output GUI states across different test case executions.

4.1.3 Platform-specific Components

Though we strive for as much platform-independence as possible in GUITAR, the need for test case execution requires platform-specific be specified in some components. These components interact with the GUI components and automate the GUI executions.

To enable the interactions between platform-specific and platform-independent components, we provide an intermediate component called *Executor* for GUI automation. Figure 4.2 shows the design of the Executor at a lower level. The Executor consists of two sub-components: The *Native GUI Automation* component is a platform-specific library such as Java Accessibility for Java JFC or Selenium WebDriver for Ajax-based web. This component directly interacts with the GUI. The *Executor Bridge* component communicates with the Native GUI Automation component to support the platform-independent Executor API. This API works as a contract between the platform-specific library and the high-level, platform-independent models defined in the Model Core. The Executor API interfaces with all other GUITAR components, so that once an Executor supports the Executor

API, the platform-specific components of the Executor can communicate with the rest of GUITAR in a platform-independent way.

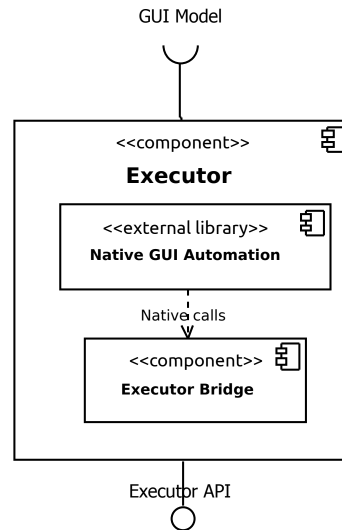


Figure 4.2: The *Executor* component.

The Executor API consists of four following interfaces:

- `GApplication`[1]: represents a GUI application and methods to initialize applications, such as starting and terminating the GUI and accessing window handlers.
- `GWindow:` represents a GUI window and methods to access window properties
- `GComponent:` represents a GUI component (i.e., a widget) and methods to access component properties
- `GEvent:` represents an event type and associated behavior, such as left-click, right-click, and text entry. A `GEvent` paired with the `GComponent` represents a specific GUI event on a GUI component (e.g., a left-click on the OK button).

[1]The prefix "G" indicates that a component is a GUITAR abstract class

The first three interfaces provide functionality to access the content of the GUI such as the GUI hierarchy and GUI properties. The last interface provides functionality to interact with the GUI. Section 4.3.2 will provide a case study discussing how to implement the Executor API to work with a specific platform.

The Executor plays an important role in GUITAR, replacing the need for manual interaction with GUIs to enable the use of much larger test suites. We currently provide two instances of the *Executor*: the *Ripper* and the *Replayer*. They implement two different automation strategies on the GUI.

4.1.3.1 The Ripper

The *Ripper* implements an algorithm (referred to as the "ripping algorithm") to reverse engineer an application's GUI structure [19]. The ripping algorithm automatically traverses the GUI, extracts all observed GUI components, and constructs a hierarchical structure of the GUI called the GUI tree. The GUI structure is stored in an XML file later for later use with various GUITAR tools.

The default Ripper behaviors can also be dynamically tuned by adding plugins called *Ripper Adapters*. A Ripper Adapter inserts additional actions at each ripping step to override the default GUI traversal strategy of the algorithm. Developers can implement a specific Ripper Adapter by extending the abstract class `GRipperAdapter`, which has two important methods:

- `isProcessed`: specifies which components that should be handled by this Ripper Adapter

- `ripComponent`: specifies how the Ripper should proceed with handling (e.g., interacting with and extracting properties from) the identified components

For example, an adapter called `IgnoreComponentAdapter` implements the capability to ignore undesired components (e.g., the 'Print' button leading to the external Printing dialog that we do not want to include in our testing process). This adapter overrides the Ripper's handling of components specified in the configuration file so that the Ripper skips these components. Section 4.3.1 will provide a more comprehensive example, where we use a Ripper Adapter to incorporate the handling of customized components into the Ripper or Java JFC.

4.1.3.2 The Replayer

The *Replayer* automatically executes test cases. It takes a test case as input, starts the application and executes the events of the test case in order, one-by-one. The users can also create plugins called *Test Monitors* to inject additional monitoring activities during execution. More specifically, Test Monitors extend the `GTestMonitor` interface with four main methods which are invoked at particular points during test case execution:

- `init`: invoked before any event is executed.

- `beforeStep`: invoked before an individual event is executed. It takes a `GTestStepEventArgs` object as argument to pass in any step-specific data (e.g., event ID).

- `afterStep`: invoked after an individual event is executed. It also takes a `GTestStepEventArgs` object as an argument.

- `term`: invoked after all events are executed.

An example of `GTestMonitor` is the built-in `StateMonitor` to capture GUI states during test case execution. In this monitor, the `afterStep` method records GUI states after the execution of the entire test case. Those states are exported as GUI Structure XML files that can be examined to determine test results. Another example of using Test Monitors is to inject a code coverage collector between each event to measure code coverage at the GUI event level.

4.1.3.3 Using Executors

Importantly, GUITAR itself does not impose any restrictions on the types of applications it can be used to test. However, a specific Executor implementation will usually only work on applications of a certain kind, due to the Executor's dependence on GUI automation. For this reason, we refer to GUITAR as supporting *platforms* of applications which can be accessed by a specific Executor implementation.

As we show in the following case study, we can implement an Executor for brand new platforms by providing implementations of all required abstract classes, then implement platform-specific Ripper and Replayer components to provide a toolchain. While we do not explore other extensions within this chapter, we could also use an existing Executor to develop new types of tools which need the automation of GUI interaction, such as a manual capture tool or alternative reverse

engineering tool.

Aside from the testing-related components above, GUITAR also provides Import and Export utilities to convert between its various XML structures and more popular data formats. These components help to incorporate powerful external tools into the GUITAR workflow. For example, the visualizations in Figure 5.2 (Chapter 5) were produced by exporting an EFG to the graph formats used by a visualization tool called Gephi[2]. The Gephi visualization can be edited with Gephi's own external graph editor, then imported back to GUITAR for use with other tools (e.g., test case generation and execution).

4.2 Creating Testing Workflow

Tools in the GUITAR framework can be used independently or stringed together into *toolchains*. In this section, we will show how testers can create GUITAR toolchain to support different testing workflows via an example.

Figure 4.3 shows a simple testing workflow using GUITAR tools. In this Figure, ovals represent processes and boxes represent testing artifacts or results. The workflow takes an application under test as input and automatically detects possible faults in the application. In particular, the workflow consists of five following steps:

1. GUI Ripping: Use the *Ripper* to reverse engineer a structural model of the GUI of an application called a GUI tree. The user may need to manually configure the *Ripper* to obtain a sufficiently valid GUI structure for the application.

[2] `https://gephi.org`

2. Model Conversion: Use the *Graph Converter* to converting GUI trees produced by the Ripper and other graphical models into to an EFG.

3. Test Case Generation: Use a *Test case generator* to automatically and systematically convert the EFG into test cases. Test cases are generated with various graph traversal algorithms.

4. Test Case Execution: Use the *Replayer* to automatically execute test cases on the application. The Replayer can instantiate values for events that require parameters (e.g., text-box, combo-box) by using user-specified or a database of default values. Runtime artifacts such as application logs and GUI state information are collected during test case execution.

5. Test Evaluation: Use the *Test Analyzer* tool analyze test case execution artifacts to determine if the test cases are passed or failed.

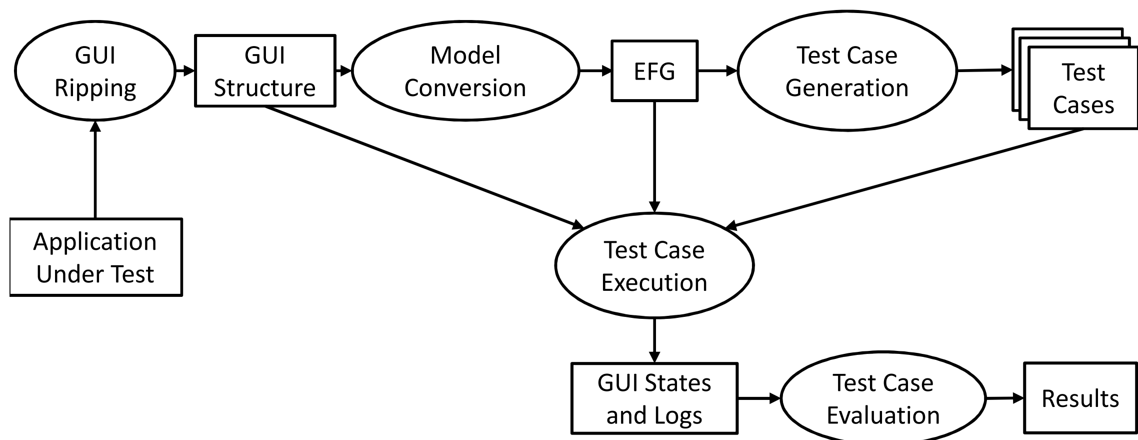


Figure 4.3: A simple GUITAR-based testing workflow.

This workflow, although simple, provides an end-to-end fully automated process to test GUI applications. It is also able to automatically collect a large amount of software artifacts such as test cases and runtime data. These features are very

important in enabling large-scale empirical studies. In Chapter 5, we will leverage GUITAR tools to evaluate a more comprehensive testing workflow - our OME* testing approach.

4.3 Extending GUITAR

With a loose-coupling design, GUITAR can be easily extended to support multiple research scenarios. Next sections describe several case studies to demonstrate how GUITAR is extended. A GUITAR extension can either work within a specific GUI platform or work cross multiple GUI platforms.

4.3.1 Within-platform Extension

In many cases, the application under test may use custom or otherwise unsupported GUI components. Custom events, custom widget-specific properties, and custom implementation can affect the Ripper's ability to extract GUI widgets and its properties. To better gather properties and interact with such components, a custom extension of GUITAR is required. In this case study, we consider an extension of GUITAR which improves testing of JabRef[3], an open-source application. JabRef is implemented using Java Swing and has some advanced GUI components which are, by default, inaccessible by GUITAR tools.

[3] `http://jabref.sourceforge.net`

4.3.1.1 Custom GUI Components

When ripping an application, the Ripper delegates the ripping of custom components to *Ripper Adapter*s (see Section 4.1.3.1). Each adapter directs the extraction of its corresponding components during ripping to make the GUI Structure richer, improving the accuracy of subsequent models and test cases.

JabRef uses a custom-developed GUI component called `GeneralTab`. This component improves the appearance of the `Preferences` window (see the left part of the window in Figure 4.4). Because of the implementation of `GeneralTab`, GUITAR by default does not know how to discover its child components, such as the components revealed on the right-hand part of the window when an item is selected on the left side. By default, when a `GeneralTab` is selected, the corresponding GUI components revealed on the right-hand side of the window do not show up directly as children of the `GeneralTab` in the GUITree. This problem occurs because the implementation of `GeneralTab` creates a separate panel and explicitly moves the affected components to their new location. Without support for this custom component, the Ripper attempts to handle the component as a standard Java Tab, missing all of the components on the right-hand side.

We implemented `GeneralTabAdapter`, which follows the specific logic of `GeneralTab` to extract the previously missed components. When the Ripper encounters a `GeneralTab` object, this adapter automatically searches for the location of the `GeneralTab`'s children and redirects the ripping of the tab to the new locations, as appropriate.

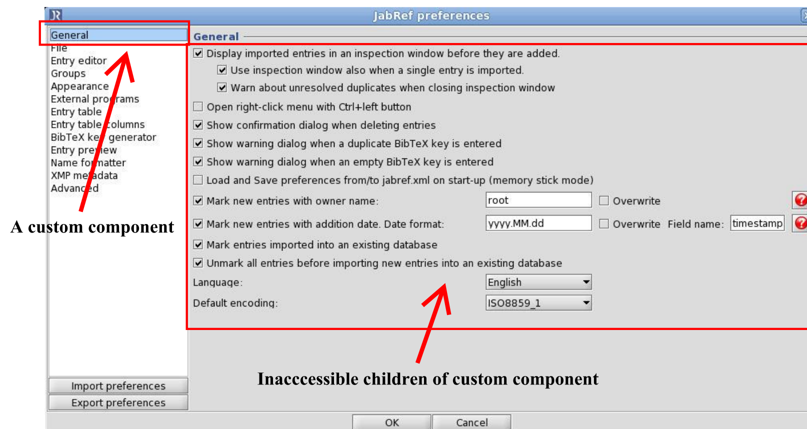


Figure 4.4: Customized component in JabRef Preferences window.

4.3.1.2 Custom Event Types

The GUITAR architecture manages GUI events separately from GUI components. GUITAR supports the implementation of customized *event types* for interacting with GUI components in custom ways. All event types in GUITAR extend the abstract `GEvent` class. There are two main methods in `GEvent` to implement:

- `isSupportedBy`: defines the class of components that support this event type.
- `performs`: defines what the event type actually performs on the components specified by the `isSupportedBy` method.

The existing GUITAR toolchain only supports a basic method to enter text at the begin of a `JTextArea`. GUITAR can also be extended with a custom event to enter text at a *specified position* in the `JTextArea`. In this case, a `GEvent` class should provide an `isSupportedBy` method which recognizes `JTextArea` objects and a `performs` method which invokes the low-level methods of `JTextArea` to insert the input text at the specified position. This additional event complements the GUITAR

toolchain's default text interaction of modifying the entire text of the component.

4.3.2 Cross-platform Extension

In this section, we describe GUITAR to how to extend an existing workflow to support multiple GUI platforms. As discussed in Section 4.1, there are two types of components in GUITAR: platform-independent and platform-specific components. To extend an existing workflow, we only need to implement the extensions for platform-specific components.

We use the Java JFC and Web platforms for illustration. We compare the extensions of the two platforms to show how to extend a testing workflow across multiple platforms.

In this case study, the JFC extension leverages the Java Accessibility Framework to monitor and drive interaction with a JFC GUI while the Web extension uses Selenium WebDriver for the same purposes. Supporting a new platform requires extension of the *Executor API* of GUITAR, as described in Section 4.1. More precisely, extending Executor requires three steps:

Step 1: Mapping platform's native objects to GUITAR's abstract objects: Each native GUI automation library (e.g., Java Accessibility Library, Selenium WebDriver) should have mechanisms for monitoring GUIs on the platform. This step involves identifying native objects in the platform which correspond to the abstract objects `GApplication`, `GWindow` and `GComponent` of the Executor API. Table 4.2 shows this mapping for both JFC and Web platforms. For

example, in the JFC platform, `GApplication` only needs to know the tested application's main class. In the Web platform, a `WebDriver` instance and the URL of the site's root web page provides analagous information.

Step 2: Accessing GUI properties: This step requires implementing methods for `GApplication`, `GWindow`, and `GComponent` objects to access GUI functionality. Columns 2 and 3 in Table 4.1 detail the required methods, with reference implementations in Columns 4 and 5 for the corresponding platforms. As we can see, the platform-specific implementation details can be very different, as long as they provide the correct functionality to the Executor API. For example, in the JFC platform, the `connect` method call invokes the `main` method in the main class, which starts the GUI application. In the Web platform, a `WebDriver` object handles the connection by starting the browser, loading the root URL, and setting up the connection between the *Executor* and the web site under test.

Step 3: Implementing event types: Finally, the platform needs support for any relevant event types. These extensions are similar to those for the custom event type described in Section 4.3.1.2. The event types extend the `GEvent` interface. For each event type, we need to specify the classes of GUI components supporting the event and how the event is actually performed in the supported components. Table 4.3 shows the summary of the event types implemented for our two example platforms. As we expect, some event types (e.g., submit) are platform-specific.

Table 4.1: Accessing GUI component information

Interface	Method	Descripion	JFC platform	Web platform
`GApplication`	`connect`	Establish a connection with the application under test and start testing	Use reflection to find and invoke the `main` method in the main class	Use the `WebDriver` to start the browser and load the root page
	`terminate`	Disconnect with the application under test	Invoke Java `System.exits` method	Invoke `quit` method from the `WebDriver`
	`getAllWindows`	Get all windows currently available	Return the values of `Frame.getFrames`	Return all open pages
`GWindow`	`isModal`	Check if the window is modal or not	Invoke the `isModal` method in `Window`	Always returns `false`
	`getContainer`	Get the window's top level component	Return the window's top `JPanel` object	Return the top level *'body'* tags
`GComponent`	`getTitle`	Get title of component	return text label or icon name of the `Component`	return tag (e.g., h1, img) of the `WebElement`
	`getClassVal`	Get class of the component	Return class name of the `Component`	Return tag type of the `WebElement`
	`getGUIProperties`	Get all GUI properties and their value	Use Java reflection to find and invoke all bean methods of `Component`	Use the `getAttributes` method to get all attributes of `WebElement`

Table 4.2: Mapping the internal GUI objects

	JFC platform	**Web platform**
`GApplication`	Application's main class	Selenium `WebDriver` object and the root page
`GWindow`	Java `Window` object	A Web page URL
`GComponent`	Java `Component` object	Selenium `WebElement` object

Table 4.3: Performing GUI events

GEvent	**JFC platform**		**Web platform**	
	Supported by	**Implementation**	**Supported by**	**Implementation**
Click	Components implementing the `AccessibleAction` interface	Invoke the `doAccessibleAction` method in `AccessibleAction`	The *'a'*, *'href'* tags and the *'input'* tags having type *'checkbox'* or *'radio'*	Invoke the `click` method in `WebElement`
EnterText	Components implementing the `Accessible-EditableText` interface	Invoke the `setTextContents` method in `AccessibleEditab-leText`	The *'input'* tags having type *'text'* and the *'textarea'* tags	Invoke the `sendKeys` method in `WebElement`
Submit	Not available	Not available	The *'input'* tags having type *'submit'*	Invoke the `submit` method in `WebElement`

GUITAR has been extended with Executor implementations for several common GUI platforms. Table 4.4 shows all platforms currently supported by GUITAR and the underlying Native GUI automation library used. The human effort required for these platform specific extensions varied considerably. For example, iOS, UNO, and Web implementations took considerably longer than their Java counterparts (typically one month by 4-member teams of undergraduate software engineering students). We attribute this difference primarily to the extra implementation required to interface between the Java core of GUITAR and the platform's native implementation.

4.4 GUITAR in Practice

Several researchers have been using GUITAR in their work. In most existing cases, researchers use the GUITAR framework to empirically study software testing in

Table 4.4: GUI platforms supported by GUITAR

GUI platform	Native GUI automation library
Java JFC	Java Accessibilty Framework
Web	Selenium Web Driver
Java SWT	Java SWT Accessibilty Framework[a]
Android	Robotium Framework
iOS	iOS Simulator[b]
UNO (Open Office)	UNO Accessibilty Framework[c]

[a]`http://wiki.eclipse.org/Accessibility`
[b]`http://developer.apple.com`
[c]`http://openoffice.org/ui/accessibility`

an automated way. For clarity, we divide the work of the primary researchers of GUITAR (i.e., from the University of Maryland) from the work of others who have applied GUITAR to their own scenarios.

We divide work by the GUITAR developers into six broad categories: developing workflows, and conducting large-scale studies of GUI testing, developing testing models from the EFG, designing GUI oracles, and repairing regression test cases.

In 2005, Memon et al. proposed the DART QA process for rapidly evolving software [35]. DART uses GUITAR tools to automate regression testing tasks, including model construction, test case generation, and analysis of test results. In later studies, researchers enhanced the test case generators in DART by incorporating the the actual usage of the application [42] and GUI runtime state feedback [58] to provide a better test case quality. These represent two interesting adaptations to workflow which required the development of custom tools to support the new workflow.

GUITAR researchers have implemented several test case generation techniques by developing Model Converters. To date, these tools input the existing EFG as a

base model and augment or filter the EFG before generating test cases. In this modified workflow, existing Test Case Generators (e.g., SequenceLength Generator) can operate on the new model, but test cases must then be reconstructed as necessary to be compatible with the standard EFG-driven Replayer. Existing techniques use the Event Interaction Graph (EIG) [40], Event Semantic Interaction Graph (ESIG) [40], and Probabilistic Event-flow Graph (PEFG) [42].

GUITAR supports the generation, execution and analysis of very large numbers of test cases. With this scalability, GUITAR toolchains provide very good support for the consideration of coverage criteria for GUI testing. In particular, researchers have used GUITAR to analyze three criteria for GUI testing coverage: event-interaction based [59], event-context based [45] and call-stack based [60]. GUITAR can provide candidate test suites for both model-based and non-modeled reduction techniques. In these workflows, we introduce the concept of a *test pool* being completely generated by GUITAR and the associated reduction techniques then executed and analyzed potential faults by GUITAR.

Strecker et al. [21] also leveraged this scalability to conduct a series of empirical experiments to studying the relationships between testing techniques and the characteristics of faults detected. The authors used GUITAR to automatically generate, execute and collect experimentation artifacts (e.g., logs of error information, code coverage, and GUI state information) supported by GUITAR tools. The experiment consisted of the execution of 100 test suites on 2 fault-seeded open-source applications. The entire process consumed nearly 100 machine-days and was executed on a cluster.

Researchers have also used Test Monitor extensions of GUITAR to perform in-depth analysis of GUI oracles. Xie et al. [38] proposed 6 types of GUI oracles and conducted a series of experiments on four fault-seeded Java applications to evaluate their strengths and weaknesses. In these experiments, a Test Monitor collected event-specific GUI states and a customized Oracle Verifier tool matched GUI states to expectations.

Using the information available in the EFG, Memon [61] proposed an approach to repair test cases for regressing testing by developing a Test Case Repairer tool which extended from the Test Case Generator modules of GUITAR. The Repairer automatically transforms all test cases detected by the Replayer as being unable to run due to GUI changes between application versions to executable ones. Later, Huang et al. [62] developed a similar tool but employed genetic algorithms instead.

Other researchers unaffiliated with GUITAR's development have also used GUITAR in their own studies of GUI testing. Swearngin et al. [63] used GUITAR to construct a model to predict human performance in HCI studies. The model creator starts by manually creating a set of methods (i.e., sequences of events on the GUI) to accomplish a specific task (e.g., changing text font face to bold). These methods are treated as GUITAR test cases and eventually executed automatically on the GUI by the Replayer. The author also added a Test monitor to collect additional widget properties required for their studies. Those states are used to infer implicit, unspecified methods. All of the methods (whether explicitly created or implicitly inferred) are supplied to a tool called CogTool to create a cognitive model of the GUI. This approach has been implemented in a tool call CogTool-Helper, which

uses the Replayer for JFC and UNO platforms as back-ends.

Some authors leverage GUITAR to support their own event models and test case generation algorithms. Huang et al. [64] introduce a weighted EFG model for test case generation. They use GUITAR to obtain a non-weighted version of the EFG, then assign weights to each node in the EFG based based on its properties. An empirical study conducted with 3 open source applications showed that the new approach can obtain a better fault detection rate than our standard workflow.

Focusing on the test case generation problem, Huang et al. [65] propose to build an feedback-directed approach on top of the standard GUITAR test case generator. They apply their "ant colony" algorithm to dynamically select the graph traversing path as test cases are executed.

GUITAR has also been used to produce benchmarks to evaluate GUI testing techniques. Mariani et al. [66] study with four open-source applications and compared the GUITAR standard workflow to their technique called AutoBlackTest. In a similar effort, Belli et al. [67] used GUITAR to evaluate a new event model called Event-sequence Graph. A case study with of two large modules of the commercial web portal ISELTA was conducted to compare the new model with the EFG.

4.5 Summary

In this chapter, we discuss GUITAR, a novel automated model-based testing framework for GUI-based applications. GUITAR supports many activities in GUI testing. GUITAR has an extensible architecture and works on multiple GUI platforms. The

framework is open-sourced and publicly available online. Next chapter will present a large-scale empirical study to evaluate OME* using the GUITAR framework.

Chapter 5

Empirical Evaluation

This chapter presents our empirical studies to evaluate the OME* paradigm discribed in Chapter 3. More specifically, we empirically determine whether the OME* paradigm improves the state-of-the-art, called the *baseline* (*BL*), in GUI testing. To this end, we will select several popular open-source software as subject applications to test. We will then generate and execute test cases (for *BL* and OME*) that attempt to satisfy predetermined adequacy criteria. Finally, we will compare the outcomes of the test runs.

5.1 Research Questions and Metrics

In this study, we are interested in answering the following two research questions:

RQ1: How effective is OME* when compared with *BL*? We will measure the *fault detection effectiveness* (FDE), *event coverage* (EC), and *code coverage* (CC) of the two approaches.

RQ2: By how much does the context-aware mapping improve the OME* approach? We will implement OME* in two ways—one with the context-aware mapping and the other without—and compare their FDE, EC, and CC.

Metrics: For FDE, we count the number of faults that led to the software crashing (terminating abnormally or throwing an uncaught exception). For EC, we mea-

sure EFG *node coverage* (E1) and EFG *edge coverage* (E2). For CC, we measure *statement (stmt.)*, *branch*, *method*, and *class* coverage.

5.2 Selecting & Setting Up Software Subjects

We select eight subject applications from two popular open-source communities Tigris.org[1] and SourceForge[2].

1. **ArgoUML:** A CASE tool for UML diagram design, code generation and reverse engineering;
2. **Buddi:** A financial tool for personal budget management;
3. **CrosswordSage:** A tool for creating and solving crosswords;
4. **DrJava:** An advanced integrated development environment (IDE) for Java programs;
5. **JabRef:** A database management tool for bibliographies management;
6. **OmegaT:** A language tool for automated translation;
7. **PdfSam:** An office utility for advanced pdf files manipulation;
8. **Rachota:** A time management tool for project time tracking;

They are all implemented in Java and rely on the GUI for user input. Table 5.1 summarizes their characteristics. The applications span a variety of domains, ranging from games to office utilities and software development tools. We selected the

[1] `http://www.tigris.org`
[2] `http://sourceforge.net`

most recent released versions at the time the study was conducted. All of them are widely used, demonstrated by the high numbers of downloads, and have broad user communities, demonstrated by the multiple numbers of languages available. They are all mature applications, in that they have been around for at least 5 years. They also have non-trivial code sizes in terms of the numbers of non-comment statements (S), branches (B), methods (M), and classes (C). Over the years, a large number of bugs have been reported by their respective communities and fixed by the developers in response.

Having identified the study subjects, we now prepare our tools to use them.

Table 5.1: Subject applications

Name	Abbv.	Version	Download	Usage Languages	Year	Bug Reports		Size**			
						Fixed	Total	S	B	M	C
ArgoUML	AU	0.33.1	N/A*	11	1999	N/A*	N/A*	69,954	32,084	16,091	1,891
Buddi	BD	3.4.0.8	897,520	13	2006	279	304	9,588	3,711	2,318	384
CrosswordSage	CS	0.3.5	4,623	1	2005	1	8	1,826	456	336	34
DrJava	DJ	r5004	1,227,393	1	2002	966	1091	64,994	17,485	15,229	2,394
JabRef	JR	2.7b	1,173,313	4	2003	564	768	44,522	18,176	7,502	1,267
OmegaT	OT	2.1.3	254,559	29	2002	462	503	19,756	6,772	4,519	714
PDFSam	PS	2.2.1	2,548,362	21	2006	71	87	6,097	2,043	1,504	194
Rachota	RC	2.3	74,107	11	2003	124	174	11,183	2,837	1,898	320

* The all-time statistics for ArgoUML are not publicly available. However, its popularity and maturity are partially demonstrated by the current more than 19,000 registered users and over 150 active developers (`http://www.isr.uci.edu/tech-transition.html`).
** S = Statements; B = Branches; M = Methods; C = Classes

5.3 Defining Functions for Unique Signatures

Our first preparation step is to ensure that we correctly identify each window and widget in the applications. As described in Section 3.2.3, we develop functions for windows (our containers) and widgets for this purpose. We start with windows, for which we need to develop $filter_p$, to select a subset of window properties, and $\phi()$ and $\Phi()$ to generate a unique window signature. It turns out that our study subjects only require the use of one window property, namely "*window title*." The value of this property is the title of the window, which, for the most part, are already unique. The exceptional cases are handled by mapping a few titles to regular expressions. For example, the title of the window "*Save file*" in ArgoUML can change dynamically during its execution; it always starts with the string *Save* followed by the full path to the file's location. The title changes when the user saves the file in another location. Hence, if we rely solely on window title, there is a danger that we consider each instance with a different title as a new window. To ensure that our tools recognize that all instances of this *Save file* window, with different titles, are in fact the same window, we map the title string, via the $\phi()$ function, to the regular expression '*Save (/.*)**'. We did this for a few windows.

The case for widgets is more complex. The *title* of widgets in our study subjects is repeated many times in the application. For example, many buttons share the title `OK`. For this reason, we use three properties to identify widgets, namely *title,* *icon*, and *class*, representing the main title/label of the widget, the file-name of the icon labeling the widget if it exists, and the object class used to implement

the widget, respectively. The values of these three properties are used in the Java *HashCodeBuilder* utility to generate a hash code for each widget.

We verified that our signatures are indeed unique. We manually examined each application and counted all its widgets and windows; we show the numbers in Table 5.2 under "*Manual Oracle*". We then used our tools to do the same. Our tools first extracted the *Unique Title Strings* and used our $\phi()$ function to map some of them to regular expressions. Similarly, our tools extracted the *Text Titles* for widgets, determined the image used for the *Icons*, and the Java *Classes* used to implement the widgets. These were then mapped to *Widget Hash codes.* Combined with their associated window hashes, we obtained unique *Mapped Widget Signatures.* As Table 5.2 shows, the resulting numbers matched our manual oracle exactly.

5.3.1 Sandbox and Text Parameters

Our second preparation step for testing the applications included setting up a *sandbox.* The key role of this sandbox was to ensure that each run of the application was independent of all prior runs. We defined a default configuration for each application. Before a test case is executed, the application is reset to its default configuration.

Finally, our third preparation step involved setting up data for parameterized events. A general parameter populating strategy was used. For events requiring a text input (e.g., text field, text area), a database that contains one instance for each of the text types in the set {*negative number, real number, zero, long random*

Table 5.2: Automated Widget/Window Identification vs. Manual Oracle.

(a) Window Identification.

SUT	**Manual Oracle**	**Automated**	
	Numbers of Windows	**Unique Title Strings**	**Mapped Title Strings**
AU	28	38	28
BD	20	24	20
CS	9	9	9
DJ	38	44	38
JR	52	56	52
OT	30	31	30
PS	6	6	6
RC	12	18	12

(b) Widget Identification.

SUT	**Manual Oracle**	**Automated**				
	Numbers of Widgets	**Text Titles**	**Icons**	**Classes**	**Widget Hashcodes**	**Mapped Widget Signatures**
AU	1040	710	20	113	1023	1040
BD	728	410	0	119	697	728
CS	137	111	0	40	130	137
DJ	1144	742	16	114	1132	1144
JR	1600	1127	30	134	1511	1600
OT	824	539	6	96	801	824
PS	323	272	18	72	315	323
RC	462	361	6	65	453	462

string, empty string, string with special characters} was used. All instances in the text type set were tried in succession for each test case.

5.4 Running the Experiment

Having prepared the study subjects, we are now ready to run the experiment, first establishing the baseline (*BL*) and then 2 instances of OME* (with and without the context-aware mapping). For each, we record coverage (*E1*, *E2*, *Stmt.*, *Branch*, *Method*, *Class*), number of *Nodes* and *Edges* in the EFGs, and number of test cases

that were *Generated*, and those that executed to completion (*Feasible*), and number of faults found.

More specifically, for each study subject, we perform the following steps:

1. Create EFG using the Ripper.

2. Generate tests from EFG, execute them, and record all metrics. This forms our *BL*.

3. As discussed in Section 3.2.2, new model elements are extracted during Step 2 above.

4. As per Section 3.2.4, enhance the EFG model.

5. Generate test suite from new EFG and execute them. Record all metrics for this suite. This forms our first implementation of OME* that does not use the context-aware mapping. We call this *noMap*.

6. As per Section 3.2.1, the context-aware mapping is created. Together with the EFG from Step 4, this forms the EFG+ model.

7. Generate test suite from new EFG+ and execute them. This forms one *iteration* of our *withMap* approach. Record all metrics for this suite and extract new model elements.

8. Enhance both EFG and mapping. Repeat Step 7 until the EFG+ model does not change.

Figure 5.1: Continuous Integration Testing System.

The total number of test cases and their execution times (in hours) are shown in Table 5.3. Note that we executed over 400,000 test cases in almost 1000 machine days. We used 120 2.8 Ghz P4 Linux nodes running in parallel.

The entire experiment process is scripted to provide fully automation. We used a distributed continuous integration tool called Jenkins[3] to control test case execution. Figure 5.1 shows a screenshot of our continuous integration testing system[4]. The left-hand side of the screen shows the list of available slave machines and the right-hand side details existing testing jobs and their current status.

[3] `http://jenkins-ci.org`

[4] See more detail at `http://samwise.cs.umd.edu:8080`

Table 5.3: Test Cases Generated and Execution Time

	BL		withMap		noMap	
	Test cases	**Exec time (h)**	**Test cases**	**Exec time (h)**	**Test cases**	**Exec time (h)**
AU	4,468	309	25,086	2,504	9,782	563
BD	2,890	50	32,712	752	8,204	124
CS	266	2	333	3	333	3
DJ	5,842	356	34,702	2,602	18,141	582
JR	39,555	2,982	170,809	10,984	54,516	3,945
OT	3,605	85	18,275	435	9,581	180
PS	6,856	68	9,135	116	9,135	100
RC	1,456	12	8,504	177	4,784	70
Total	**64,938**	**3,863**	**299,556**	**17,573**	**114,476**	**5,567**

5.5 Threats to Validity

As is the case with all empirical studies, our experiments suffer from several threats to validity. *Threats to external validity* are factors that may impact our ability to generalize our results to other situations. We have used eight open-source Java applications. Although carefully selected, they do not reflect the spectrum of all possible GUIs that are available today. Moreover, the applications are extremely GUI-intensive, i.e., most of the code is written for the GUI. We expect that results may be different for applications that (1) have complex underlying business logic and a fairly simple GUI, (2) are developed using other programming paradigms, and (3) are tested in-house for commercial applications. Finally, we initialized various values for text-fields manually and stored them in a database. We may see different results for different values.

Threats to internal validity are possible alternative causes for experimental results. Because we wanted to achieve full automation, we developed functions to identify widgets/windows uniquely. The instruments used for run-time state collec-

tion of GUI widgets were based on Java Swing API. These widgets may have additional properties that are not exposed by the API. Hence the states captured may be incomplete, causing us to map different windows/widgets into the same unique element. Moreover, because GUI execution requires frequent painting/repainting of windows, the captured state will be inaccurate if captured too early in the painting process; we set long artificial delays to allow the GUI to finish repainting.

Threats to construct validity are discrepancies between the concepts intended to be measured and the actual measures used. We used the number of crashes as our fault detection effectiveness metric; event and code as coverage metrics; these might not be useful metrics in all situations.

5.6 Results

We summarize our results in terms of the metrics that we collected for all 3 suites, i.e., *BL*, *noMap*, and *withMap*, in Table 5.4. We also break up the results of *withMap* by iteration so as to see the effect of OME*. From this raw data, we want to bring several points to the reader's attention.

Number of Iterations: Technique *noMap* has a single iteration as opposed to several for *withMap*. This is because even though new model elements were discovered during test execution in *BL* and used in Iteration 1 of *noMap*, very few of them were in fact reachable because of the absence of the mapping. This led to a large number of "infeasible" test cases that did not execute to completion. We revisit this point in more detail later.

Table 5.4: Data for RQ1 and RQ2.

AU

		BL	withMap					noMap
			1	2	3	4	5	
EFG	# Nodes	328	372	385	418	473	-	372
	# Edges	4,468	9,062	11,731	17,147	20,485	-	9,062
Mapping	# Entries	-	8,485	11,014	15,600	18,481	-	-
Test Cases	# Gen.	4,468	5,314	3,911	6,621	4,772	-	5,314
	# Feas.	3,763	4,771	2,743	5,376	3,223	-	556
Event Cov.	% E1	66.81	84.78	86.47	88.16	89.01	-	69.40
	% E2	18.37	41.66	55.05	81.29	97.03	-	19.21
Code Cov.	% Stmt	22.45	24.87	24.89	24.91	24.91	-	24.15
	% Brnch	10.31	11.83	11.86	11.88	11.88	-	11.29
	% Mthd	26.21	28.22	28.23	28.23	28.23	-	27.59
	% Class	52.09	54.63	54.63	54.63	54.63	-	53.83
# New Faults		-	2	2	0	0	-	0
# Total Faults		4	6	8	8	8	-	4

BD

		BL	withMap					noMap
			1	2	3	4	5	
EFG	# Nodes	249	297	344	412	457	500	297
	# Edges	2,890	7,129	11,011	17,019	24,396	30,196	7,129
Mapping	# Entries	-	5,486	8,130	13,993	20,661	25,369	-
Test Cases	# Gen.	2,890	4,289	4,271	6,757	7,943	6,562	4,289
	# Feas.	2,324	3,721	3,434	5,966	7,094	5,577	2,471
Event Cov.	% E1	40.00	48.20	56.60	65.40	68.60	72.00	31.01
	% E2	7.70	20.02	31.39	51.15	74.64	93.11	15.88
Code Cov.	% Stmt	38.54	47.61	48.90	49.04	49.19	49.34	38.67
	% Brnch	17.06	21.77	22.42	22.50	22.90	22.96	17.08
	% Mthd	36.45	42.23	42.92	43.18	43.27	43.49	36.71
	% Class	64.06	75.00	75.78	75.78	76.30	76.30	64.32
# New Faults		-	4	3	0	0	0	0
# Total Faults		1	4	7	7	7	7	1

CS

		BL	withMap					noMap
			1	2	3	4	5	
EFG	# Nodes	40	40	-	-	-	-	40
	# Edges	266	330	-	-	-	-	330
Mapping	# Entries	-	283	-	-	-	-	-
Test Cases	# Gen.	266	67	-	-	-	-	67
	# Feas.	230	64	-	-	-	-	7
Event Cov.	% E1	97.50	97.50	-	-	-	-	97.50
	% E2	69.70	89.09	-	-	-	-	71.82
Code Cov.	% Stmt	25.19	26.62	-	-	-	-	25.19
	% Brnch	8.55	8.77	-	-	-	-	8.55
	% Mthd	25.60	28.27	-	-	-	-	25.60
	% Class	41.18	41.18	-	-	-	-	41.18
# New Faults		-	3	-	-	-	-	0
# Total Faults		5	8	-	-	-	-	5

DJ

		BL	withMap					noMap
			1	2	3	4	5	
EFG	# Nodes	275	399	462	523	-	-	399
	# Edges	5,842	12,299	19,876	30,613	-	-	12,299
Mapping	# Entries	-	9,614	17,260	28,272	-	-	-
Test Cases	# Gen.	5,842	9,120	8,189	11,551	-	-	9,120
	# Feas.	4,237	6,108	7,256	11,359	-	-	2,094
Event Cov.	% E1	46.85	69.41	78.78	82.22	-	-	33.25
	% E2	13.84	33.79	57.50	94.60	-	-	20.68
Code Cov.	% Stmt	26.09	27.70	28.95	29.63	-	-	26.30
	% Brnch	17.74	19.90	21.11	22.08	-	-	18.17
	% Mthd	28.28	29.58	30.61	31.26	-	-	28.43
	% Class	51.92	52.80	54.51	55.18	-	-	51.96
# New Faults		-	1	0	0	-	-	0
# Total Faults		3	4	4	4	-	-	3

JR

		BL	withMap					noMap
			1	2	3	4	5	
EFG	# Nodes	483	603	830	1,058	1,177	-	603
	# Edges	39,555	54,272	68,063	123,757	168,658	-	54,272
Mapping	# Entries	-	45,622	62,960	109,672	145,840	-	-
Test Cases	# Gen.	39,555	14,961	14,686	56,199	45,408	-	14,961
	# Feas.	30,850	21,164	14,302	53,607	44,601	-	12,248
Event Cov.	% E1	39.25	50.89	70.18	88.62	98.81	-	28.00
	% E2	18.29	30.84	39.32	71.10	97.55	-	25.55
Code Cov.	% Stmt	29.12	33.70	37.16	38.36	38.63	-	29.15
	% Brnch	12.04	15.12	17.53	18.84	19.16	-	12.17
	% Mthd	29.95	34.95	38.10	39.24	39.43	-	29.95
	% Class	51.62	57.62	62.83	63.38	63.54	-	51.62
# New Faults		-	6	3	0	0	-	1
# Total Faults		4	10	13	13	13	-	5

OT

		BL	withMap					noMap
			1	2	3	4	5	
EFG	# Nodes	309	333	337	347	-	-	333
	# Edges	3,605	7,705	10,988	15,961	-	-	7,705
Mapping	# Entries	-	4,716	8,065	13,652	-	-	-
Test Cases	# Gen.	3,605	5,976	3,517	5,177	-	-	5,976
	# Feas.	2,712	4,308	3,356	5,001	-	-	2,730
Event Cov.	% E1	81.56	93.08	94.81	96.54	-	-	75.13
	% E2	16.99	43.98	65.01	96.34	-	-	34.10
Code Cov.	% Stmt	40.97	45.96	46.44	48.31	-	-	41.45
	% Brnch	23.36	29.43	29.71	32.21	-	-	24.08
	% Mthd	38.22	41.93	42.38	43.46	-	-	38.44
	% Class	62.61	65.97	66.67	67.23	-	-	62.75
# New Faults		-	1	0	0	-	-	0
# Total Faults		3	4	4	4	-	-	3

PS

		BL	withMap					noMap
			1	2	3	4	5	
EFG	# Nodes	111	118	-	-	-	-	118
	# Edges	6,856	8,273	-	-	-	-	8,273
Mapping	# Entries	-	7,557	-	-	-	-	-
Test Cases	# Gen.	6,856	2,279	-	-	-	-	2,279
	# Feas.	6,086	1,675	-	-	-	-	445
Event Cov.	% E1	94.07	100.00	-	-	-	-	45.38
	% E2	73.56	93.81	-	-	-	-	91.03
Code Cov.	% Stmt	41.74	42.43	-	-	-	-	41.74
	% Brnch	15.91	16.84	-	-	-	-	15.91
	% Mthd	36.97	37.50	-	-	-	-	36.97
	% Class	68.04	68.56	-	-	-	-	68.04
# New Faults		-	5	-	-	-	-	0
# Total Faults		2	7	-	-	-	-	2

RC

		BL	withMap					noMap
			1	2	3	4	5	
EFG	# Nodes	151	167	171	185	185	-	167
	# Edges	1,456	4,726	6,136	6,849	8,324	-	4,726
Mapping	# Entries	-	3,444	5,684	6,188	7,490	-	-
Test Cases	# Gen.	1,456	3,328	1,474	739	1,507	-	3,328
	# Feas.	1,107	3,288	1,451	718	1,476	-	1,237
Event Cov.	% E1	73.51	84.32	87.03	96.22	96.22	-	83.24
	% E2	13.30	52.80	70.23	78.86	96.59	-	38.10
Code Cov.	% Stmt	61.19	64.37	65.62	66.37	66.40	-	61.19
	% Brnch	33.45	37.12	38.14	38.74	38.88	-	33.45
	% Mthd	46.21	47.89	48.89	50.53	50.53	-	46.21
	% Class	81.88	85.63	86.25	86.25	86.25	-	81.88
# New Faults		-	1	2	1	0	-	0
# Total Faults		2	3	5	6	6	-	2

EFG: The EFG$^+$ model improves—gets bigger in size—with each iteration of *withMap*. In most cases, the number of EFG edges is significantly larger (e.g., 1044% for Buddi) compared to *BL*, showing that we were able to observe, model, and exercise a larger

number of GUI events, hence test more functionality. The number of entries in the context-aware mapping also grows steadily with each iteration, indicating that we are able to reach and execute new coverage elements. This is all directly reflected in improved fault-detection effectiveness and increased event and code coverage.

We pictorially examine and explain the growth in the EFG model via an example. Figure 5.2 shows a bird's eye view of the EFGs of our subject application *Buddi* for *BL* and 5 iterations of *withMap*[5]. Our goal is not to show details of the EFGs; rather, we want to show very high level pictures of the EFGs so that the reader can visually appreciate the changes from one EFG to the next. The EFGs have been drawn in such a way that the (x, y) location of each node in the EFG is fixed across iterations. For example, the *OK* event labeled in Figure 5.2(a) is in the same location, relative to all other nodes in Figures 5.2(b) through 5.2(f). We add labels to highlight specific parts that we discuss in the text.

At a high level, there are stark differences between the EFGs of Figure 5.2(a) (*BL* technique) and Figure 5.2(f) (final iteration of *withMap*). For example, the *Language Items*, *New Account*, *Edit Account Types* clusters and a large number of edges do not even appear in Figure 5.2(a); all these are observed only during the OME* process. Hence, *BL* has no way to cover these events/edges.

Buddi has a context sensitive GUI, which changes the set of available events based on the end-user's current "working perspective," i.e., at any time during its execution, events related to specific perspective are displayed. For example, during

[5]Additional visualizations of the EFGs, detailed code coverage reports, and actual fault reports are available at `http://www.cs.umd.edu/~atif/OME`

the execution of the *Ripper*, the *Edit* menu is only exercised only once in the *Report creation* perspective. Hence, only the *report* related sub-menu items are observed by the *Ripper*, and hence the *BL* technique.

In contrast, *withMap* is able to exercise *Edit* several times in many other perspectives. As a result, new sub-menu items are observed. As marked in Figure 5.2(b), for example, three new events *Edit All Transaction*, *Edit account types*, and *Create Account* are added to the original EFG when *Edit* is executed in the *Account Management* perspective. These events, when performed in subsequent iterations, will in turn, open new windows to extend the EFG even further (marked by the ovals in Figure 5.2(c)).

Our example illustrations also show that changes to the EFGs across iterations may not necessarily be changes in events, i.e., new edges may be added between previously observed events. For example, events in both *Edit Account Types* and *New Account* windows have been observed during Iteration 2 (via menu items in the main window). However, the edges linking these two windows are only revealed during Iteration 3, when the *New Account* event in *Edit Account Types* is exercised (shown using a "New edges" label at the bottom-left of Figure 5.2(d)). These new edges provide a new way to exercise events in the *New Account* window.

We also note that even a very small number of newly observed events may lead to a significant change in model size. Because *Buddi* allows users to work in multiple perspectives simultaneously (e.g., adding a new account when generating a report), the windows are mostly implemented as modeless. For that reason, as soon as a new event is observed and added to the model, it gets connected to all previously

known events, making the graph very dense. As shown in Figure 5.2(d) through Figure 5.2(f), the edge cluster from the *All Transactions* window's events back to all the events in the main window (the center of the EFG) grows significantly across iterations.

Mapping and Test Case Feasibility: Our context-aware mapping size also grows across iterations of *withMap*. This mapping plays a big role in ensuring that a larger number (compared to *noMap*) of test cases remain feasible. The 4 most important data points to illustrate this are the *# Generated* and *# Feasible* entries under Iteration 1 of *withMap*, and under *noMap*. For example, in *ArgoUML*, only 556 of 5314 test cases were feasible, i.e., executed to completion, with *noMap*. In contrast, 4771 of 5314 test cases were feasible with *withMap*. This shows that by retaining when events were observed to be executable and using this information when exercising these events again proved to be very successful at making test cases executable.

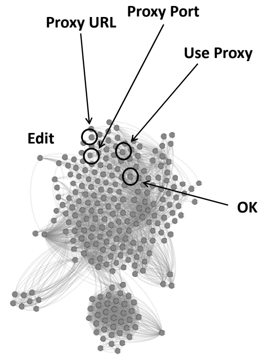
The *JabRef* data of *# Generated* and *# Feasible* entries under Iteration 1 of *withMap* highlights another important aspect of our mapping. Even though we generated 14,961 new test cases during this iteration, a total of 21,164 test cases were feasible and executed. This is because some of the test cases from the previous run (in this case from *BL*) that remained unexecutable earlier, were now executable due to new entries in the mapping.

Event and Code Coverage: We see that we gradually increase the amount of code and events that we cover across iterations. However, we never achieve 100% coverage of our criterion, i.e., cover all EFG edges. This means that we observe

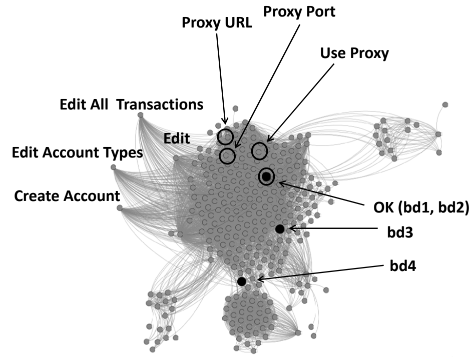
edges during some iterations but never get to reach them in subsequent iterations. This is due to the highly context-sensitive nature of our GUIs, where a context-aware mapping entry is no longer valid for a subsequent iteration. Addressing these cases is a subject for future work.

Faults: In total, we discovered 34 new faults that have not been detected before. Table 5.5 provides the detail of faults detected only in the iterative phases of OME* (recall that the first phase of OME* is actually BL). These faults were only detected in the later iterations because the new model elements (i.e., EFG edges) were only able to reach by leveraging the information collected in the previous iterations.

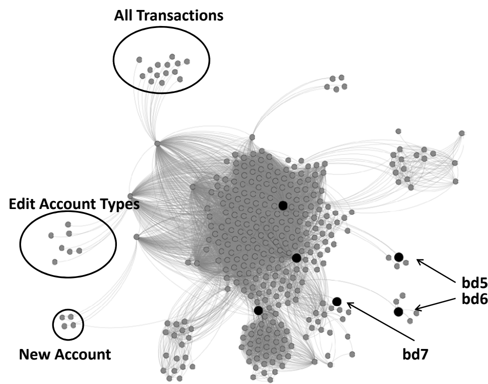
We also reported these faults to the developers of the subject applications. In response to our report, they have confirmed and fixed some of these faults in subsequent releases of the applications. There is only a few cases the reported faults were not fixed (e.g., in CrosswordSage). The reason is because at the time we reported the faults, the projects were no longer under an active development. More details of the faults detected can be found online at `http://www.cs.umd.edu/baonn/projects/guitar/bugs`.



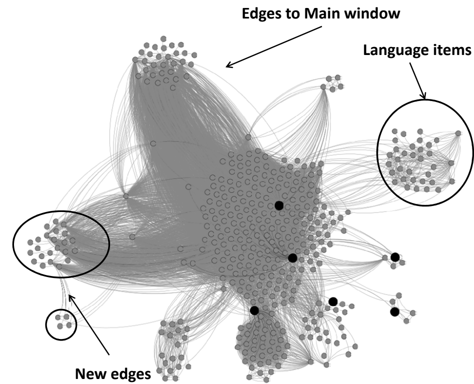
(a) BL: Initial Model.



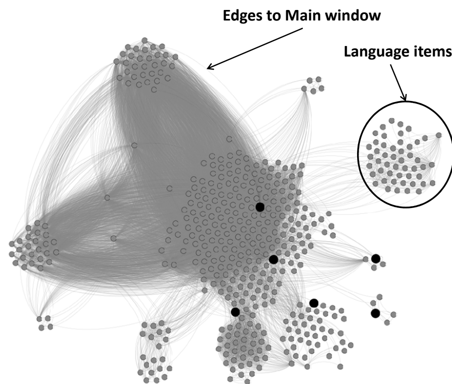
(b) Iteration 1.



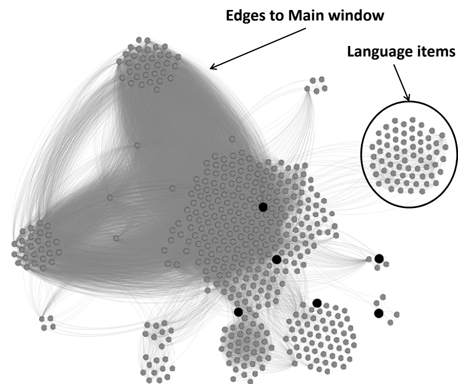
(c) Iteration 2.



(d) Iteration 3.



(e) Iteration 4.



(f) Iteration 5: Final Model.

Figure 5.2: OME* sees more of the EFG with each iteration.

Table 5.5: Summary of faults detected in the iterative phases of OME*

BugID	Iteration	Confirmed	Fixed	Description
AU4	1	✓	✓	ProfileException when using ‘/crash/crash’ as name to save a user profile
AU5	2	✓	✓	NullPointerException when changing layout of an empty Activity diagram
AU6	1	✓	✓	NullPointerException when trying to ‘Revert to Saved’ an unsaved document
AU7	2	✓	✓	InvalidObjectException when keeping the ‘Open Project’ window open and saving another project
BD1	1	✓	✓	IllegalArgumentException with an out-of-range Proxy Port
BD2	1	✓	✓	IllegalArgumentException when updating with an non-existing proxy address
BD3	1	✓	✓	InvalidValueException when creating a transaction with an empty name
BD4	1	✓	✓	FileNotFoundException when saving with a non-encrypted file with name containing special characters
BD5	2	✓	✓	FileNotFoundException when saving with an encrypted file with name containing special characters
BD6	2	✓	✓	ZipException when using a plugin name containing special characters
BD7	2	✓	✓	FileNotFoundException when configuring with a non-existing language file
CS1	1	×	×	FileNotFoundException when providing an invalid file name to save
CS2	1	×	×	NullPointerException when generating Write Clue with an empty crossword
CS3	1	×	×	NullPointerException when generating Suggest Word with an empty crossword
DJ1	1	✓	✓	IOException when saving file with a file name containing special characters
JR1	1	×	×	StringIndexOutOfBoundsException when using ”!#́” as a ‘Preview text’ in Preferences
JR2	1	✓	✓	ArrayIndexOutOfBoundsException when allowing to generate keyword for a disabled bibtex entry
JR3	1	✓	✓	ArrayIndexOutOfBoundsException when generating keyword for an already closed bibtex file

Continued on next page

Table 5.5 – Summary of faults detected in the iterative phases of OME* (continue)

BugID	Iteration	Confirmed	Fixed	Description
JR4	2	✓	✓	Pattern Exception when searching with an regular expression containing the special characters ‘ []’
JR5	2	×	×	IOException when setting default owner name containing a ‘{’ character
JR6	1	✓	✓	ClassNotFoundException when providing a non-existing class to setup the ‘Look and Feel’
JR7	1	✓	✓	ServerSocketException with an out-of-range Proxy port
JR12	1	✓	✓	ArrayIndexOutOfBoundsException when changing properties of an already closed bibtex file
JR13	2	×	×	StringIndexOutOfBoundsException when import with a non-exist ImportFormat plugin
OT1	1	✓	✓	NullPointerException when checking spell with a blank spell-checking plugin name
PS1	1	×	×	ParseException when providing a non-numeric value for the split after these pages text filed
PS2	1	×	×	ParseException when providing a non-numeric value for the split every “n” pages text field
PS3	1	×	×	ParseException when providing a non-numeric value for the split at this size text file
PS4	1	✓	✓	NullPointerException when providing an invalid split by bookmarks level
PS7	1	×	×	FileNotFoundException when saving with an invalid environment file name
RC1	2	✓	✓	NullPointerException when generating report with a non-existing file name
RC2	2	×	×	NullPointerException when adjusting time after switching to a previous day
RC3	1	✓	✓	NumberFormatException when entering a non-numberic value to the Inactivity time text field
RC7	3	×	×	NullPointerException when simultaneously adjusting starting time and correcting the scheduled time

5.7 Discussion

To better understand the role of dynamic input space exploration on fault detection and adequacy, we analyzed the faults that were detected only by the OME* technique. This section provides details of our analysis and findings.

A total of 7 new faults were detected in *Buddi*. All 7 were detected by the *withMap* technique. Of these 7, one fault was also detected by *noMap*. These faults are also indicated in Figure 5.2. The faulty events (i.e., the last events in the failed test cases) are solid dark nodes with their IDs (*bd*1–*bd*7) pointed by an arrow. We now discuss these faults and the test cases that detected them.

Fault *bd1* results in an *IllegalArgumentException* when setting an out-of-range proxy port for network configuration. It is detected by a test case consisting of 6 events: ⟨e_1: *Expand Edit menu*; e_2: *Open Preferences window*; e_3: *Select Network tab*; e_4: *Enable Use Proxy option*; e_5: *Enter a large number for proxy port*; e_6: *Click OK*⟩. This fault did not occur earlier in the *BL* iteration because, by default, the *Proxy Port* text box (i.e., e_5) is disabled. It is unable to change the proxy port unless the *Enable Use Proxy* check box is checked (i.e., performing e_4). However, this information is not available at *BL*. During the *BL* process, event e_5 was observed after the execution of event e_4. Hence, a new context aware mapping entry was created to reach e_5. In the next iteration, test cases were generated to cover e_5. One of them led to the failure. Fault *bd*2 is similar to *bd*1 except that it throws an *UnknownHostException* when using a non-existing proxy URL (with a valid port).

This is a particularly interesting test case because event e_6, i.e., *Click OK*, the

one that revealed the failure had been executed several times in *BL*. However, the failure was manifested only when e_6 executed in the context of e_5. Moreover, e_6 revealed a different failure, *bd*2, when executed after setting a non-existing proxy URL. All events in the fault-revealing test cases were available in the initial model (as marked in Figure 5.2(a)). However, the faults were revealed only when the events were tested in specific combinations.

The remaining faults in *Buddi* were detected due to the *discovery of new events*. For example, fault *bd3* causes an *InvalidValueException* when saving a transaction with an empty name. In *Buddi*, a transaction can be saved only after a document change. However, during ripping, the events related to the save functionality (e.g., *Save*, *Save All* buttons) were all executed before any document changing event. Hence, the *Save Transaction* dialog, which is opened by these events, did not show up. In subsequent iterations, however, it was opened and tested in new executing contexts. As a result, the fault was detected.

Faults *bd5*, *bd6*, *bd7* were detected during Iteration 2 when new events were observed and exercised. Detecting faults in later iterations is not uncommon, as can be seen in Table 5.4. The reason for this is that exercising fault-revealing events requires going through multiple other events, performed in a sequence. By iteratively identifying the enabling/opening relationships between events, the OME* test case generator is able to compose test cases that reveal faults. Consider, for instance, Fault *rc*7 in *Rachota*; this was detected in Iteration 3. It is an uncaught *NullPointerException*, thrown when simultaneously adjusting the current date, and setting the starting time of an active task to empty (recall that *Rachota* is an

application for tracking project time). The event sequence leading to this fault is: $\langle$$e_{11}$: *Select a task*; e_{12}: *Start selected task*; e_{13}: *Expand Tool menu*; e_{14}: *Open Adjust start time window*; e_{15}: *Change to a previous date*; e_{16}: *Click OK to confirm the start time adjustment*$\rangle$. This sequence brings the GUI through a series of states where events are reached in a chain: first, e_{11} enables e_{12} in *BL*, and then e_{12} in turn enables e_{14} in Iteration 1. When performed in Iteration 2, e_{14} opens a new *Adjust starting time* window (e_{13} is known earlier to expand the *Tool* menu to reach e_{14}). This window contains the events allowing the user to adjust the starting time of the active task (e.g. e_{16}). Finally, the pair (e_{15}, e_{16}) is exercised in Iteration 3, leading to the exception.

We conclude our discussion of these faults by noting that due to the complexity of the GUI, the *state-based relationships between events* that led to failures are difficult to predict manually. Similarly, because GUI event handlers are often spread across multiple independent modules/classes [68], such faults cannot be detected by code analysis techniques such as static analysis.

Going back to our research questions, we were able to show that OME* is more effective than *BL*, when using FDE, EC, and CC as our metrics. Further, we implemented OME* in two ways, with and without the mapping, and showed that because *noMap* does not maintain the context-aware mapping to reach coverage elements, it yields many infeasible test cases. The mapping is key to the success of OME*.

5.8 Benchmark

The data, scripts and tools used in this study require thousands of computation-days and hundreds of person-hours to create. We packaged all of these experimentation artifacts into a benchmark and made it available online for the research community[6]. The purpose of this benchmark is to provide other researchers with a similar experimentation environment when they want to compare their work with OME*. The benchmark will allow them to objectively experiment with a common set of tools and subjects, using similar models, processes and experimentation assumptions.

We also provide detail documentation for artifacts in the benchmark. Researchers may reuse our data to perform new studies. Similarly, they can extend our experimentation tools and scripts to support their new testing techniques.

5.9 Summary

This chapter empirically studies the OME* testing paradigm. An extensive experiment on 8 open-source applications showed that OME* did much better compared to the current state-of-the-art. In some cases, we observed more than 200% improvement in the set of events that we executed. We also discovered 34 new faults that have not been detected before. This result confirmed our believe on the applicability of the new testing paradigm for modern, event-driven GUI application. To make our study more transparent and replicable for other researchers, we released our tools and experiment data as an online experimentation benchmark.

[6] `http://www.cs.umd.edu/~atif/OME`

Chapter 6

Conclusions

As software systems have grown increasingly complex, the testers are tasked with verifying that these systems function correctly. However, often the testers do not have a complete knowledge of the systems' overall input spaces, i.e., the spaces of all possible input that may be supplied to the systems. This problem is severely compounded in GUIs that have immense or even infinite input spaces. GUI testers routinely miss allowable event sequences, any of which may cause failures once the software is released. The tester may also fail to discover that the software implementation allows the execution of some disallowed sequences. In practice, testing GUI-based applications still remains largely an ad-hoc and labor-intensive activity.

In this dissertation, we show that GUI-based applications can be *effectively* and *efficiently* tested by systematically and incrementally leveraging the application runtime execution observations. To demonstrate this thesis we have developed a novel testing paradigm called *Observe-Model-Exercise* (OME*)*. The OME* testing paradigm iteratively *observes* the GUI runtime behaviors, incrementally *models* the GUI input space and automatically generates test cases to *exercise* the newly observed GUI elements. To realize OME*, we have developed a new model to capture the context-sensitive behaviors of GUI applications and algorithms to incrementally

explore the GUI input space and generate context-aware test cases. We have implemented OME* in the open-source GUITAR testing framework. GUITAR was used to conduct a comprehensive empirical study with 8 popular open-source GUI applications and detected 34 previously unknown faults.

In next sections, we further discuss OME* in more detail and provide directions for future work.

6.1 Discussion

We evaluate OME* in terms of its effectiveness and efficiency. Then we discuss some limitations of the approach.

6.1.1 Effectiveness

OME* is an effective testing approach. It improves the *fault detection effectiveness* and *test adequacy* in testing GUI-based applications. By leveraging the additional information available at runtime, OME* is able to more precisely and completely capture the GUI input space. As a result, more effective test cases are automatically generated to reach the "deep" parts of the GUI, e.g., those guarded by special event sequences and only available in a particular context. The empirical study in Chapter 5 demonstrated that OME* could improve the test coverage, both at the code level and the GUI level. It detected new faults in GUI elements which were not reachable or even not "seen" by the other state-of-the-art testing techniques. These faults, while missed by the testers, were often considered severe to the applications.

At the management level, OME* can aid in improving the quality of a software testing process. A valuable output of OME* is the formal model representing the overall GUI input space. With this concrete picture of the input space, test managers can define metrics to *quantitatively* control the software testing process. Such metrics would help the managers in objectively predicting and tracking the testing effectiveness to make multiple sorts of forecasts, judgments and trade-offs during the software life-cycle.

6.1.2 Efficiency

OME* is an efficient testing approach. It leverages the already-available data to improve the test coverage obtained. In GUI testing, the GUI states are always needed to collect to determine if a test case is passed or failed. Hence, the only overhead in OME* is the effort to further analyze these GUI states and expand the GUI model. As the effort to execute test cases often dominates the entire testing process, this overhead can be considered small and negligible.

OME* is *fully automated* in all phases of the testing process: from model creation, to test case generation, test case execution, and model enhancement. This approach therefore can be used without any human intervention throughout the software development life-cycle. Potentially, OME* can be implemented with a continuous integration system [69] to provide an end-to-end regression testing workflow [35, 70].

In traditional GUI testing, GUI test cases are often obtained by manually

writing automation scripts. With this labor-intensive process, the number of test cases created is often small. In contrast, OME* can automatically generate a large number of test cases; thus, it is able to cheaply cover a broad range of a system's behaviors. The testing efficiency can even be further improved by adopting a distributed testing workflow [71] to run test cases in parallel, as partially demonstrated in Chapter 5.

6.1.3 Limitations

The current implementation of OME* has certain known limitations. First, as in the case with all automated model-based testing techniques, the test suites obtained by OME* are not optimal. There are often many "redundant" test cases leading to the same faults. Second, the GUI input space models obtained do not perfectly represent the GUI behaviors as they are derived through dynamic analysis. As a result, test cases generated from these models might be infeasible, leading to false positive fault reports. Third, to avoid any human intervention, the current OME* implementation only relies on runtime analysis to construct the GUI model and completely ignores the application's specifications. On one hand, this strategy preserves OME* as a fully automated technique. On the other hand, it fails to incorporate application domain knowledge into the testing process. Exploring the trade-off between automated and manual model construction is something we need to consider in the future.

Most importantly, the OME* testing paradigm misses parts of the GUI. Hence,

the input spaces discovered by OME* are still incomplete. There are several reasons for this limitation:

1. *Special inputs are required:* Certain parts of the GUI are guarded by a specific input value. For example, in DrJava, the "Project Properties" window is triggered by the "Project Properties" menu item in the main window (see Figure 6.1(b)). However, initially, this menu item is disabled. It is only enabled if a specific project name is supplied in the "Open Project" window to open an existing project (Figure 6.1(a)). However, OME* uses a standard set of inputs for all text fields and none of them matches with an existing project name. Thus, the "Project Properties" menu item is never enabled during the testing process and as a result, the "Project Properties" window is never included in the final input space model.

2. *Stronger test adequacy criterion are required:* The OME* paradigm relies on the test case execution observations to explore the GUI input space. However, these test cases are generated based on a specific test adequacy criteria. If there are parts of the GUI only requires a stronger criteria, they will be missed. For example, in ArgoUML, the "New To Do Item" window is triggered by the "To Do" button in the main window (Figure 6.2). This button is only enabled by the event triple $\langle e_1$: *Insert a class object*, e_2: *Select the object*, e_3: *Select ToDo Item tab*$\rangle$. However, to meet the pairwise event interaction testing criteria, OME* only attempts to generate test cases to cover all possible event pairs. Therefore, it never executes these three events together and the "New To Do

Item" window is missed.

3. *Non-GUI operations are required:* Sometimes, the GUI elements are triggered by a non-GUI event. Because the test cases in OME* are restricted to sequences of GUI events, these elements will be missed. For example, in Rachota, the "User inactivity detected" warning window only shows up after the user does not interact with the keyboard for a specified period of time (Figure 6.3). The event to trigger this window is actually a timing event instead of a GUI event. In this case, OME* is unable to detect the window.

In the future, the OME* paradigm should be improved to overcome the above limitations. Next section will discuss in more about our future work.

6.2 Future Work

We discuss our future work in short, medium and long terms.

In the immediate future, we will extend our subject application pool. In particular, we want to use non-Java, non-desktop (e.g., web, mobile) as well as industrial applications to reduce the external threats to validity in our empirical studies. Furthermore, we used natural faults (i.e., crashes) to measure fault detection effectiveness. This approach, on one hand, provides evidence that our technique can detect actual faults. On the other hand, we are limited in the analysis that we can perform. For example, we cannot examine faults that were missed. For this reason, we will seed artificial faults in future work. Cross empirical studies between seeded faults and natural faults can provide more insight to our approach. In a similar line,

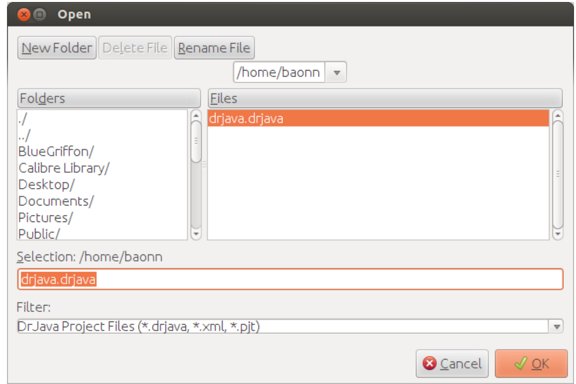
we want to use different types of test oracles such as GUI state matching [38] or invariant checking [24] to evaluate the results. Finally, for input supplied to text fields, we intend to use domain specific data instead of our current general purpose database. One possible direction is to leverage some existing static analysis tools to analyze source code and generate application specific text input, similar to what has been done in a recent work for JavaScript testing [6].

In the medium term, we will apply our paradigm to other test case generation techniques (e.g., capture/replay, AI planning) as well as other test adequacy criteria (e.g., event system interaction [40] and event context [45] coverages). Also, as partially shown in Chapter 5, there is a correlation between the model exploration phase and test case generation phase. It would be insightful to study the impacts of test case generation on model creation and vice versa. Finally, in this work, the GUI states collected are only used to explore and expand the input space. In the future, we also plan to leverage those valuable information as feedback for test case selection, similar to the work we have done in the past [22].

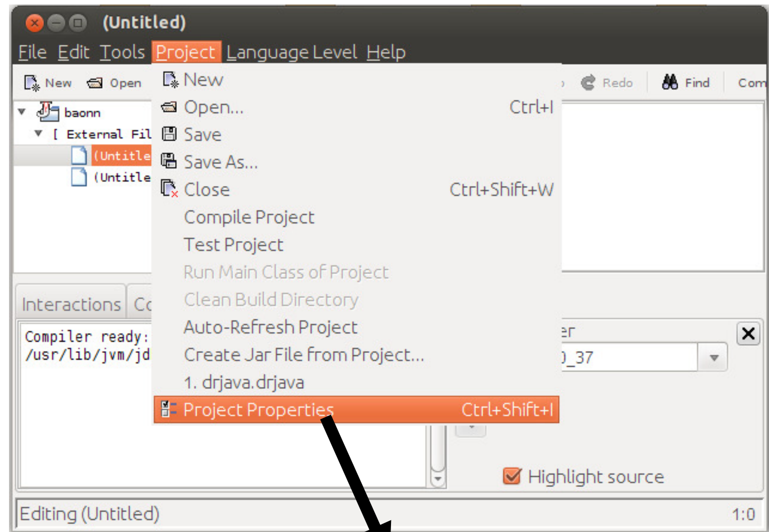
Some of the challenges of GUI testing are also relevant to testing of event-driven software [29]. In the long term, we will explore similar techniques for these application domains too. For example, we will apply our techniques to test web applications and object-oriented software as they also share similar input space characteristics with GUI applications. One way to test these classes of software is to generate test cases that are sequences of events (e.g., web user actions and method calls). Some of the techniques developed in this research may be used to better test these systems. Another direction for future work is studying different

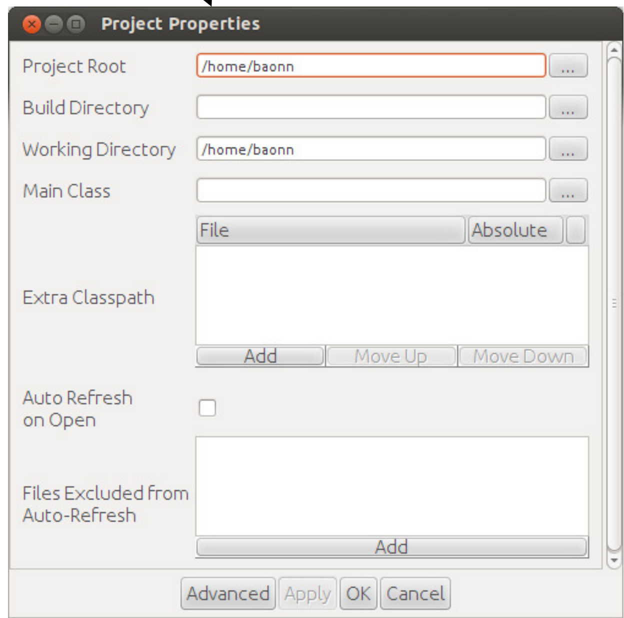
GUI input space exploration strategies. For example, the initial model created by GUI ripping may significantly influence the subsequent GUI input space exploration results. If we start the OME* process with different application configurations, we may get different results. To address this bias, we can rip the GUI multiple times, each for a different configuration. The specification documents can also be employed to maximize the spectrum of explored GUI behaviors.

In summary, in this dissertation, we have shown that the combination of dynamic input space exploration and model-based testing can cope well with the challenges in testing GUI applications. To provide focus, we have limited our discussion in the context of GUI testing. However, the general idea of OME* can be broadly extended for testing applications in other domains with similar, hard-to-determine input spaces. Examples of such domains include object-oriented programs [9], component-based systems [28, 72], and distributed systems [73]. In the future, we will empirically explore the applications of our approach on these domains.



(a) The "Open Project" window





(b) Opening the "Project Properties" window

Figure 6.1: Reaching the "Project Properties" window in DrJava.

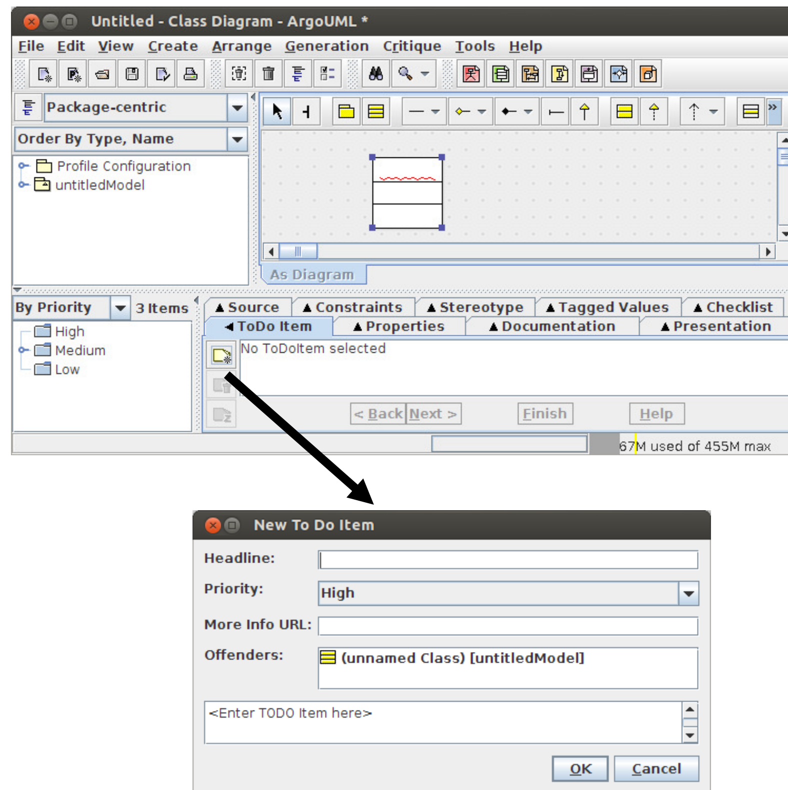


Figure 6.2: Reaching the “New To Do Item” window in ArgoUML.

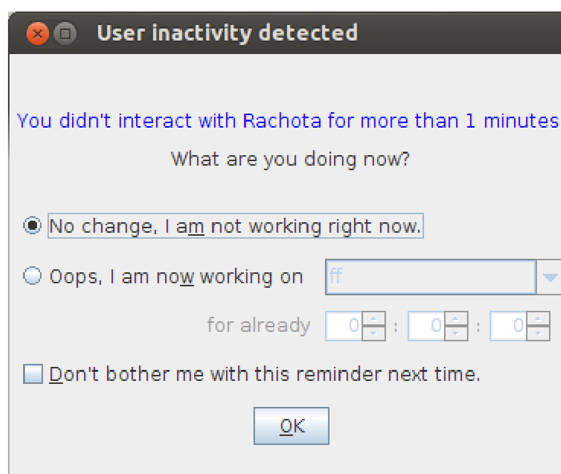


Figure 6.3: The user inactivity warning window in Rachota.

Bibliography

[1] Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. Feedback-directed random test generation. In *ICSE'07, Proceedings of the 29th International Conference on Software Engineering*, pages 75–84, Minneapolis, MN, USA, May 23–25, 2007.

[2] Cristian Cadar, Daniel Dunbar, and Dawson Engler. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, OSDI'08, pages 209–224, Berkeley, CA, USA, 2008. USENIX Association.

[3] Koushik Sen, Darko Marinov, and Gul Agha. CUTE: a concolic unit testing engine for C. *SIGSOFT Softw. Eng. Notes*, 30:263–272, September 2005.

[4] Walter I. Wittel, Jr. and Theodore G. Lewis. Integrating the MVC Paradigm into an Object-Oriented Framework to Accelerate GUI Application Development. Technical Report 91-60-06, Department of Computer Science, Oregon State University, 1991.

[5] J. Bach. What is exploratory testing. *E. van Veenendaal, The Testing Practitioner–2nd edition, UTN Publishing*, pages 90–72194, 2004.

[6] Prateek Saxena, Devdatta Akhawe, Steve Hanna, Feng Mao, Stephen McCamant, and Dawn Song. A Symbolic Execution Framework for JavaScript. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, SP '10, pages 513–528, Washington, DC, USA, 2010. IEEE Computer Society.

[7] Patrice Godefroid, Michael Y. Levin, and David Molnar. SAGE: whitebox fuzzing for security testing. *Commun. ACM*, 55(3):40–44, March 2012.

[8] Wilbert O. Galitz. *The essential guide to user interface design: an introduction to GUI design principles and techniques.* John Wiley & Sons, Inc., New York, NY, USA, 1997.

[9] Sai Zhang, David Saff, Yingyi Bu, and Michael D. Ernst. Combined static and dynamic automated test generation. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ISSTA '11, pages 353–363, New York, NY, USA, 2011. ACM.

[10] James A. Whittaker. *Exploratory Software Testing: Tips, Tricks, Tours, and Techniques to Guide Test Design.* Addison-Wesley Professional, 1st edition, 2009.

[11] José L. Silva, José Creissac Campos, and Ana C. R. Paiva. Model-based User Interface Testing With Spec Explorer and ConcurTaskTrees. *Electron. Notes Theor. Comput. Sci.*, 208:77–93, 2008.

[12] Theodore D. Hellmann and Frank Maurer. Rule-Based Exploratory Testing of Graphical User Interfaces. In *Proceedings of the 2011 Agile Conference*, AGILE '11, pages 107–116, Washington, DC, USA, 2011. IEEE Computer Society.

[13] Juha Itkonen, Mika V. Mantyla, and Casper Lassenius. How do testers do it? An exploratory study on manual testing practices. In *Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement*, ESEM '09, pages 494–497, Washington, DC, USA, 2009. IEEE Computer Society.

[14] James H Hicinbothom and Wayne W Zachary. *A Tool for Automatically Generating Transcripts of Human-Computer Interaction*, volume 2, page 1042. 1993.

[15] Mark Utting and Bruno Legeard. *Practical Model-Based Testing: A Tools Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2007.

[16] Atif M. Memon and Bao N. Nguyen. Advances in automated model-based system testing of software applications with a GUI front-end. In Marvin V. Zelkowitz, editor, *Advances in Computers*, volume 80, pages nnn–nnn. Academic Press, 2010.

[17] Jonathan Jacky, Margus Veanes, Colin Campbell, and Wolfram Schulte. *Model-Based Software Testing and Analysis with C#*. 1 edition.

[18] Atif M. Memon, Adithya Nagarajan, and Qing Xie. Automating regression testing for evolving GUI software. *Journal of Software Maintenance*, 17(1):27–64, 2005.

[19] A. Memon, I. Banerjee, and A. Nagarajan. GUI ripping: Reverse engineering of graphical user interfaces for testing. In *Reverse Engineering, 2003. WCRE 2003. Proceedings. 10th Working Conference on*, pages 260–269, 2003.

[20] Alan Page and Ken Johnston. *How We Test Software at Microsoft*. Microsoft Press, 2008.

[21] Jaymie Strecker and Atif M. Memon. Accounting for Defect Characteristics in Evaluations of Testing Techniques. *ACM Trans. on Softw. Eng. and Method.*, NN(N), 2011.

[22] Xun Yuan and Atif M. Memon. Generating event sequence-based test cases using GUI runtime state feedback. *IEEE Transactions on Software Engineering*, 36:81–95, 2010.

[23] David Martin, John Rooksby, Mark Rouncefield, and Ian Sommerville. 'Good' Organisational Reasons for 'Bad' Software Testing: An Ethnographic Study of Testing in a Small Software Company. In *Proceedings of the 29th international conference on Software Engineering*, ICSE '07, pages 602–611, Washington, DC, USA, 2007. IEEE Computer Society.

[24] Ali Mesbah and Arie van Deursen. Invariant-based automatic testing of AJAX user interfaces. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 210–220, Washington, DC, USA, 2009. IEEE Computer Society.

[25] Shay Artzi, Julian Dolby, Simon Holm Jensen, Anders Møller, and Frank Tip. A framework for automated testing of javascript web applications. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 571–580, New York, NY, USA, 2011. ACM.

[26] Valentin Dallmeier, Nikolai Knopp, Christoph Mallon, Sebastian Hack, and Andreas Zeller. Generating test cases for specification mining. In *Proceedings of the 19th international symposium on Software testing and analysis*, ISSTA '10, pages 85–96, New York, NY, USA, 2010. ACM.

[27] Neil Walkinshaw, John Derrick, and Qiang Guo. Iterative Refinement of Reverse-Engineered Models by Model-Based Testing. In *FM '09: Proceedings of the 2nd World Congress on Formal Methods*, pages 305–320, Berlin, Heidelberg, 2009. Springer-Verlag.

[28] Cesare Bartolini, Antonia Bertolino, Sebastian Elbaum, and Eda Marchetti. Bringing white-box testing to Service Oriented Architectures through a Service Oriented Approach. *J. Syst. Softw.*, 84:655–668, April 2011.

[29] Atif M. Memon. An event-flow model of GUI-based applications for testing. *Softw. Test. Verif. Reliab.*, 17:137–157, September 2007.

[30] Sadik Esmelioglu and Larry Apfelbaum. Automated Test Generation, Execution, and Reporting. In *Proceedings of Pacific Northwest Software Quality Conference*. IEEE Press, Oct 1997.

[31] R. K Shehady and D. P. Siewiorek. A Method to Automate User Interface Testing Using Variable Finite State Machines. In *Proceedings of The Twenty-Seventh Annual International Symposium on Fault-Tolerant Computing (FTCS'97)*, pages 80–88, Washington - Brussels - Tokyo, June 1997. IEEE Press.

[32] Lee White and Husain Almezen. Generating test cases for GUI responsibilities using complete interaction sequences. In *Proceedings of the International Symposium on Software Reliability Engineering*, pages 110–121, October 8–11 2000.

[33] Fevzi Belli. Finite-State Testing and Analysis of Graphical User Interfaces. In *ISSRE '01: Proceedings of the 12th International Symposium on Software Reliability Engineering*, page 34, Washington, DC, USA, 2001. IEEE Computer Society.

[34] Myra B. Cohen, Peter B. Gibbons, Warwick B. Mugridge, and Charles J. Colbourn. Constructing test suites for interaction testing. In *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, pages 38–48, Washington, DC, USA, 2003. IEEE Computer Society.

[35] Atif M. Memon and Qing Xie. Studying the Fault-Detection Effectiveness of GUI Test Cases for Rapidly Evolving Software. *IEEE Trans. Softw. Eng.*, 31(10):884–896, 2005.

[36] Frances E. Allen. Control flow analysis. *SIGPLAN Not.*, 5(7):1–19, 1970.

[37] Atif M. Memon, Martha E. Pollack, and Mary Lou Soffa. Hierarchical GUI Test Case Generation Using Automated Planning. *IEEE Transactions on Software Engineering*, 27(2):144–155, February 2001.

[38] Qing Xie and Atif M. Memon. Designing and comparing automated test oracles for GUI-based software applications. *ACM Trans. Softw. Eng. Methodol.*, 16(1):4, 2007.

[39] Lee J. White. Regression Testing of GUI Event Interactions. In *ICSM '96: Proceedings of the 1996 International Conference on Software Maintenance*, pages 350–358, Washington, DC, USA, 1996. IEEE Computer Society.

[40] Qing Xie and Atif M. Memon. Using a Pilot Study to Derive a GUI Model for Automated Testing. *ACM Trans. on Softw. Eng. and Method.*, 2008.

[41] David J. Kasik and Harry G. George. Toward automatic generation of novice user test scripts. In *CHI '96: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 244–251, New York, NY, USA, 1996. ACM.

[42] Penelope A. Brooks and Atif M. Memon. Automated GUI testing guided by usage profiles. In *ASE '07: Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 333–342, New York, NY, USA, 2007. ACM.

[43] Renée C. Bryce, Charles J. Colbourn, and Myra B. Cohen. A framework of greedy methods for constructing interaction test suites. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 146–155, New York, NY, USA, 2005. ACM.

[44] David M. Cohen, Siddhartha R. Dalal, Michael L. Fredman, and Gardner C. Patton. The AETG System: An Approach to Testing Based on Combinatorial Design. *IEEE Transactions on Software Engineering*, 23(7):437–444, 1997.

[45] Xun Yuan, Myra B. Cohen, and Atif M. Memon. GUI interaction testing: Incorporating event context. *IEEE Transactions on Software Engineering*, 37(4):559–574, 2011.

[46] D. M. Cohen, S. R. Dalal, A. Kajla, and G. C. Patton. The automatic efficient test generator (AETG) system. In *Proceedings of the Fifth International Symposium on Software Reliability Engineering*, pages 303–309. IEEE Computer Society Press, November 1994.

[47] Antti Kervinen, Mika Maunumaa, Tuula Pakkonen, and Mika Katara. Model-based testing through a GUI. In *In Proceedings of the 5th International Workshop on Formal Approaches to Testing of Software (FATES 2005), number 3997 in Lecture Notes in Computer Science*, pages 16–31. Springer, 2006.

[48] A.C.R. Paiva, N. Tillmann, J.C.P. Faria, and R. Vidal. Modeling and testing hierarchical GUIs. *Proceedings of ASM 2005*, 2005.

[49] A.C.R. Paiva, J.C.P. Faria, and R.F.A.M. Vidal. Towards the integration of visual and formal models for GUI testing. *Electronic Notes in Theoretical Computer Science*, 190(2):99–111, 2007.

[50] H Buwalda. Action Figure. *STQE Magazine*, 2003.

[51] M. Barnett, K.R.M. Leino, and W. Schulte. The Spec# programming system: An overview. *Lecture Notes in Computer Science*, 3362:49–69, 2005.

[52] Luciano Baresi and Michal Young. Test Oracles. Technical Report CIS-TR-01-02, University of Oregon, Dept. of Computer and Information Science, Eugene, Oregon, U.S.A., August 2001.

[53] Scott McMaster and Atif M. Memon. An Extensible Heuristic-Based Framework for GUI Test Case Maintenance. In *TESTBEDS'09: Proceedings of the First International Workshop on TESTing Techniques & Experimentation Benchmarks for Event-Driven Software*, Washington, DC, USA, 2009. IEEE Computer Society.

[54] Andrea Adamoli, Dmitrijs Zaparanuks, Milan Jovic, and Matthias Hauswirth. Automated GUI performance testing. *Software Quality Control*, 19:801–839, December 2011.

[55] Alex Ruiz and Yvonne Wang Price. GUI Testing Made Easy. In *Proceedings of the Testing: Academic & Industrial Conference - Practice and Research Techniques*, pages 99–103, Washington, DC, USA, 2008. IEEE Computer Society.

[56] Luca de Alfaro and Thomas A. Henzinger. Interface theories for component-based design. In *Proceedings of the First International Workshop on Embedded Software*, EMSOFT '01, pages 148–165, London, UK, UK, 2001. Springer-Verlag.

[57] Scott W. Ambler. *The Elements of UML 2.0 Style.* Cambridge University Press, 2005.

[58] Xun Yuan and Atif M. Memon. Iterative execution-feedback model-directed GUI testing. *Information and Software Technology*, 52(5):559–575, 2010.

[59] Atif M. Memon, Mary Lou Soffa, and Martha E. Pollack. Coverage criteria for GUI testing. In *Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering*, volume 26 of *ESEC/FSE-9*, pages 256–267, New York, NY, USA, September 2001. ACM.

[60] Scott McMaster and Atif Memon. Call-Stack Coverage for GUI Test Suite Reduction. *IEEE Transactions on Software Engineering*, 34:99–115, 2008.

[61] Atif M. Memon. Automatically Repairing Event Sequence-Based GUI Test Suites for Regression Testing. *ACM Trans. on Softw. Eng. and Method.*, 2008.

[62] Si Huang, Myra Cohen, and Atif M. Memon. Repairing GUI Test Suites Using a Genetic Algorithm. In *ICST 2010: Proceedings of the 3rd IEEE International Conference on Software Testing, Verification and Validation*, Washington, DC, USA, 2010. IEEE Computer Society.

[63] Amanda Swearngin, Myra B. Cohen, Bonnie E. John, and Rachel K. E. Bellamy. Easing the Generation of Predictive Human Performance Models from Legacy Systems. In *Proceedings of the 2011 annual conference on Human factors in computing systems*, 2012. to appear.

[64] Chin-Yu Huang, Jun-Ru Chang, and Yung-Hsin Chang. Design and analysis of GUI test-case prioritization using weight-based methods. *J. Syst. Softw.*, 83(4):646–659, April 2010.

[65] Y. Huang and L. Lu. Apply ant colony to event-flow model for graphical user interface test case generation. *IET Software*, 6(1):50–60, 2012.

[66] Leonardo Mariani, Mauro Pezzè, Oliviero Riganelli, and Mauro Santoro. AutoBlackTest: a tool for automatic black-box testing. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 1013–1015, New York, NY, USA, 2011. ACM.

[67] Fevzi Belli, Christof J. Budnik, and Lee White. Event-based modelling, analysis and testing of user interactions: approach and case study: Research Articles. *Softw. Test. Verif. Reliab.*, 16:3–32, March 2006.

[68] Atanas Rountev, Scott Kagan, and Michael Gibas. Evaluating the imprecision of static analysis. In *Proceedings of the 5th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, PASTE '04, pages 14–16, New York, NY, USA, 2004. ACM.

[69] Jez Humble and David Farley. *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. Addison-Wesley Professional, 1st edition, 2010.

[70] David Saff and Michael D. Ernst. An experimental evaluation of continuous testing during development. *SIGSOFT Softw. Eng. Notes*, 29(4):76–85, July 2004.

[71] Adam Porter, Cemal Yilmaz, Atif M. Memon, Douglas C. Schmidt, and Bala Natarajan. Skoll: A Process and Infrastructure for Distributed Continuous Quality Assurance. *IEEE Transactions on Software Engineering*, 33:510–525, 2007.

[72] Ilchul Yoon, Alan Sussman, Atif Memon, and Adam Porter. Testing Component Compatibility in Evolving Configurations. *Information and Software Technology*, 55(2):445–458, 2013.

[73] Lionel C. Briand, Yvan Labiche, and Johanne Leduc. Toward the Reverse Engineering of UML Sequence Diagrams for Distributed Java Software. *IEEE Transactions on Software Engineering*, 32:642–663, 2006.