ABSTRACT

Title of dissertation:	Feedback-Directed Model-Based GUI Test Case Generation
	Xun Yuan, Doctor of Philosophy, 2008
Dissertation directed by:	Professor Atif M. Memon Department of Computer Science

Most of today's software users interact with the software through a graphical user interface (GUI), which is a representative of the broader class of event-driven software (EDS). As the correctness of the GUI is necessary to ensure the correctness of the overall software, its quality assurance (QA) is becoming increasingly important. During software testing, an important QA technique, test cases are created and executed on the software. For GUIs, test cases are modeled as sequences of user input events. Because each possible sequence of user events may potentially be a test case and because today's GUIs offer enormous flexibility to end users, in principle, GUI testing requires a prohibitively large number of test cases. Any practical test case generation technique must sample the vast GUI input space. Existing techniques are either extremely resource intensive or do not adequately model complex GUI behaviors, thereby limiting fault detection.

This research develops new models, algorithms, and metrics for automated GUI test case generation. A novel aspect of this work is its use of software run-time information collected as feedback during GUI test case execution, and used to generate additional test cases that model complex GUI behaviors. One set of empirical studies show that the feedback-directed technique significantly improves upon existing techniques and helps to identify serious problems in fielded GUIs. Another set of studies conducted on in-house software applications show that the test suites generated by the new technique outperform their coverage equivalent counterparts in terms of fault detection.

Although the focus of this work is on the GUI domain, the techniques developed are general and are applicable to the broader class of EDS. In fact, this work has already had an impact on research and practice of testing other EDS. In particular, the work has been extended by other researchers to test web applications.

Feedback-Directed Model-Based GUI Test Case Generation

by

Xun Yuan

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
2008

Acknowledgments

I owe my gratitude to all the people who have made this thesis possible and because of whom my graduate experience has been one that I will cherish forever.

First and foremost I'd like to thank my advisor, Professor Atif M. Memon for giving me an invaluable opportunity to work on challenging and extremely interesting projects over the past six years. He has always made himself available for help and advice and there has never been an occasion when I've knocked on his door and he hasn't given me time. It has been a pleasure to work with and learn from such an extraordinary individual.

I would also like to thank my previous advisor, Professor Willam A. Arbaugh for bring me to University of Maryland, College Park, and providing me everything I need for classes and doing research. He is always like a good friend of me.

My colleagues at the GUITAR group and Skoll team have enriched my graduate life in many ways and deserve a special mention. Xie Qing helped me start-off by introducing the basic tools, running environments and her valuable experiences. Gan Bin provided help by implementing and approving various new tools which are crucial to the later experiments. My interaction with Cyntrica Eaton, Jaymie Strecker, Scott McMaster, Penelope Brook and Bao Nguyen has been very fruitful.

I owe my deepest thanks to my family - my dear mother, father, husband and two sisters who have always stood by me and guided me through my career, and have pulled me through against impossible odds at times. Words cannot express the gratitude I owe them.

It is impossible to remember all, and I apologize to those I've inadvertently left out.

Lastly, thank you all!

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation

As computers and embedded devices play an increasingly important role aiding end users, researchers, and businesses in today's inter-networked world, several classes of *event-driven software* (EDS) applications are becoming ubiquitous. Common examples include graphical user interfaces (GUIs), web applications, network protocols, embedded software, software components, and device drivers [35]. An EDS takes internal/external events (*e.g.*, commands, messages) as input (*e.g.*, from users, other applications), changes its state, and sometimes outputs an event sequence [39]. An EDS is typically implemented as a collection of event handlers designed to respond to individual events. Nowadays, EDS is gaining popularity because of the advantages this "event-handler architecture" offers to both developers and users. From the developer's point of view, the event handlers may be created and maintained fairly independently; hence, complex system may be built using these loosely coupled pieces of code. In interconnected/distributed systems, event handlers may also be distributed, migrated, and updated independently. From the user's point of view, EDS offers many degrees of usage freedom. For example, in GUIs, users may choose to perform a given task by inputing GUI events (mouse clicks, selections, typing in text-fields) in many different ways in terms of their type, number and execution order.

Quality assurance (QA) is becoming increasingly important for EDS as its correctness may affect the quality of the entire system in which the EDS operates. *Software testing* is a popular QA technique employed during software development and deployment to help improve its quality. During software testing, test cases are created and executed on the software. One way to test an EDS is to execute each event individually and observe its outcome, thereby testing each event handler in isolation. However, the execution outcome of an event handler may depend on its internal state, the state of other entities (objects, event handlers) and/or the external environment. Its execution may lead to a change in its own state or that of other entities. Moreover, the outcome of an event's execution may vary based on the sequence of preceding events seen thus far. Consequently, in EDS testing, each event needs to be tested in different states. EDS testing therefore involves generating and executing sequences of events.

The event-driven nature of EDS creates several challenges for testing. One important challenge stems from the enormous space of possible event interactions with the EDS. Because each possible event sequence may potentially be a test case, EDS testing, in principle, may require a prohibitively large number of test cases. Practical EDS testing techniques attempt to sample the vast input space of all possible sequences with the goal of detecting faults; for effective testing, it is important to sample this space carefully.

This research develops new models, algorithms, and metrics for automated EDS test case generation. To provide focus, this research studies one sub-class of

EDS, *i.e.*, GUIs, which have became very popular as more and more software uses them as front-ends. Specifically, the GUIs that are studied in this research react to discrete events performed only by a single user and the events are deterministic, *i.e.*, their outcomes are completely predictable.

The remainder of this chapter introduces GUIs, GUI testing, existing GUI testing techniques, and presents an overview of the models, test case generation techniques, and processes developed in this research.

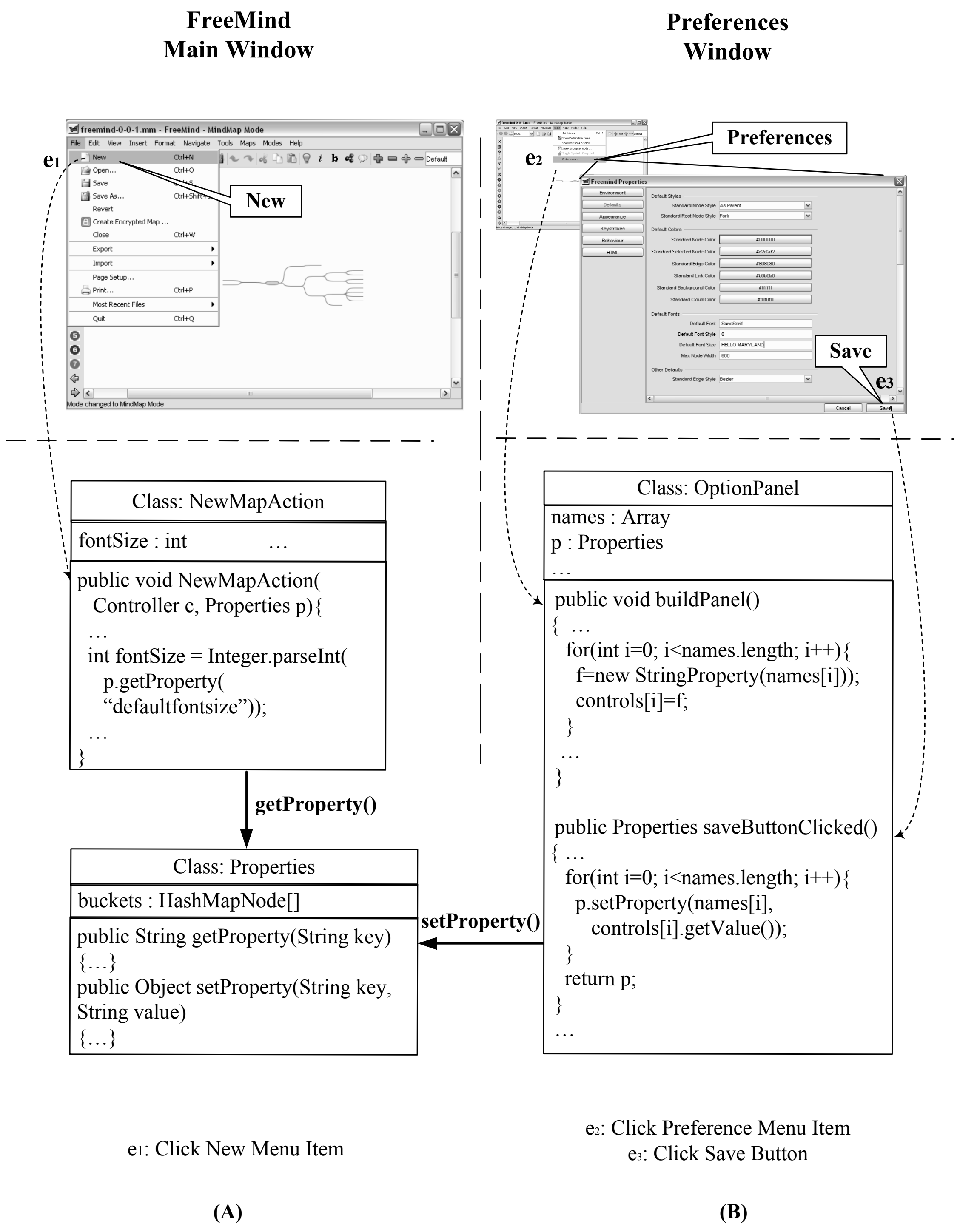
1.2 What is a GUI?

A GUI is a front-end to underlying "business logic." It allows an end user to perform complex tasks via familiar visual cues by executing events on GUI *widgets*. Typical examples of widgets include checkboxes, buttons, and text-fields with associated *events*: uncheck-checkbox, click-on-button and type-in-text-field. The GUI responds to the events by invoking corresponding event handlers, performing computation in the underlying code and presenting returned results, *e.g.*, by changing the appearance of some GUI widgets. In today's typical software applications, GUI code makes up 45-60% of overall application code [37].

As outlined above, the GUI provides a user-friendly middle layer between a software user and the underlying system, and it manages communication back and forth from each side. Because the GUI is often the only communication channel between a user and the underlying system, incorrect execution of the GUI may affect the execution of the underlying code. For example, data values passed incorrectly from the GUI to the underlying software may lead to failures.

1.3 Significance of GUI Testing

Due to the increasing popularity of the object-oriented programming paradigm, code for GUI event handlers and underlying business logic code is usually implemented in different packages, modules and classes; third-party event handlers (from libraries and/or open-source) are often incorporated into the code. Much of this code is integrated together at the GUI level. In fact, most of the modules interact with each other only through the GUI, as can be demonstrated via an example from a fielded GUI application called FreeMind (used later in Chapter 4) presented in Figure 1.1. The top half of this figure shows part of the GUI layer and the bottom half shows some of the code. Two windows and partial code are shown. The left window is the main window, where the event e_1 represents clicking the menu item `New` to create a new "mind map." The event handler for e_1 is `NewMapAction`; preferences (*e.g.*, font size) of the new mind map are obtained from a `Properties` object initialized from a file at application startup. The right-side window, invoked using event e_2, is the `Preferences` window, in which a user may set some fields of the `Properties` object and save them for future use using e_3. The handlers for these events are implemented in the `OptionPanel` class; the method `buildPanel` handles e_2 and the method `saveButtonClicked` handles e_3. It is important to note that the `NewMapAction` and `OptionPanel` classes interact with one another only via the GUI when a user performs e_2 followed by e_3 and then e_1; no other interaction is evident



```
Class: NewMapAction
fontSize : int          …
public void NewMapAction(
  Controller c, Properties p){
  …
  int fontSize = Integer.parseInt(
    p.getProperty(
    "defaultfontsize"));
  …
}
```

```
Class: Properties
buckets : HashMapNode[]
public String getProperty(String key)
{…}
public Object setProperty(String key,
String value)
{…}
```

```
Class: OptionPanel
names : Array
p : Properties
…
public void buildPanel()
{ …
  for(int i=0; i<names.length; i++){
    f=new StringProperty(names[i]));
    controls[i]=f;
  }
  …
}

public Properties saveButtonClicked()
{ …
  for(int i=0; i<names.length; i++){
    p.setProperty(names[i],
      controls[i].getValue());
  }
  return p;
}
…
```

Figure 1.1: Event Handlers Interact through GUI

at the code level.

Conventional testing of this FreeMind code may be done in two ways. The first is unit testing, in which each class/method is tested individually. Unit testing by its very nature is unable to test interactions between the classes. Any interactions are typically masked by using mock stubs during unit testing. The second, integration testing, tests multiple classes/methods together. During integration testing, a tester manually identifies sets of classes and methods that need to be tested together, *e.g.*, if they share a variable/object or invoke one another. However, it is difficult, in general, to determine which classes to test together. The class interactions may be indirect and there may be no obvious sharing of objects; the use of multi-language implementation, callbacks for event handlers, virtual function calls, and reflection in GUIs also makes it difficult to identify good candidates for integration testing. For example, the interactions between the classes in Figure 1.1 cannot be determined by simply examining the code.

Testing GUI-based software at the GUI level, in terms of sequences of events, has the advantage of exposing the software's work-flows, *i.e.*, as allowable sequences of events that may be executed on the software. In the example of Figure 1.1, the following fault was detected by GUI testing: a user opens the `Preferences` window by performing e_2, incorrectly inputs a text string (instead of an integer) in the `Default Font Size` text-field and saves the preference value in the `Properties` object by executing e_3. Later, the user tries to create a new mind map and performs e_1. While obtaining the current mind map preference setting, the handler for e_1 incorrectly assumes that the value for the `Default Font Size` property is an integer; invocation of the `Integer.parseInt` method on the non-integer value causes a failure. This example illustrates that while event handlers for GUIs are implemented as a collection of objects with no apparent interactions at the code level, the GUI layer helps to expose these interactions; hence GUI testing may be viewed as integration testing of this code.

1.4 The GUI Input Space Explosion Problem

This research focuses on a type of GUI testing that advocates generating event sequences and executing them as test cases on the GUI. The execution of a test case $< e_1; e_2; \cdots; e_n >$ may be visualized as starting in a GUI run-time state S_0, and transitioning through S_1, S_2, $\cdots$, S_n where S_i is the GUI state obtained after the execution of event e_i.[1] An instance of this visualization is shown in Figure 1.2. The circles represent GUI run-time states and directed edges represent event execution. The number next to each node indicates the number of events that may be performed in that state. The GUI starts in state S_0; a number of events (in fact 77 for the FreeMind application) may be performed in S_0 on the GUI, each resulting in a (potentially) new state. Subsequent events performed in any of these states will drive the GUI into new states or return to one of the existing states. In general, because states may be repeated, the tree structure shown in Figure 1.2 may be a directed graph. As the numbers next to each node indicate, the branching factor for GUI states is enormous, leading to a very large state space. This growth is typical

[1]The term "GUI state" will later (Chapter 3) be defined in terms of the GUI's constituent widgets.

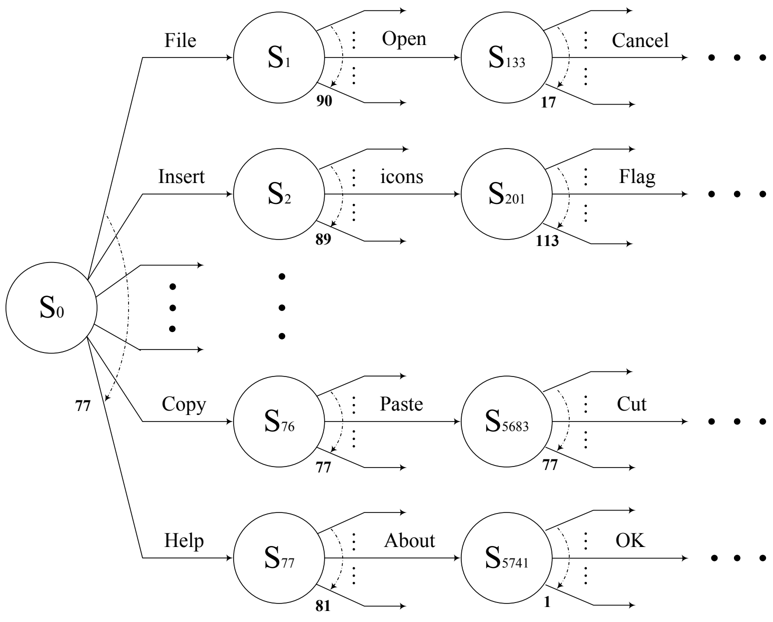


Figure 1.2: Example of Events and States in GUI Testing

for non-trivial GUIs; the number of event sequences grows exponentially with length.

Each existing GUI testing technique attempts to expose failures (*i.e.*, erroneous run-time states) given limited testing resources. In terms of Figure 1.2, each technique attempts to traverse, via sequences of events, states of the GUI. The next section presents an overview of some popular techniques used for GUI testing.

1.5 Existing GUI Testing Techniques

Several GUI testing techniques have been proposed by researchers; some have been implemented as tools and adopted by practitioners. All these techniques automate some aspects of GUI testing as shown in Table 1.1, including model creation (for those based on model-based testing), test case generation, test oracle[2] gen-

[2]A test oracle is used to determine whether or not a software executed correctly for a test case.

eration, test execution, and regression testing (rerunning selective test cases after software modifications). Although the nature and type of test cases may vary across techniques, all of them explore the GUI's state space via sequences of GUI events, attempting to expose faults.

Technique	Model Creation	Test Case Generation	Test Oracle Generation	Test Execution	Regression Testing
Unit Testing	N/A			√	
Capture/Replay Tools	N/A			√	
FSM model		√	√		
AI planning		√			
Genetic Algorithm		√			
Graph Model	√	√	√	√	√

Table 1.1: Automation (marked with √) in Existing GUI Testing Techniques

Unit Testing: Unit testing tools such as *JFCUnit*, *Abbot*, *Pounder* and *Jemmy Module* [1] are used to manually create unit test cases for GUIs. A unit test case consists of method calls to an instance of the class under test. Assertions are inserted in the test cases to determine whether the classes/methods executed correctly. The test cases are automatically executed on the GUI under test. Assertion violations are reported as failures. The parts of the GUI state space explored depends largely on the nature of the test cases.

Capture/replay Tools: Because manual coding of test cases can be tedious, a popular alternative "captures" sequences of events that testers perform manually on the GUI. Hence this technique treats a test case as a sequence of input events. These test cases can be "replayed" automatically on the GUI [24]. Tools used for this "capture" and "replay" are called *capture/replay* tools. As was the case with unit testing, the test case creation is manual (in terms of the event sequence) and the tools facilitate only the execution of test cases. The "goodness" of the test cases depends on the tester's ability to obtain fault-exposing sequences.

FSM Models: Model-based techniques have been used to automate GUI test case generation and (sometimes) test oracle creation. Several are based on *finite state machine* (FSM) models [49, 52]. With these models, GUI test cases may be automatically generated to cover the defined states. However, because these models are manually created by the tester, they end up being too small and thus too simplistic to model complex GUI behaviors. Therefore, the fault-detection effectiveness of generated test cases from these models is also limited.

AI Planning: Another technique, based on AI planning, models the infinite state space of a GUI and hence does not suffer from the "simplistic" model of states [38]. A description of the GUI is manually created by a tester; this description is in the form of *planning operators*, which model the preconditions and effects (post-conditions) of each GUI event. Test cases are automatically generated from tasks (pairs of initial and goal states) by invoking a planner, which searches for a path from the initial state to the goal state. However, the quality of the test cases is determined by the choice of tasks. Moreover, the manual operator definition and task selection may be expensive for large GUIs.

Genetic Algorithm: Another search technique, based on genetic algorithms, has been used for GUI testing [28]. The main focus of the work is to automatically generate test cases that mimic novice users. The approach requires that a tester first generate an initial event sequence manually; then genetic algorithms are used to generate "similar" sequences resembling the way a novice user would use the application. A manually defined fitness function directs test case generation.

Graph Models: Techniques based on *graph* models of the GUI's structure have recently been developed with the goal of minimizing manual work for testing. These techniques leverage a standard reverse engineering technique [36] to semi-automatically create the structural graph model. The most successful graph models that have been used for GUI test case generation include *Event Flow Graphs* (EFG) [34, 37] and *Event Interaction Graphs* (EIG) [40]. The nodes in these graphs represent GUI events; the edges represent different types of *structural* relationships between pairs of events. These structural relationships are based on the physical structure of the GUIs; short test cases are automatically generated, each covering an edge in the graph.

Summary: In summary, the most promising automated GUI testing technique developed thus far is based on the structural graph model of the GUI. However, it suffers from three major weaknesses. First, because it is restricted in its encoding of the GUI structure, it represents all possible executable paths in the GUI; generating and executing long test cases, *i.e.*, in terms of number of events, is impractical due to exponential growth. Second, because the graph models are not always accurate, they yield many unexecutable sequences. Finally, it uses a simplistic bounded depth-first traversal algorithm to generate 2-way covering test cases – all possible sequences of length two, *i.e.*, testing only *2-way interaction* between GUI event pairs. Using this algorithm for longer test cases, *i.e.*, testing *multi-way interaction*, is impractical. These weaknesses have resulted in limited application and adoption of the existing techniques.

1.6 Feedback-Directed GUI Test Case Generation

Each of these weaknesses is addressed in this research to develop new models, algorithms, and metrics, and combined into a new technique for automated GUI test case generation. The novel features of this technique are that it utilizes software run-time information as feedback to annotate important interactions between GUI events and helps to target testing to only these interacting event sets, improves the accuracy of the GUI's structural model, and explores the use of combinatorial techniques to generate test cases.

The technique is fully automatic. As the scale and complexity of modern software increases, testers are less willing to spend time creating and maintaining models; they are more likely to adopt a turn-key solution. The technique does not require source code analysis. Most of today's GUIs are created using widgets from libraries or third-party components; source code for these components is rarely available. Dependency on source code will severely limit the applicability of the technique. The technique is based on GUI models that are able to abstract manageable important parts of the input space so as to be able to generate long event sequences. For example, as discussed earlier in Section 1.5, the GUI structure alone is not sufficient. The technique is able to better address the GUI space space explosion problem and better sample the input space.

As mentioned earlier, the new technique is based on feedback from the execution of available test cases. The key motivation behind this idea is that run-time behavior of events helps to automatically determine whether an event influences another's execution; these *interacting* events are good candidates for testing together in longer sequences. For example, the interacting events *Cut*, *Copy*, *Paste* and *Select All* should be carefully tested together rather than with other non-interacting events such as *Open User Manual*. The feedback (run-time information) is in the form of the set of run-time widget properties.

This research uses the feedback in two ways. First, it identifies *event semantic interaction* (ESI) relations between pairs of GUI events (*i.e.*, one event influences another) and augments the EIG model by annotating it with these relationships. The annotated model is called the *Event Semantic Interaction Graph* (ESIG). Nodes in the ESIG represent GUI events involved in an ESI relationship and edges represent the corresponding ESI relationships. A graph-traversal algorithm is used to generate *multi-way* test cases from the ESIG model. These test cases test *multi-way interactions* among GUI events that are ESI related.

The second approach developed to use the ESI relationships is to alternate GUI test case generation and execution. This approach is called ALT. In ALT, GUI test cases are generated in batches, by leveraging GUI run-time information from a previously run batch and obtaining new ESI relationships between events sequences and events to obtain the next batch. Each successive batch consists of "longer" test cases that expand the state space to be explored, yet prune the "unimportant" states. The "alternating" nature of ALT also allows it to enhance the next batch by leveraging certain relationships (*e.g.*, one enables the other) between

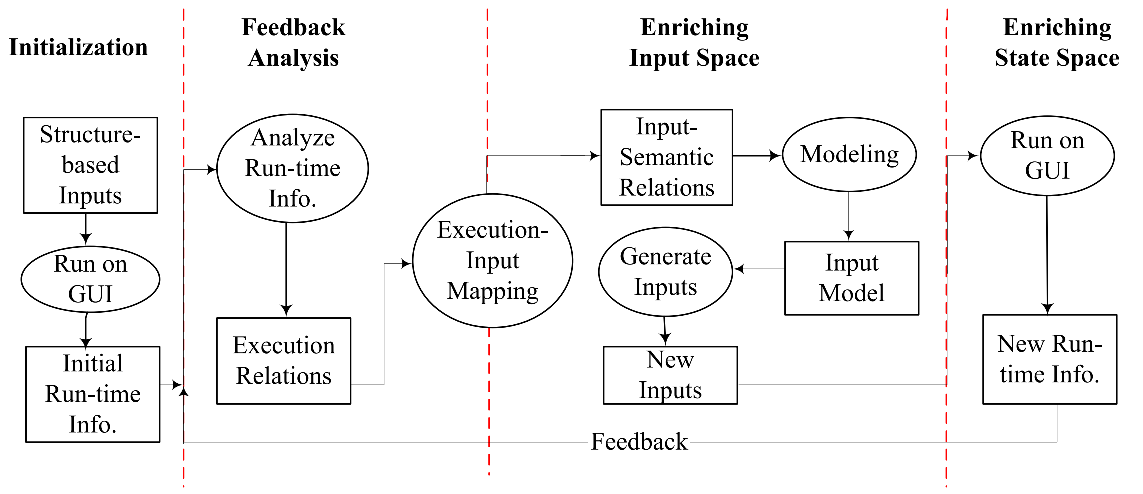


Figure 1.3: Overview of the Test Case Generation Process

GUI events that are revealed only at run-time and non-trivial to infer statically. This "anytime technique" continues iteratively, generating and executing additional test cases until resources are exhausted or testing goals have been met. The newly generated test cases are stronger than 2-way; generating these multi-way test cases is feasible because the underlying model is much smaller (in terms of event sequence) compared to the EIG model.

The new test case generation technique is summarized as a high-level process in Figure 1.3. The ovals in the figure represent activities, rectangles represent the resulting outputs from the activities which in turn are inputs for subsequent ones, and arrows show the direction of the work/data flow. As discussed earlier, the central feature of this process is the execution feedback (in the form of GUI run-time information) collection and analysis (loop). The process starts (in the **Initialization** phase) by generating and executing a seed suite on the GUI application under test;

the test cases in this suite are generated by using the existing structural GUI graph model. During the execution, GUI run-time information is recorded corresponding to each GUI event in each test case. This information is analyzed (in the **Feedback Analysis** phase) to determine the semantic interaction relationships between GUI events. The structural model, augmented with these relationships (in the **Enriching Input Space** phase) is used to selectively generate additional test cases that target only those new relationships and test multi-way GUI event interaction. The newly generated test cases are executed on the GUI (in the **Enriching State Space** phase) to collect additional run-time information, search for new ESI relationships and refine the model and obtain more test cases. This process continues to loop, generating new test cases that exercise newly discovered ESI relationships, until no new relationships are obtained, no more test cases are generated, testing goals have been met, or resources are exhausted.

Several studies of this new feedback-directed model-based test case generation technique have been conducted on two sets of GUI-based open-source software. The results demonstrate that the technique is able to significantly improve existing techniques and help identify/report serious problems in the open-source applications. When reported, these problems were fixed by the developers of the applications in subsequent versions. Finally, a new combinatorial interaction-based test case generation algorithm has also been explored.

1.7 Structure of the Dissertation

The next chapter presents an overview of the existing GUI testing techniques. The basic concepts of feedback, ESI relationships and their identification are described in Chapter 3. Chapter 4 presents and evaluates the ESIG-based test case generation approach. Chapter 5 presents the ALT approach. A preliminary exploration of combinatorial techniques for GUI test case generation is given in Chapter 6. Finally, Chapter 7 concludes with a discussion of broader impacts of the new technique.

Chapter 2

Background and Related Work

This is the first work that utilizes run-time information as feedback for model-based GUI test case generation. However, run-time information has previously been employed for various aspects of test automation, and model-based testing has been applied to conventional software as well as EDS. This chapter presents an overview of related research in the areas of model-based and EDS testing, GUI test case generation, and the use of execution feedback for test generation.

2.1 Model-Based Testing

Model-Based testing automates some aspect of software testing by employing a model of the software. The model is an abstraction of the software's behavior from a particular perspective (e.g., software states, configuration, values of variables, etc.); it may be at different levels of abstraction, such as abstract states, GUI states, internal variable states, or path predicates. Models may be derived from a formal specification of the software or reverse engineered by observing the software's execution behavior. They may be described using various languages and mathematical objects.

State Machine Models: The most popular models that have been used in software testing are *state machine models*. They model the software's behavior in terms of its abstract or concrete states; they are typically represented using state-transition diagrams. Several types of state machine models have been used for software testing, such as *Finite State Machine Models* (FSM) [4, 19, 25, 3], *UML Diagram-based Models* [33] and *Markov Chains* [27, 54].

For example, Microsoft researchers [4] modeled the control flow of an object-oriented software under test as an FSM and described it using the *Abstract State Machine Language* (AsmL `research.microsoft.com/fse/asml`). A traversal engine (part of Spec Explorer, a tool for advanced model-based specification and conformance testing), used the resulting finite state machine to produce behavioral tests to cover all explored transitions. Hong *et al.* also used FSMs for unit testing of classes in object-oriented programs; they used FSMs to model interactions between class data members and member functions [25]. The FSMs, called *class state machines* (CSM), were then transformed into *class flow graphs* (CFG); test case generation was done by selecting test cases according to the locations of definitions and uses of variables in the CFG. In other reported research, Farchi *et al.* used FSM models to test implementations of the *POSIX* standard and *Java* exception handling [19]. Both state machine models were created from the software specifications and represented using the *GOTCHA Definition Language* (GDL). The GOTCHA-TCBean test generator was then used to automatically explore the state space from the model and generate an abstract test suite.

Various extensions of FSMs have also been used for testing. These extensions use variables to represent *context* in addition to states; the goal is to limit the total number of states by using an orthogonal mechanism, in the form of explicit variables, to select state transitions. For example, an *extended finite state machine* (EFSM) is used by a tool called TestMaster [3] to generate test cases by traversing all paths from the start state to the exit state.

Because test cases for EDS are sequences of events, most practitioners and researchers have found it natural to use state machine models for testing EDS systems [12, 29, 32, 42]. The EDS is modeled in terms of states; events form the transitions between states. Algorithms traverse these machine models to generate sequences of events. For example, Campbell *et al.* have applied state machine models to test object-oriented reactive systems [12]. Object states were modeled in terms of instance variable values. Transitions were obtained from method invocations. Test cases were sequences of method calls and were generated by traversing the model.

Table-based Models: Table-based models define software behavior in the form of tables relating model elements such as system modes, conditions, events, and terms. These tables are then used as the basis for test case generation. The table-based modeling approach SCR, which is an abbreviation of *software cost reduction*, has been used for security functional testing [6] and *Mars Polar Lander* software to identify faults [7].

Grammars: Production grammars have been used to test large, complex and safety-critical software systems; a popular example is the Java Virtual Machine [50]. These grammars are collections of non-terminal to terminal mappings that resemble regular parsing grammars. A production grammar produces a program (*i.e.*, a set of terminals, or tokens) starting from a high-level description (*i.e.*, a set of non-terminals). The composition of the generated programs models the restrictions placed on the software by the production grammar.

Summary: The above model-based testing techniques rely heavily on the manual or semi-manual construction of the abstract model. Consequently, they are prone to errors. Moreover, any change to the software requires reconstruction of the model, which is typically cumbersome and time consuming.

2.2 GUI Test Case Generation

Several techniques have been developed for GUI test case generation. All of them use a model of the software and algorithms to generate test cases from the model.

State-Based Techniques: Finite State Machines (FSM) have been used to model GUIs [52, 5]. A GUI's state is represented in terms of its windows and widgets; each event triggers a transition in the FSM. A path in the FSM represents a test case. Due to the large number of possible states and transitions in modern GUIs, FSMs have scaling problems. Several attempts have been made to handle the scalability issue. For example, Belli [5] used an algorithm to convert FSM into equivalent regular expressions. The regular expressions were used to efficiently generate event sequences. Shehady *et al.* [49] proposed variable finite state machine (VFSM), which augmented a normal FSM with a number of global variables that can assume a finite number of values during the execution of a test case sequence. The value of each variable is used to determine the next state and output in response to an event.

Each transition may modify values of these variables.

As mentioned in Section 1.5, AI planning has been used to manage the state-space explosion by eliminating the need for explicit states. AI planning models the infinite state space of a GUI and hence does not suffer from the "simplistic" model of states [38]. A description of the GUI is manually created by a tester; this description is in the form of *planning operators*, which model the preconditions and effects (post-conditions) of each GUI event. Test cases are automatically generated from tasks (pairs of initial and goal states) by invoking a planner which searches for a path from the initial state to the goal state. However, the quality of the test cases is determined by the choice of tasks. Moreover, the manual operator definition and task selection may be expensive for large GUIs.

Genetic Algorithm: Test cases have been generated using genetic algorithms to mimic novice users [28]. The approach uses an expert to generate an initial event sequence manually and then uses genetic algorithm techniques to generate longer sequences. The assumption is that experts take a direct path when solving a problem via the GUI, whereas novice users take longer, indirect paths. Although useful for generating multiple test cases, the technique relies on an expert to generate the initial test case. The final test suite depends largely on the paths taken by the expert user. The idea of using a task and generating an initial test case may be better handled by using planning, because multiple test cases may be generated automatically according to some predetermined coverage criterion.

Directed Graph Models: In order to reduce manual work, several new systematic techniques based on graph models of the GUI have recently been developed.

The most successful graph models that have been used for GUI test case generation include *Event Flow Graphs* (EFG) [37] and *Event Interaction Graphs* (EIG) [40].

The EFG model [37] was the first GUI model used to fully automate GUI testing. It can be constructed semi-automatically using a reverse engineering technique called GUI Ripping [36]. The *GUI Ripper* automatically traverse a GUI under test and extracts the hierarchical structure of the GUI and events that may be performed on the GUI. The result of this process is the EFG. An EFG is a directed graph in which nodes represent GUI events and edges represent the *follows* relationship. An edge from node n_x to n_y means that the event represented by n_y may be performed *immediately* after the event represented by node n_x. Therefore, an EFG models all possible event sequences that may be executed on a GUI. GUI test cases, *i.e.*, sequences of events, correspond to *paths* in the EFG. A bounded depth-first graph traversal algorithm was used to generate test cases from an EFG.

The EIG model was derived from the EFG to improve the overall test process [40]. It was based on the observation that the code for certain types of events (*e.g.*, events that open pull-down menus) is straightforward and usually generated automatically by visual GUI-building tools. This code does not interact with code for other events; hence, one can expect that very few errors are revealed by executing interactions between these events. In contrast to the EFG, the EIG contains only certain types of events, and hence is more compact.

For both EFG and EIG, test cases are systematically generated to satisfy various types of adequacy criteria. One criterion (called the *event-interaction criterion* [40]) requires each edge in an EIG to be covered by at least one test case; test cases (called smoke tests) are generated by picking the two events on each edge and using a shortest-path algorithm to reach these events from the application's main window. Such techniques are automated and the algorithms always produce the same test suites, making the results repeatable.

Summary: FSM model and genetic algorithm-based GUI testing suffer from the problem of manual creation of the model, *i.e.*, state machine and fitness function respectively. The primary problem with the GUI graph models is that the number of event sequences grows exponentially with length. Hence, the existing graph-model-based GUI test case generation algorithms have only been able to generate test cases that cover all edges in the graph models, *i.e.*, they test *2-way* interactions between GUI events. Experiments have shown that some GUI faults can only be detected by test cases with more complex event interactions [40].

2.3 Execution Feedback for Test Case Generation

Execution feedback refers to information obtained dynamically during software execution. It has been used to guide automatic test case/test suite generation. This is called *dynamic test case generation* and was originally proposed by Miller and Spooner [43]. In their technique, software source code is instrumented to obtain execution feedback. The overall test case generation process starts by executing on an initial test case, which may be a test suite or a single test case. The execution feedback is collected and analyzed. The results are used to evaluate the "closeness" of the previous execution to the desired outcome; the model used to generate test cases is then modified accordingly and a new test case is generated. This loop stops when the "closeness" evaluation is satisfied according to some criterion.

Various types of execution feedback, models, and algorithms have been used for test case generation. For example, branch predicate evaluations along an execution path has been used with a gradient descent approach [30, 21, 22] and a chaining approach [20], condition-decision coverage has been used with genetic search [41], and object states have been used with a hybrid approach [58].

Branch Predicate Evaluations: Branch predicate evaluation refers to the flow of control during an execution. It has been used with the gradient descent approach to compute an input, *i.e.*, test case, that will execute a given path in the program [30, 21, 22]. It has also been used with the chaining approach to generate a test case that covers a selected statement [20]. The branch predicate evaluations, which encode control flow information, are collected during software execution on an initial test case. The generation of the test case is modeled as an object function minimization or optimization problem. The evaluation results are applied to gradually adjust the current test case so that it gets closer and closer to the desired test case. One disadvantage of these approaches is that they can get stuck in a local minima during test case generation.

Object Properties: Xie and Notkin have developed a feedback-based framework that uses object states to generate new test cases [58]. This framework integrates two techniques: (1) specification-based test generation and (2) dynamic specification inferences for test case generation. This integration provides value considerably beyond what the separate methods can provide alone.

Specification-based test generation is based on formal specifications, which express the desired behavior of a program. However, because formal specifications are difficult to obtain, dynamic specification inference attempts to infer specifications, in the form of operational abstractions, automatically from software execution. The discovered operational abstractions consist of object properties that hold for all the observed executions; these object properties are used to indicate the deficiency of test cases.

The test case generation process starts from an existing test suite. Through executions of these test cases, object states (values of variables and parameters, and return values) are recorded at the entry and exit of method executions. Based on the collected traces and a set of pre-defined axiom-pattern templates, equality patterns are searched to create operational abstractions. By removing or relaxing inferred preconditions on parameter values in the operational abstractions, both legal and illegal test cases are generated. The newly generated test cases are executed. Because they were generated by relaxing inferred preconditions, some of these test cases may cause an uncaught runtime exception. The other, non-crashing test cases are used to obtain new operational abstractions, which are again used to generate additional test cases.

Method-call Sequences: Pacheco *et al.* [44] have improved random unit test generation by incorporating feedback obtained from executing test inputs as they are created. They build inputs incrementally by randomly selecting a method call to apply and finding arguments from among previously-constructed inputs. The key idea of their work is that they build upon a legal sequence of method calls, each of whose intermediate objects is sensible and none of whose methods throw an exception indicating a problem. As soon as an input is built, it is executed and checked against a set of contracts and filters. The result of the execution determines whether the input is redundant, illegal, contract-violating, or useful for generating more inputs. The technique outputs a test suite consisting of unit tests for the classes under test. Passing tests can be used to ensure that code contracts are preserved across program changes; failing tests (that violate one or more contract) point to potential errors that should be corrected.

Similarly, Boyapati *et al.* employ a feedback-based technique to obtain all non-isomorphic inputs (test cases) for a method [8]. A programmer develops (1) a "guided test generation engine" that outputs test cases to explore the method's input space and (2) a predicate from the method's preconditions to check the validity of the generated input. This technique prunes a large portion of the input space by monitoring the execution of the predicate on an initial test suite, guiding the engine and yielding a suite of all non-isomorphic inputs.

Code Coverage Reports: All other techniques in this category instrument elements (lines, branches, etc.) of the program code, execute an initial test case/suite, obtain a coverage report that contains the outcomes of conditional statements, and use automated techniques to generate better test cases. The techniques differ in their goals (*e.g.*, cover a specific program path, satisfy condition-decision coverage, cover a specific statement) and their test case generation algorithms. For example, Miller *et al.* [43] use code coverage and decision outcomes to generate floating-point test data.

Genetic algorithms have also been used to automatically generate test suites that satisfy the *condition-decision* adequacy criterion [41]. Condition-decision criterion requires that each condition in the program be true for at least one test case and false for at least one test case. A fitness function is defined for each branch. An initial test suite is obtained and executed. The fitness functions are used to evaluate the "goodness" of each test case. If a test case covers a new condition-decision, it is considered to be "more fit." The test cases in the gene pool evolve to obtain a new generation of test cases. The process stops when a desired level of fitness is obtained.

Summary: All the above execution feedback-based techniques have been used for a specific type of test case, that is, numerical data values. The feedback (in the form of branch predicate evaluations, condition-decision coverage, and object states) is used to perturb these numerical values in order to improve overall coverage. These techniques are not directly applicable to GUI testing because a GUI test case is a sequence of events. There is no clear notion of perturbing the test case to improve coverage.

Although the techniques discussed in this chapter are not directly applicable to feedback-directed GUI test case generation, many of the underlying concepts are used in this research. For example, execution feedback is used to generate GUI test cases, the EIG model is used to generate the original seed suite, and traversal techniques from model-based testing are used to cover nodes and edges in the annotated GUI model.

Chapter 3

Event Semantic Interaction Relationship

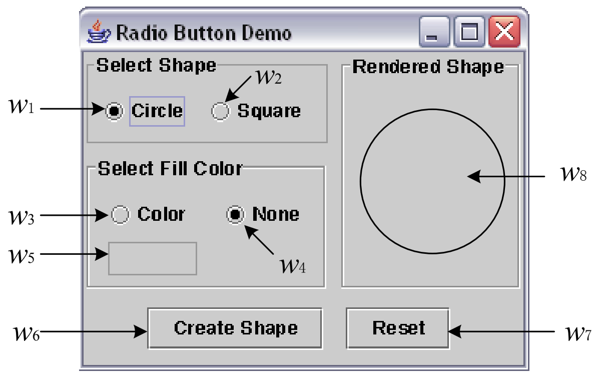
The cornerstone of this research is the *event semantic interaction* (ESI) relationship that is obtained from feedback of test case execution. This chapter lays the foundation for the ESI relationship by formally defining it. It first presents preliminary GUI concepts needed to understand the ESI relationship.

3.1 GUI Preliminaries

In this research, a GUI is defined as a set W of *widgets* (*e.g.*, buttons, text-fields); each widget $w \in W$ has a set P_w of *properties* (*e.g.*, color, size, font). At any time instant, each property $p \in P_w$ has a unique *value* (*e.g.*, red, bold, 16pt); each value is evaluated using a function from the set of the widget's properties to the set of values V_p. With this GUI definition, the GUI's run-time state is defined as follows:

Definition: The GUI *run-time state* S at any time instant is a set of triples (w, p, v), where $w \in W, p \in P_w$ and $v \in V_p$. $\square$

Figure 3.1 shows the partial GUI run-time state of the main window of a simple GUI application "`Radio Button Demo`." The figure shows several widgets, their properties and values. The set of properties for each widget may be different as may the set of values for each property. A special set of GUI run-time states S_I



Widgets: {RadioButton1, TextField1, Button1, Panel, …}
Properties: {Content, Selected, Caption, Enabled, Visible, Color, Weight, Height, …}

(a)

State = {
(RadioButton1, Enabled, *TRUE*),
(RadioButton1, Selected, *TRUE*),
...
(Button1, Caption, "*Create Shape*"),
(Button1, Visible, *TRUE*),
(Button1, Enable, *TRUE*),
...
(TextField1, Content, *EMPTY*),
(TextField1, Weight, *0.55*),
(TextField1, Height, *0.2*),
…
(Panel1, Caption, "*Rendered Shape*"),
(Panel1, Color, "*#cccccc*"),
...
}

(b)

Figure 3.1: (a) `Radio Button Demo` GUI, (b) its Partial State

is called the *valid initial state set* for a particular GUI if the GUI may be in any state $S_i \in S_I$ when it is first invoked.

The GUI run-time state is not static; events e_1, e_2, …, e_n performed on the GUI change its run-time state and hence are modeled as functions that transform one state of the GUI to another. The function notation $S_j = e_x(S_i)$ denotes that S_j is the state resulting from the execution of event e_x in state S_i. If state $S_0 \in S_I$ is the initial state of the GUI, then $e_1(S_0)$ is the GUI run-time state after performing e_1, $e_2(S_0)$ is the GUI run-time state after performing e_2, and $e_2(e_1(S_0))$ is the GUI run-time state after performing the event sequence $< e_1; e_2 >$.

GUIs contain two types of windows: (1) *modal windows*[1] (*e.g.*, `FileOpen`, `Print`) that, once invoked, monopolize the GUI interaction, restricting the focus

[1]Standard GUI terminology, *e.g.*, see http://java.sun.com/products/jlf/ed2/book/HIG.Dialogs.html.

of the user to the range of events within the window until explicitly terminated (*e.g.*, using `Ok`, `Cancel`), and (2) *modeless windows* (*e.g.*, `Find/Replace`) that do not restrict the user's focus. If, during an execution of the GUI, modal window $\mathcal{M}_x$ is used to open another modal window $\mathcal{M}_y$, then $\mathcal{M}_x$ is called the *parent* of $\mathcal{M}_y$ for that execution.

A GUI contains several types of events. *Termination* events close modal windows. Other *structural* events are used to open and close menus, modeless windows and modal windows. The remaining events, called *system-interaction* events, do not manipulate the structure of the GUI.

3.2 ESI Relationships

The main idea behind the event semantic interaction relationship is that events influence one another's execution. Because each event is executed using its corresponding event handler, one could hypothesize that all events whose event handlers interact in terms of code elements (*e.g.*, share variables, exchange messages, share data) should be tested together. For example, consider the event handlers for the events e_2 and e_6 shown in Figure 3.2. The events corresponds to widget w_2 and w_6 in the "`Radio Button Demo`" GUI. As these event handlers interact via the variable `currentShape`, the events e_2 and e_6 should be tested together. However, because the handlers for e_2 and e_3 do not interact, these events need not be tested together.

A variety of static program-analysis techniques may be employed to identify such interactions [48]; they can certainly be used successfully in this example.

e_2:: *click radio button* `Square`

```
public void SquareAction (java.awt.event.ActionEvent evt) {
    currentShape = SHAPE_SQUARE;
    if (created) {
        imagePanel.setShape(currentShape);
        imagePanel.repaint();
    }
}
```

e_3:: *click radio button* `Color`

```
public void ColorAction(java.awt.event.MouseEvent evt) {
    colorText.setEditable(true);
    currentColor = getColor();
    if (created) {
        imagePanel.setFillColor(currentColor);
        imagePanel.repaint();
    }
}
```

e_6:: *click button* `Create Shape`

```
public void CreateAction(java.awt.event.ActionEvent evt) {
    if (color.isSelected()) {
        currentColor = getColor();
    }
    imagePanel.setFillColor(currentColor);
    imagePanel.setShape(currentShape);
    imagePanel.repaint();
    created = true;
}
```

Figure 3.2: Example Event Handlers

However, the limitations of static analysis in the presence of multi-language GUI implementations, callbacks for event handlers, virtual function calls, reflection, and multi-threading are well known [48]. Also, because most GUI applications employ a large number of library elements (*e.g.*, Java Swing), source code may not be available for parts of the GUI.

This research avoids static analysis; instead it approximates the identification of interactions between event handlers by analyzing feedback from the run-time state of the GUI on an initial test suite. As discussed in Section 1.5, it is quite practical to generate a test suite (containing short test cases) from the structural model; this suite is a good candidate to use as a starting point to collect the feedback. The remaining question is: *what dynamic GUI run-time behavior constitutes event interaction?*

Informally, event e_x *interacts with* event e_y, if, when executed together in a sequence $< e_x; e_y >$, they produce a GUI run-time state that is, in some sense, *different from* the two states that would be obtained had e_x and e_y been executed in isolation. Consider the example shown in Figure 3.3. The top-left shows the *initial state* (S_0) of the "`Radio Button Demo`" application. After an event e_2 (event handler shown in Figure 3.2) is executed, the GUI changes its state to the one shown in the top-right ($e_2(S_0)$). In this state, the `Square` radio button is selected. Starting from S_0, one can execute another event (e_6) and obtain the state shown in the bottom-left ($e_6(S_0)$); a circle is created by clicking the `Create Shape` button. If, however, the sequence $< e_2; e_6 >$ is executed in S_0, a new state ($e_6(e_2(S_0))$), shown in the bottom-right is obtained; a square has been created. This execution is equivalent to

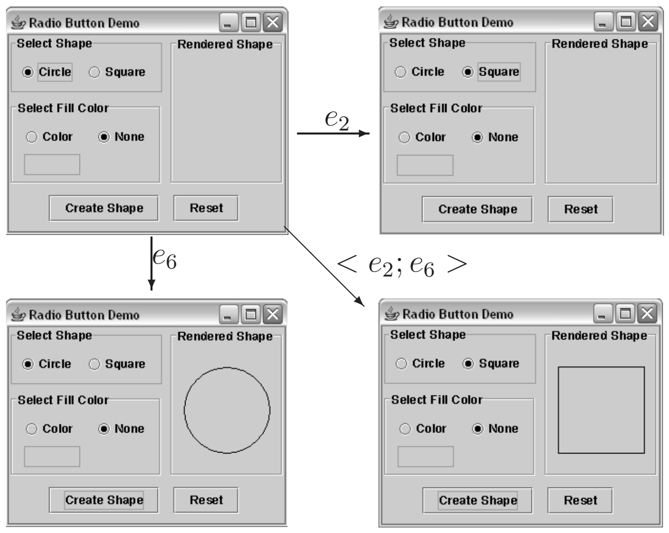


Figure 3.3: Execution of Events e_2 and e_6

the execution of event e_6 in the state $e_2(S_0)$. According to the intuition presented in the beginning of this paragraph, because the sequence $< e_2; e_6 >$ produces a GUI state that is different from the two states that would be obtained had e_2 and e_6 been executed in isolation, event e_2 interacts with event e_6, and should be tested together to check for interaction problems. The event handlers for e_2 and e_6 also show this. They share the variables `created` and `currentShape`; e_6 sets `created` to `TRUE` and influences e_2's flow of control; e_2 sets `currentShape` to a square, which e_6 uses as a parameter to `setShape()`; hence it's not surprising that they interact. As for e_2 and e_3, although they also share the variable `created`, they both only use it without modifying the variable. If the method calls do not modify it as a side effect, there is no information transition between the two events; they need not be tested together.

The usage of "different from" above is somewhat misleading. It seems to suggest that checking state non-equivalence would be sufficient to identify interacting events, *i.e.*, by using a predicate $\mathcal{P}$ such as $(e_2(S_0) \neq e_6(e_2(S_0))) \vee (e_6(S_0) \neq e_6(e_2(S_0)))$. However, this is not the case. Consider an example of two non-interacting events, e_x and e_y, which toggle the states of two independent check-box widgets $\square_x$ and $\square_y$, respectively. Starting in a state $S_0 = \{\square_x, \square_y\}$, *i.e.*, both boxes unchecked, each event would "check" its corresponding checkbox, *i.e.*, $e_x(S_0) = \{\boxtimes_x, \square_y\}$, $e_y(S_0) = \{\square_x, \boxtimes_y\}$, and $e_y(e_x(S_0)) = \{\boxtimes_x, \boxtimes_y\}$. Even though $\mathcal{P}$ would evaluate to `TRUE` for this example, events e_x and e_y are non-interacting and need not be tested together. In order to avoid this confusion, the notion of interacting events needs to be formalized.

3.3 Formalizing the ESI Relationships

It turns out that the example illustrated in Figure 3.3 is just one case of how the GUI state may be used to pinpoint interactions between event handlers – there are many more. This research provides a starting point by identifying a total of twelve cases. They are presented because they were encountered numerous times in previous work on GUI testing. These cases are not exhaustive and new cases may be added, as needed, in the future. The twelve cases will describe (as evaluative predicates) situations in which events e_1 and e_2 interact, *i.e.*, the combined effect of e_1 and e_2 is *different from* the effect of the individual events e_1 and e_2. In these cases, if e_1 and e_2 are system-interaction events in modeless windows; this situation will be called *Context 1*.

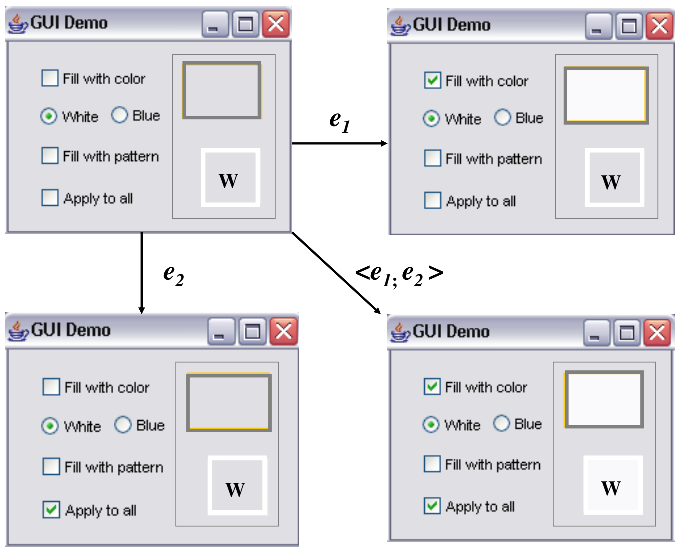


Figure 3.4: **Case 1**: e_1: *Check* `Fill with color`; e_2: *Check* `Apply to all`

Case 1: $\exists w \in W, p \in P_w, v \in V_p, v' \in V_p, s.t.$[2] $((v \neq v') \wedge ((w, p, v) \in \{S_0 \cap e_1(S_0) \cap e_2(S_0)\}) \wedge ((w, p, v') \in e_2(e_1(S_0))))$; 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 $< e_1; e_2 >$ is executed, *i.e.*, the value of w's property p changes from v to v'.

Figure 3.4 gives an example of **Case 1**. This is a "`GUI Demo`" application with several widgets. The `Fill with color` checkbox fills the currently selected shape (highlighted with a deep grey border) with the chosen color determined by the radio buttons `White` and `Blue`. Checkbox `Fill with pattern` determines whether to fill the selected shape with a pattern. Checking `Apply to all` sets all shapes in the right panel with the same color and pattern.

For the purpose of **Case 1**, e_1 is *Check* `Fill with color` and e_2 is *Check*

[2]Notation for "such that."

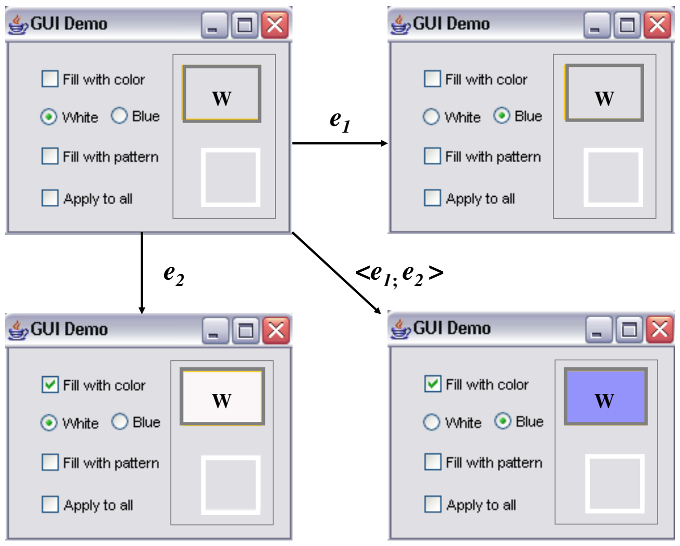


Figure 3.5: **Case 2**: e_1: *Click radio button* `Blue`; e_2: *Check* `Fill with color`

`Apply to all`. The initial state has the rectangle widget selected and color is set to white. The square widget (marked with **W**) is not modified by e_1 or e_2 individually; however, the event sequence $< e_1; e_2 >$ fills the square with the white color. Hence **Case 1** is applicable here and e_1 is ESI related to e_2 because e_1 influences e_2 and their combination modifies the previously unmodified widget **W**.

Case 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 $< e_1; e_2 >$.

An example of **Case 2** using the "`GUI Demo`" application is given in Figure 3.5, where e_1 now represents *Click radio button* `Blue` and e_2 is *Check* `Fill with color`. The initial state has the rectangle selected and color is set to white. Individually, in this initial state, event e_1 sets the current color to blue; event e_2 fills the rectangle

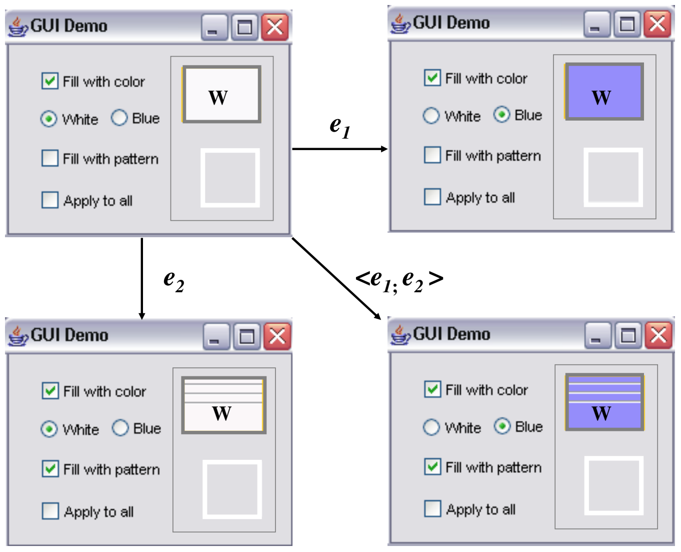


Figure 3.6: **Case 3**: e_1: *Click radio button* `Blue`; e_2: *Check* `Fill with pattern`

with the white color. However, executing $< e_1; e_2 >$ now fills the rectangle with the color blue. **Case 2** applies here as e_1 influences e_2 execution; the widget (marked with **W**) is not modified by e_1; it is modified by e_2; however, it is modified differently by the sequence $< e_1; e_2 >$.

A variation of **Case 2** is called **Case 2.1**, in which the roles of e_1 and e_2 are exchanged with the combined sequence $< e_1; e_2 >$ remaining the same.

Case 3: $\exists w \in W, p \in P_w, v \in V_p, v' \in V_p, v'' \in V_p, \bar{v} \in V_p, s.t.\ ((v \neq v') \wedge (v \neq v'') \wedge (v'' \neq \bar{v}) \wedge ((w, p, v) \in S_0) \wedge ((w, p, v') \in e_1(S_0)) \wedge ((w, p, v'') \in e_2(S_0)) \wedge ((w, p, \bar{v}) \in e_2(e_1(S_0))))$; 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 $< e_1; e_2 >$.

Figure 3.6 shows one example of this case using the "`GUI Demo`" application. In this example, the initial state has `Fill with color` checked, white is set to be the current color and the rectangle is selected. Event e_1 here is *Click radio button*

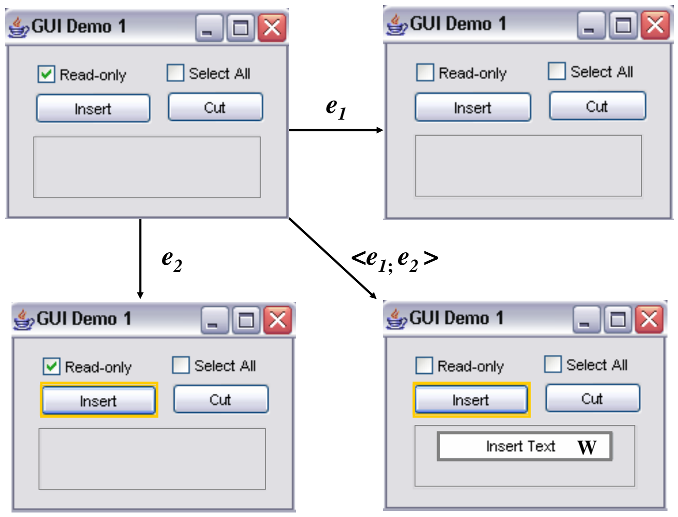


Figure 3.7: **Case 4**: e_1: *Uncheck* `Read-only`; e_2: *Click button* `Insert`

`Blue` and e_2 is *Check* `Fill with pattern` that fills the current shape with a pattern. Events e_1 and e_2 modify the rectangle individually; however, executing $< e_1; e_2 >$ now modifies the rectangle differently. Therefore, e_1 influences e_2, *i.e.*, resulting different modification of the existing widget (marked with **W**), and **Case 3** applies.

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

Case 4: $\exists w \in W, \exists p \in P_w, \exists v \in V_p, \forall \bar{p} \in P_w, \forall \bar{v} \in V_p, s.t.\ (((w, \bar{p}, \bar{v}) \notin S_0) \wedge ((w, \bar{p}, \bar{v}) \notin e_1(S_0)) \wedge ((w, \bar{p}, \bar{v}) \notin e_2(S_0)) \wedge ((w, p, v) \in e_2(e_1(S_0))))$; there is at least one *new* widget w with property p and value v in state $e_2(e_1(S_0))$, *i.e.*, it is created by event sequence $< e_1; e_2 >$; but it does not exist in state S_0 and could not be created by either e_1 or e_2 individually, *i.e.*, no triple involving widget w exists in

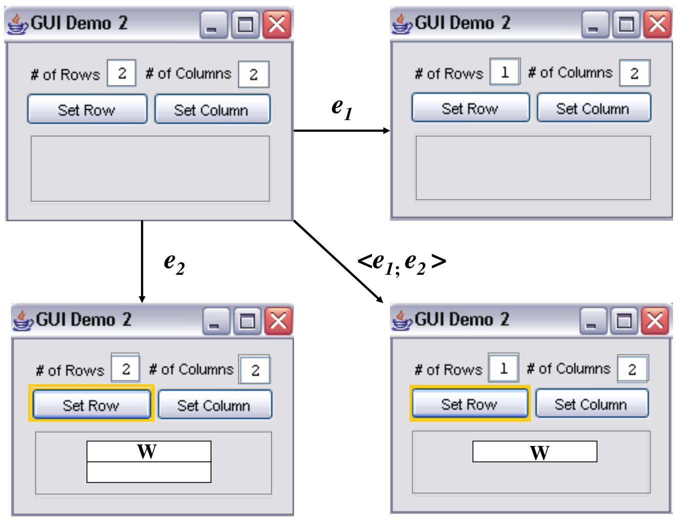


Figure 3.8: **Case 5.1**: e_1: *Input row number*; e_2: *Click button* `Set Row`

any of the states S_0, $e_1(S_0)$ and $e_2(S_0)$.

The example using "`GUI Demo 1`" for this case is shown in Figure 3.7. In this application, checking `Read-only` forbids inserting text into the bottom panel; checking `Select All` selects all the widgets in the bottom panel. Clicking button `Insert` creates a text-field for inputing text, and clicking button `Cut` removes the current selection (either text-field in the panel or text in text-field). To illustrate **Case 4**, assume that the initial state has `Read-only` checked and an empty bottom panel. Event e_1 unchecks `Read-only` and e_2 clicks the button `Insert`. It is clear that e_2 cannot insert the text-field into the bottom panel with `Read-only` checked. However, when executing $< e_1; e_2 >$, e_1 first removes the read-only restriction to the panel, and then e_2 creates a text-field. Hence, e_1 influences e_2 by making it create a new widget (marked with **W**) previously non-existent in the initial state; **Case 4** is applicable here.

Case 5: $\exists w \in W, \exists p \in P_w, \exists v \in V_p, \exists v' \in V_p, \forall \bar{p} \in P_w, \forall \bar{v} \in V_p, s.t.\ ((v \neq v') \wedge$

$((w, \bar{p}, \bar{v}) \notin S_0) \wedge ((w, p, v) \in e_1(S_0)) \wedge ((w, \bar{p}, \bar{v}) \notin e_2(S_0)) \wedge ((w, p, v') \in e_2(e_1(S_0))))$; there is at least one widget w that does not exist in the initial state S_0; it is created by e_1 with property p and value v; e_2 does not create w. However, w is created differently when the sequence $< e_1; e_2 >$ is executed, *i.e.*, the value of w's property p is now v' (not v).

A variation of **Case 5** is called **Case 5.1** in which the roles of e_1 and e_2 are exchanged and the combined sequence $< e_1; e_2 >$ remains the same. Figure 3.8 shows an example for **Case 5.1**. The "`GUI Demo 2`" application is used in this example. It is used to create a table with a given number of rows and columns. One can input the desired number of rows and columns in the text-fields labeled as `# of Rows` and `# of Columns`. By clicking either the button `Set Row` or `Set Column`, a table with the specified number of rows and columns is created in the bottom panel; or if a table already exists in the panel, the number of rows and columns are changed to the given numbers.

For **Case 5.1**, the initial state has the row and column number both set to 2 and an empty bottom panel. In this initial state, event e_1 inputs 1 into the text-field to set the number of rows; e_2 clicks the button `Set Row` and creates a new table widget with two rows. However, when executing $< e_1; e_2 >$, a table with only one row is created. Therefore, e_1 influences e_2 and modifies its creation of the new widget, *i.e.*, the table (marked with **W**); **Case 5.1** is applicable here.

Case 6: $\exists w \in W, \exists p \in P_w, \exists v \in V_p, \exists v' \in V_p, v'' \in V_p, \forall \bar{p} \in P_w, \forall \bar{v} \in V_p, s.t. ((v' \neq v'') \wedge ((w, \bar{p}, \bar{v}) \notin S_0) \wedge ((w, p, v) \in e_1(S_0)) \wedge ((w, p, v') \in e_2(S_0)) \wedge ((w, p, v'') \in e_2(e_1(S_0))))$; there is at least one *new* widget w that does not exist in state S_0;

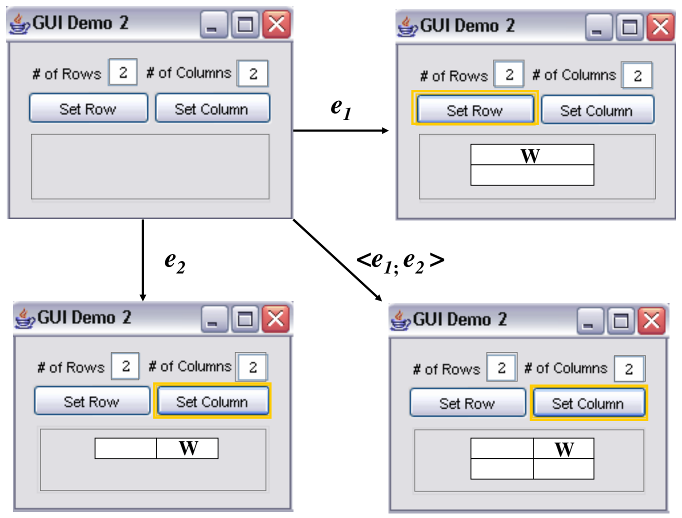


Figure 3.9: **Case 6**: e_1: *Click button* `Set Row`; e_2: *Click button* `Set Column`

but w is created by e_1 and e_2 individually. However, it is created by the sequence $< e_1; e_2 >$ with a different value v'' for property p.

This case is also demonstrated using the "`GUI Demo 2`" application in Figure 3.9. In this example, the initial state has row and column number set to 2 and an empty bottom panel. Event e_1 clicks the button `Set Row` and e_2 clicks the button `Set Column`. Event e_1 individually creates a new table with one column and two rows; event e_2 creates a one-row and two-column table. Executing $< e_1; e_2 >$ creates a table with two rows and two columns. Hence, e_1 influence e_2, *i.e.*, resulting in different creation of new widget (marked with **W**), and **Case 6** is applicable here.

Event execution may also "remove" existing widgets from a GUI, *e.g.*, by cutting selected components. The next three cases handle removed widgets.

Case 7: $\exists w \in W, \exists p \in P_w, \exists v \in V_p, \exists v' \in V_p, v'' \in V_p, \forall \bar{p} \in P_w, \forall \bar{v} \in V_p, s.t. (((w, p, v) \in S_0) \wedge ((w, p, v') \in e_1(S_0)) \wedge ((w, p, v'') \in e_2(S_0)) \wedge ((w, \bar{p}, \bar{v}) \notin e_2(e_1(S_0))))$; there is at

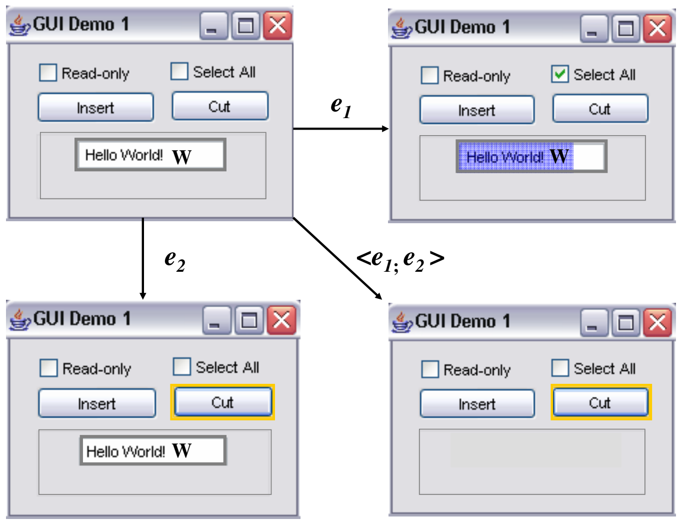


Figure 3.10: **Case 7**: e_1: *Check* `Select All`; e_2: *Click button* `Cut`

least one widget w that exists in the initial state S_0 with property p and value v; it is not removed by e_1 and e_2 individually. However, it is removed when the sequence $< e_1; e_2 >$ is executed.

Case 7 is illustrated via an example using "`GUI Demo 1`" application in Figure 3.10. In this example, the initial state has all checkboxes unchecked and a text-field with text `Hello World` in the bottom panel. Event e_1 checks `Select All` and selects the text in the text-field; e_2 clicks the button `Cut`. They individually do not remove the text-field in the bottom panel. However, executing $< e_1; e_2 >$ results in an empty bottom panel. Therefore, e_1's selection of widgets influences e_2 and enables it to remove an existing widget (marked with **W**); **Case 7** is applicable here.

Case 8: $\exists w \in W, \exists p \in P_w, \exists v \in V_p, \forall \bar{p} \in P_w, \forall \bar{v} \in V_p, s.t.\ (((w, \bar{p}, \bar{v}) \notin S_0) \wedge ((w, p, v) \in e_1(S_0)) \wedge ((w, \bar{p}, \bar{v}) \notin e_2(S_0)) \wedge ((w, \bar{p}, \bar{v}) \notin e_2(e_1(S_0))))$; there is at least one widget w that does not exist in the initial state S_0; it is created by e_1 with

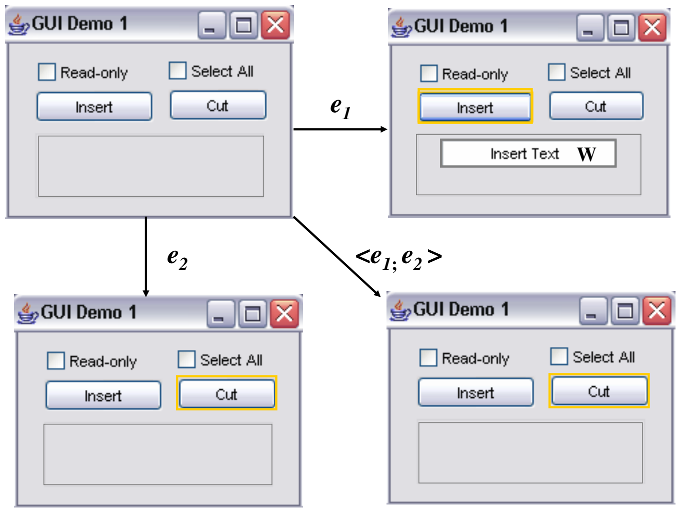


Figure 3.11: **Case 8**: e_1: *Click button* `Insert`; e_2: *Click button* `Cut`

property p and value v; e_2 does not create w individually. However, it is removed when the sequence $< e_1; e_2 >$ is executed.

Case 8 is demonstrated using "`GUI Demo 1`" in Figure 3.11. The initial state in this example has all checkboxes unchecked and an empty bottom panel. Event e_1 is *Click button* `Insert`; e_2 is *Click button* `Cut`. Event e_1 inserts a text-field into the bottom panel; e_2 removes selected items in the panel if there are any. However, executing $< e_1; e_2 >$ first inserts the text-field (selected at the time of creation) by e_1, then e_2 removes the text-field. Therefore, e_1 influences e_2, *i.e.*, e_2 removes widget (marked with **W**) newly created by e_1, and **Case 8** is applicable here.

A variation of **Case 8** is called **Case 8.1** in which the roles of e_1 and e_2 are exchanged and the combined sequence $< e_1; e_2 >$ remains the same. Figure 3.12 shows an example illustrating **Case 8.1**. In this example, the initial state has unchecked checkboxs and an empty panel. Event e_1 is *Check* `Read-only` and e_2 is *Click button* `Insert`. Event e_2 creates a text-field widget in the panel. However,

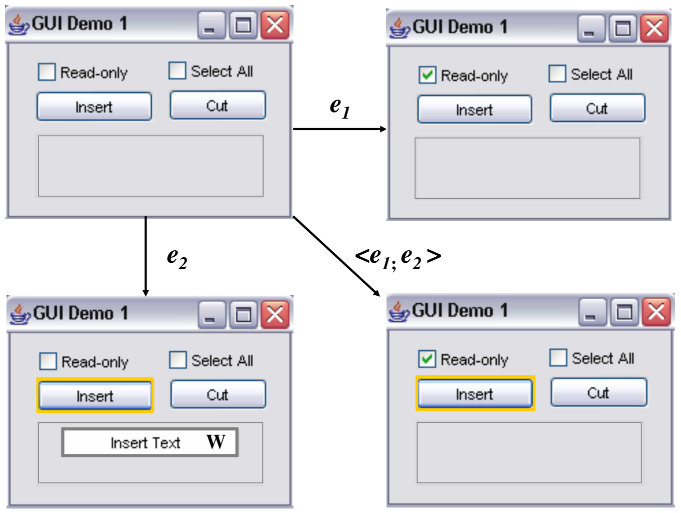


Figure 3.12: **Case 8.1**: e_1: *Check* `Read-only`; e_2: *Click button* `Insert`

executing $< e_1; e_2 >$ does nothing to the panel because e_1 first sets the panel to read-only; e_2 cannot create a new text-field. Hence, e_1 influences e_2's creation of the new widget (marked with **W**; **Case 8.1** is applicable here.

Case 9: $\exists w \in W, \exists p \in P_w, \exists v \in V_p, \exists v' \in V_p, \forall \bar{p} \in P_w, \forall \bar{v} \in V_p, s.t. (((w, \bar{p}, \bar{v}) \notin S_0) \wedge ((w, p, v) \in e_1(S_0)) \wedge ((w, p, v') \in e_2(S_0)) \wedge ((w, \bar{p}, \bar{v}) \notin e_2(e_1(S_0))))$; there is at least one *new* widget w that does not exist in state S_0; but it is created by e_1 with property p and value v, and by e_2 with property p and value v' individually. However, it is removed by the sequence $< e_1; e_2 >$, *i.e.*, no triple involving widget w is in state $e_2(e_1(S_0))$.

The next two cases describe interactions in which existing widgets are removed by individual events, but are re-created by the sequence $< e_1; e_2 >$.

Case 10: $\exists w \in W, \exists p \in P_w, \exists v \in V_p, \exists v' \in V_p, \exists v'' \in V_p, \forall \bar{p} \in P_w, \forall \bar{v} \in V_p, s.t. (((w, p, v) \in S_0) \wedge ((w, \bar{p}, \bar{v}) \notin e_2(S_0)) \wedge ((w, p, v') \in e_1(S_0)) \wedge ((w, p, v'') \in$

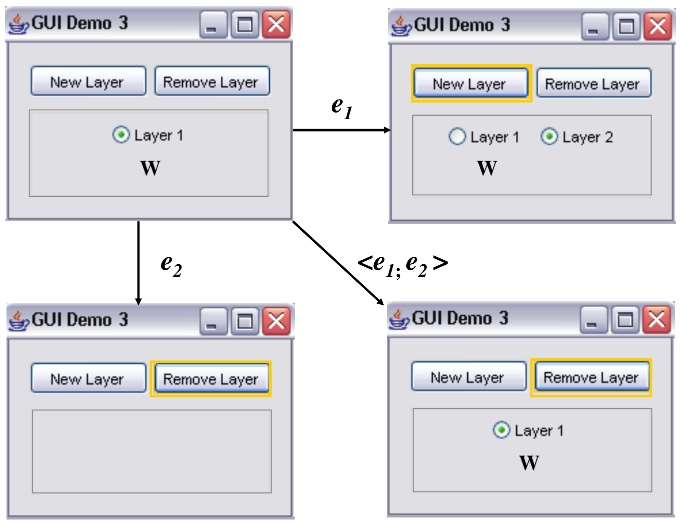


Figure 3.13: **Case 10**: e_1: *Click button* `New Layer`; e_2: *Click button* `Remove Layer`

$e_2(e_1(S_0))))$; there is at least one widget w that exists in the initial state S_0 with property p and value v; it is removed by e_2; it is modified by e_1 with property p and value v'. However, it is re-created when the sequence $< e_1; e_2 >$ is executed, *i.e.*, the triple (w, p, v'') is in state $e_2(e_1(S_0))$.

The example shown in Figure 3.13 demonstrates **Case 10**. The application used here is "`GUI Demo 3`." In this application, clicking button `New Layer` creates a radio button labeled with a layer number in the bottom panel. Clicking button `Remove Layer` removes the radio button labeled with the highest layer number. The example has the initial state with a created `Layer 1` in the panel. Event e_1 clicks button `New Layer` and creates `Layer 2`; e_2 clicks button `Remove Layer` and removes the existing `Layer 1`. However, when executing $< e_1; e_2 >$, e_2 now removes the newly created `Layer 2` instead of the original `Layer 1`. Hence, e_1 influences e_2, *i.e.*, keeping the widget (marked with **W**) that would have been removed. **Case 10** captures this scenario.

A variation of **Case 10** is called **Case 10.1** in which the roles of e_1 and e_2 are exchanged and the combined sequence $< e_1; e_2 >$ remains the same.

Case 11: $\exists w \in W, \exists p \in P_w, \exists v \in V_p, \exists v' \in V_p, \forall \bar{p} \in P_w, \forall \bar{v} \in V_p, s.t.\ (((w, p, v) \in S_0) \wedge ((w, \bar{p}, \bar{v}) \notin e_1(S_0)) \wedge ((w, \bar{p}, \bar{v}) \notin e_2(S_0)) \wedge ((w, p, v') \in e_2(e_1(S_0))))$; there is at least one widget w that exists in the initial state S_0 with property p and value v; it is removed by e_1 and e_2 individually. However, it is re-created when the sequence $< e_1; e_2 >$ is executed, *i.e.*, the triple (w, p, v') is in state $e_2(e_1(S_0))$.

Finally, 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 12: $\exists w \in W, \texttt{ENABLED} \in P_w, \texttt{TRUE} \in V_{\texttt{ENABLED}}, \texttt{FALSE} \in V_{\texttt{ENABLED}}, s.t.\ (((w, \texttt{ENAB-LED}, \texttt{FALSE}) \in S_0) \wedge ((w, \texttt{ENABLED}, \texttt{TRUE}) \in e_1(S_0)) \wedge \texttt{EXEC}(e_2, w))$; 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 $\texttt{EXEC}(e_2, w)$.

Modal windows create special situations for Cases 1 through 12 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, 3* and *4*.

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 $< e_1; \texttt{TERM} >$, $e_2(S_0)$ is the state of the GUI after the execution of the event sequence $< e_2; \texttt{TERM} >$, and $e_2(e_1(S_0))$ is the state of the GUI after the execution of the event sequence $< e_1; e_2; \texttt{TERM} >$. All the predicates defined in Cases 1 through 12 apply, using these modified definitions, for e_1 and e_2 in the same modal window.

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 $< e_1; \texttt{TERM} >$, $e_2(S_0)$ 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 $< e_1; \texttt{TERM}; e_2 >$. All the predicates defined in Cases 1 through 12 apply.

Context 4: If e_1 is associated with a widget contained in a modal window with termination event $\texttt{TERM}_1$, and e_2 is associated with a widget contained in another modal window with termination event $\texttt{TERM}_2$ (*i.e.*, the window that is derived through another path from the main window) then $e_1(S_0)$ is the state of the GUI after the execution of the event sequence $< e_1; \texttt{TERM}_1 >$, $e_2(S_0)$ is the state of the GUI after the execution of the event sequence $< e_2; \texttt{TERM}_2 >$, and $e_2(e_1(S_0))$ is the state of the GUI after the execution of the event sequence $< e_1; \texttt{TERM}_1; \texttt{R}; e_2; \texttt{TERM}_2 >$, where R is the sequence of events needed to open the modal window containing e_2. All the predicates defined in Cases 1 through 12 apply.

There is an *Event Semantic Interaction* (ESI) relationship between two events

e_1 and e_2 if and only if at least one of the predicates in Cases 1 through 12 evaluates to `TRUE`; this relationship is written as $e_1 \xrightarrow{n(m)} e_2$, where the number n is one of the case numbers 1 through 12; m is the context number. If multiple cases apply, then one of the case numbers is used. Due to the specific ordering of the events in the sequence $< e_1; e_2 >$, the ESI relationship is not symmetric.

3.4 Summary

In this chapter, the concept of GUI execution feedback, *i.e.*, GUI run-time information was defined in terms of the GUI run-time state. The ESI relationships were described and formalized using 12 predicates. Because the run-time state is defined as a set, and the usual set operations (set equivalence, intersection, union, etc.) are used in the ESI identification predicates, all cases can be evaluated automatically.

The EIG-based test cases may be used as the basis for the predicate evaluation. With the needed GUI run-time state information collected during test cases execution, the predicates are evaluated for each pair of system-interaction events that are either (1) directly connected by an edge in the EIG (Context 1) or (2) connected by a path that does not contain any intermediate system-interaction events (Context 2, 3 and 4). If one of the predicates evaluates to `TRUE`, the two events are ESI-related.

In the next two chapters, two approaches utilizing the ESI relationships for test case generation are described and studied. The first one is presented in Chapter 4, which uses the ESI relationships obtained from the initial run-time information to generate new test cases. The second approach, presented in Chapter 5, alternates test case generation and execution. New ESI relationships are obtained from previously generated test cases and used to generate new test cases.

Chapter 4

ESIG-Based Test Case Generation

The first feedback-directed test case generation approach follows directly from the earlier EIG-based approach. Instead of generating test cases from the entire EIG, certain EIG edges are annotated with ESI relationships, and test cases are generated only from these annotated edges. The sub-graph containing only the annotated edges and associated nodes is called the *event semantic interaction graph* (ESIG). 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. *Multi-way* interaction test cases are generated from the ESIG model using a graph-traversal algorithm.

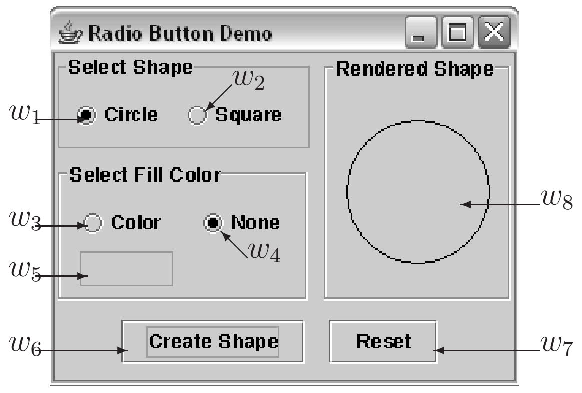


Figure 4.1: A Simple GUI Application

4.1 Overview of the ESIG-Based Test Case Generation Process

The test case generation steps used in this technique are now presented using the simple "`Radio Button Demo`" GUI introduced in Section 3.1 (Figure 4.1). The GUI contains seven widgets labeled w_1 through w_7 on which a user can perform corresponding events e_1 through e_7. The application's functionality is very straightforward – the *initial state* has `Circle` and `None` selected; the text-field corresponding to w_5 is empty; and the `Rendered Shape` area (widget w_8) is empty. Event e_6 creates a shape in the `Rendered Shape` area according to current settings of $w_1 \ldots w_5$; event e_7 resets the entire software to its initial state.

The other events behave as follows. Event e_1 sets the shape to a circle; if there is already a square in the `Rendered Shape` area, then it is immediately changed to a circle. Event e_2 is similar to e_1, except that it changes the shape to a square. Event e_3 enables the text-field w_5, allowing the user to enter a custom fill-color, which is immediately reflected in the shape being displayed (if there is a shape there). Event e_4 reverts back to the initial state.

The GUI of this application is simple, yet quite flexible. The numbers of 1-, 2-, 3-, 4-, and 5-way event sequences (and hence possible test cases) that may be executed in the initial state of the GUI are 6 (remember that e_5 is initially disabled), 37, 230, 1491, and 9641, respectively. This is clearly too large a number to test on such a small GUI.

The ESIG-based test case generation process has the following steps:

(1) *Obtain the EIG.* As mentioned in Section 2.2, this is done via automated reverse

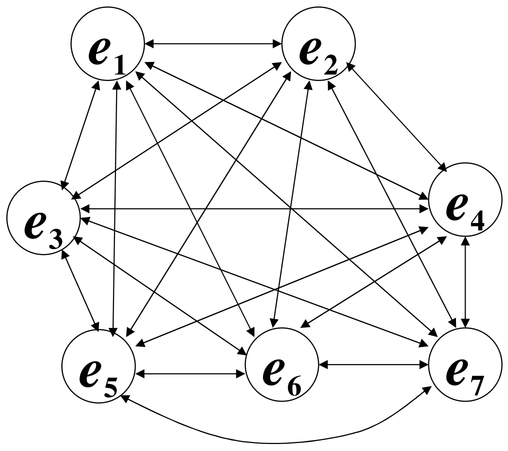


Figure 4.2: EIG of "`Radio Button Demo`" GUI

engineering techniques [36]. Because of current limitations of the reverse engineering process, it is unable to automatically infer the (enable) relationship between e_3 and e_5; hence the EIG is a fully connected directed graph with seven nodes, corresponding to the seven events. Figure 4.2 shows the obtained EIG for this simple GUI.

(2) *Generate and execute the 2-way covering test suite.* This suite consists of all 2-way covering test cases, which are obtained by simply enumerating each pair of system-interaction events that are either (1) directly connected by an edge in the EIG (Context 1) or (2) connected by a path that does not contain any intermediate system-interaction events (Context 2, 3 and 4). Each of these sequences is executed in the software's initial state. As expected, none of the sequences starting with e_5 executed. However, the sequence $< e_3; e_5 >$ executed successfully, indicating that e_3 enables e_5.

Also, the entire state of the GUI is captured after each event for each test case. This includes all the properties of all the GUI's widgets. Here, the discussion is restricted to the state of interest for this example, which includes the state of each radio button, *i.e.*, selected/not-selected and the contents of `Rendered Shape` area. This part of the state is used to compute the ESI relationships.

(3) *Compute ESI relationships.* The ESI relationship between two events is based on the ability of an event to influence another event's execution, as captured in the GUI's state. As described in Section 3.2, e_2 influences e_6. Event e_6 alone from the initial state renders a circle in the `Rendered Shape` area. However, executing e_2 before e_6 changes the behavior of e_6, yielding a square instead. This "interaction" is captured by the ESI predicate **Case 5.1** (described in Section 3.3) and represented as $e_2 \xrightarrow{5.1(1)} e_6$.

Another interesting relation in this example is $e_6 \xrightarrow{5(1)} e_2$. In the default initial state, e_6 creates a circle. However, the sequence $< e_6; e_2 >$ yields a square because e_2 changes the shape. The predicate in the set used to compute this relation is **Case 5** described in Section 3.3. This interaction is due to the variables `created` and `currentShape` shared between the code of e_6 and e_2.

Another ESI relationship is obtained from **Case 12** (described in Section 3.3) that addresses the "enable relationship." This predicate applies because widget w_5 is disabled in the initial state but enabled by e_3.

In summary, the three relations found in this step are: $e_2 \xrightarrow{5.1(1)} e_6$, $e_6 \xrightarrow{5(1)} e_2$, and $e_3 \xrightarrow{12(1)} e_5$.

(4) *Create ESIG and generate multi-way test cases.* The three ESI relationships obtained from the previous step are used to annotate the EIG, which yields the ESIG shown in Figure 4.3. Generating all possible test cases of length 3 returns two

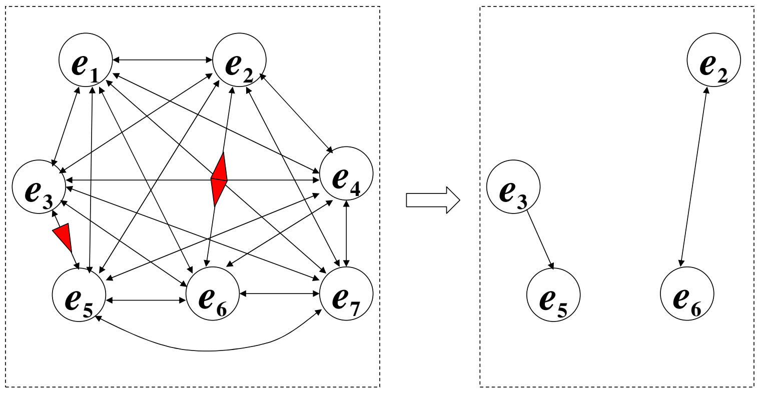


Figure 4.3: Annotated EIG and ESIG for "`Radio Button Demo`" GUI

test cases $< e_2; e_6; e_2 >$ and $< e_6; e_2; e_6 >$.

This simple example demonstrates the ESIG-based test case generation process. All the tools needed for this process were implemented in a GUI testing framework called **GUITAR**. The test case generation algorithm, called `GenTestCases`, was also implemented. In summary, this graph-based test case generation takes a directed graph as input and outputs all paths of a specified length. For example, all paths of length 2 contain 2 events; they are simply the edges. The paths of length 3 are obtained by outputing all pairs of adjacent edges. The effectiveness of the ESIG-based test cases is evaluated next.

4.2 Study 1 of ESIG-Based Approach: Evaluating the ESIG-Based Approach on Fielded Applications

Study 1 was conducted to evaluate the fault detection of ESIG-based test case generation. The *test oracle*, *i.e.*, a mechanism that determines whether a GUI executed correctly for a test case, used in this study is that a GUI is considered to have *passed* a test case if it did not "crash" (terminate unexpectedly or throw an uncaught exception) during the test case's execution; otherwise it *failed*. Such crashes may be detected automatically by the script used to execute the test cases. The EIG and ESIG, and their respective test cases may also be obtained automatically. Hence, the entire end-to-end feedback-directed GUI testing process for "crash testing" could be executed without human intervention. Note that, in the Study 2 discussed in next session, the work is extended by employing a more powerful test oracle to detect additional failures.

A *Run-Time State Analyzer* and an *Annotator* have been implemented and integrated into the **GUITAR** testing framework. Implementation of the crash testing process included setting up a database for text-field values. Because the overall process needed to be fully automatic, a database containing one instance for each of the text types in the set {*negative number, real number, long file name, empty string, special characters, zero, existing file name, non-existent file name*} was used. Note that if a text-field is encountered in the GUI, one instance for each text type is tried in succession by `GenTestCases`.

Finally, because nodes in the EIG and ESIG do not represent events to open or close menus, or open windows, the sequences obtained from these models may not be executable. At execution time, other events needed to reach the events are automatically generated, yielding an executable test case [40]. To allow a clean application exit, a test case is also automatically augmented by `GenTestCases` with associated termination events that close all open modal windows before the test case terminates.

4.2.1 Research Questions

The crash testing process provided a starting point to evaluate the ESIG-generated test cases. The following questions needed to be answered to determine the usefulness of the overall feedback-directed process:

C4Q1: How many test cases are required to test 2-way interactions in an EIG? How does this number grow for 3-, 4-, ..., n-way interactions?

C4Q2: In how many ESI relationships does a given event participate? How many test cases are required to test 2-way interactions in an ESIG? How does this number grow for 3-, 4-, ..., n-way interactions?

C4Q3: How do the ESIG- and EIG-generated test suites compare in terms of fault-detection effectiveness? Do the former detect faults that were not detected by the latter?

4.2.2 Process and Results

To answer the above questions while minimizing threats to external validity, this study was conducted using four extremely popular GUI-based open-source software (OSS) applications downloaded from SourceForge. The fully-automatic crash testing process was executed on them and the cause (*i.e.*, the *fault*) of each crash in the source code was determined. The process of the study is as follows:

STEP 1: Selection of subject applications. Four popular GUI-based OSS

(FreeMind 0.8.0, GanttProject 2.0.1, jEdit 4.2, OmegaT 1.7.3) were downloaded from SourceForge. These applications have been used in previous experiments [55].

1. **FreeMind**[1], which is a mind-mapping[2] software written in Java. It has an all-time activity of 99.72%.

2. **GanttProject**[3], which is a project scheduling application written in Java and featuring Gantt chart, resource management, calendaring, import/export (MS Project, HTML, PDF, spreadsheets). It has an all-time activity of 98.12%.

3. **jEdit**[4], which is a programmer's text editor written in Java. It uses the Swing toolkit for the GUI and can be configured as a powerful IDE. When tested, it had an all-time activity of 99.95%.

4. **OmegaT**[5], which is a multi-platform Computer Assisted Translation tool with fuzzy matching, translation memory, keyword search and glossaries. It has an all-time activity of 99.80%.

The characteristics of these OSS are showed in Table 4.1.

Subjects	**Windows**	**Widgets**	**LOC**	**Classes**	**Methods**
FreeMind	30	611	13,463	765	3114
GanttProject	18	326	22,711	840	5189
jEdit	27	498	48,444	829	5582
OmegaT	18	228	22,708	274	1522
TOTAL	93	1663	107,326	2798	15,407

Table 4.1: Subject Applications for Study 1

All of the above applications were chosen due to their popularity, active com-

[1]http://sourceforge.net/projects/freemind

[2]http://en.wikipedia.org/wiki/Mind_map

[3]http://sourceforge.net/projects/ganttproject

[4]http://sourceforge.net/projects/jedit

[5]http://sourceforge.net/projects/omegaT

munity of developers, and high all-time activity. Due to their popularity, these applications have undergone quality assurance before release. To further eliminate "obvious" bugs, a static analysis tool called *FindBugs* [26] was executed on all the applications; after the study, it was verified that none of the reported bugs were detected by FindBugs.

STEP 2: Generation of EIGs & seed test suites. The EIGs of all subject applications were obtained using reverse engineering. To address **C4Q1** above, the number of test cases required to test 2-, 3-, 4-, and 5-way interactions was computed. The result for each application is shown as a solid line in Figure 4.4 (the y-axis in all these plots is a logarithmic scale). The plot shows that the number of test cases grows exponentially with the number of interactions. The number quickly becomes unmanageable for more than 2- and 3-way interactions. In this study, only 2-way interactions were tested by the seed test suites. The seed test suites contained 309,136, 84,681, 204,304 and 38,809 test cases for FreeMind, GanttProject, jEdit and OmegaT, respectively.

STEP 3: Execution of the seed test suite. The entire seed suite executed without any human intervention. 50 machines were used to run the test cases in parellel, each running Linux at 2GHz with 1GHz of RAM. The seed test suite executed in 1293.54, 294.68, 662.53 and 143.39 hours on FreeMind, GanttProject, jEdit and OmegaT, respectively. In all, 40,509, 1,330, 10,062, and 2,775 test cases caused crashes; these crashes were caused by 7, 8, 7 and 4 *faults* (as defined earlier) for FreeMind, GanttProject, jEdit and OmegaT, respectively. The GUI's run-time state was recorded during test execution. All faults were fixed in the applications

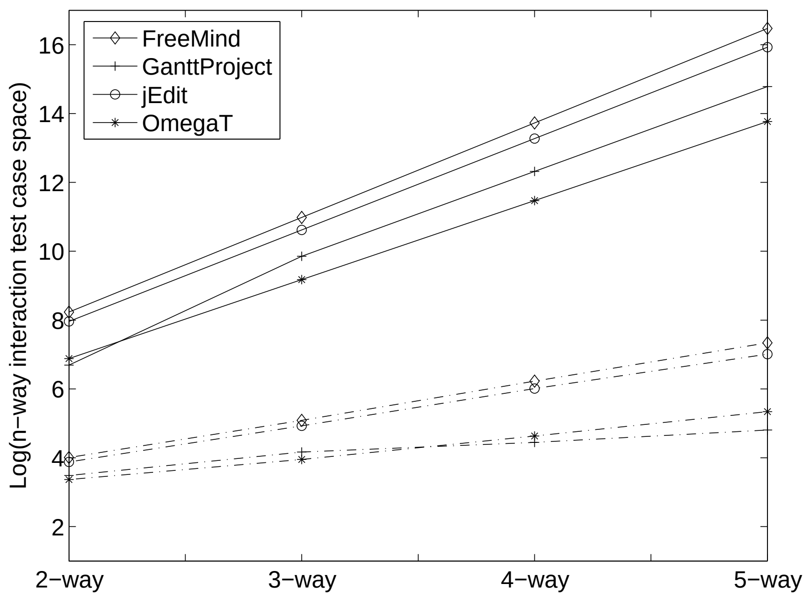


Figure 4.4: Test Case Space Growth

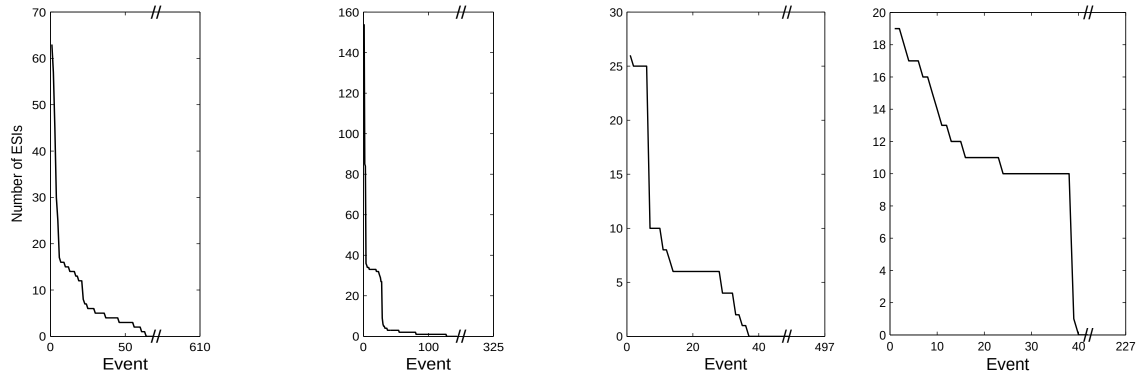
to avoid the masking of other faults that need longer test cases to detect.

STEP 4: Generation of the ESIG. The above feedback was used to obtain the ESIs for each application. To address **C4Q2**, the total number ESI relationships found is summarized in Table 4.2, and the number of ESI relationships in which each event participates is shown in Figure 4.5. Each event in the GUI has been assigned a unique integer ID; all event IDs are shown on the x-axis. The y-axis shows the number of ESI relationships in which each event participates.

The result shows that certain events (around 25%) dominate the ESI relationship in GUIs. Manual examination of these "dominant" events revealed that the nature of the subject applications – most have a single dominant object (mind map, project schedule, editor panel, translation window) that is the focus of most events – is such that several key events influence a large number of other events. In future

Subject Application	2-way Suites
FreeMind	614
GanttProject	710
jEdit	591
OmegaT	469

Table 4.2: ESI relationships



(a) FreeMind-0.8.0 (b) GanttProject-2.0.1 (c) jEdit-4.2 (d) OmegaT-1.7.3

Figure 4.5: ESI Distribution in OSS

work, a classification of these dominant events may be created. Moreover, several events participate in very few or no ESI relations. These events include parts of the `Help` menu that have no interaction with other application events, and windowing events such as scrolling for which no developer-written code exists.

The ESIs were used to obtain the ESIGs and, subsequently, additional test cases. The number of test cases required to test 2-, 3-, 4-, and 5-way interactions using an ESIG is shown, for each application, as a dotted line in Figure 4.4. This result shows that the growth of the ESIG-generated test cases appears manageable for 3-, 4-, or even (given sufficient resources) 5-way interactions. For example, for 3-way interaction, it is in fact reduced from the EIG by 99.99%. In this study, for FreeMind, 3-way test cases were generated from its ESIG; jEdit had 3- and 4-way

ESIG-test cases; GanttProject and OmegaT, due to the relatively small scale of their ESIGs, had all 3-, 4- and 5-way ESIG test cases generated. Table 4.3 shows the total number of test cases for these interactions (a "-" indicates that there is no such entry).

Subjects	3-way	4-way	5-way	Total
FreeMind	10,208	-	-	10,208
GanttProject	3070	14,742	27,933	45,745
jEdit	7572	84,488	-	92,060
OmegaT	2335	8935	42,859	54,129

Table 4.3: Multi-way Interaction Test Cases of ESIG

STEP 5: Execution of the test cases. To address **C4Q3**, all the newly-generated test cases were executed. The execution took 58.784, 240.368, 400.915 and 171.81 hours on FreeMind, GanttProject, jEdit and OmegaT, respectively, and lasted for several days on the 50-machine cluster. In all, 156, 115, 15, and 0 test cases caused crashes; they were caused by 2, 3, 2 and 0 faults for FreeMind, GanttProject, jEdit and OmegaT, respectively. These faults had not been detected by the 2-way test cases. The result summarized in Figure 4.6 shows that the ESIG-based test cases help to detect additional faults.

4.2.3 Discussion

This study demonstrated that test suites for multi-way GUI event interactions are able to detect additional faults compared to 2-way interactions. In earlier work, it has been shown that 2-way interactions yield high code coverage, while multi-way interactions cover little additional code [37]. The additional fault-detection

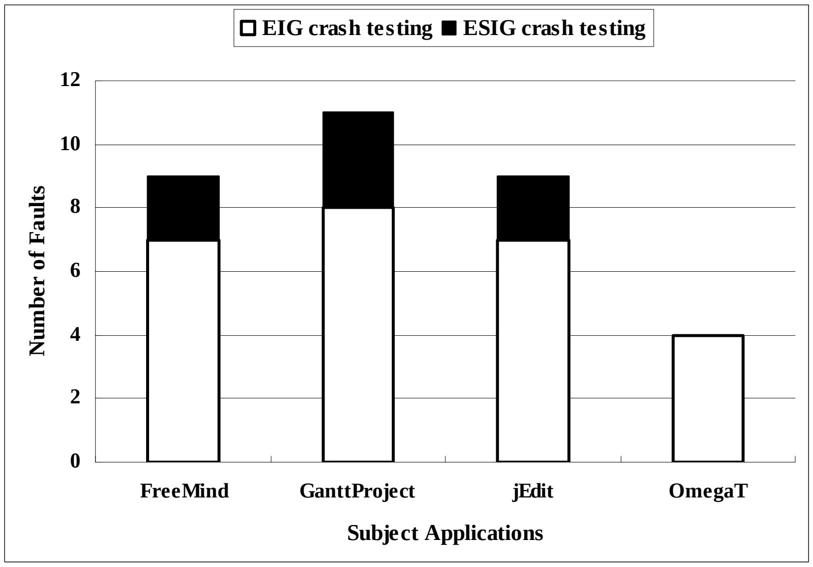


Figure 4.6: Fault-Detection Effectiveness

effectiveness of multi-way interactions is due to the execution of combinations of events in different execution orders. Also, this study did not use the newly-generated test cases in its seed suite to generate additional test cases; the extension is explored and its benefits demonstrated in Chapter 5.

Several lessons were learned from this study. First, the developers of the applications felt that the crashes revealed important faults in the code. Several crashes were reported on each application's bug-reporting site. In response, some of them have already been fixed in subsequent releases of the applications. For example, Bugs #1536224 [6], #1536229 [7], and #1536205 [8], (SourceForge-assigned numbers) have been fixed by the developers of FreeMind.

Second, the study provided evidence that the intuition behind using the GUI's run-time state to find sets of interacting events was useful. Upon closer examination,

[6]http://sourceforge.net/tracker/?func=detail&atid=107118&aid=1536224&group_id=7118

[7]http://sourceforge.net/tracker/?func=detail&atid=107118&aid=1536229&group_id=7118

[8]http://sourceforge.net/tracker/?func=detail&atid=107118&aid=1536205&group_id=7118

several test cases that caused crashes had executed events that shared some code elements. The first evidence of the usefulness of the state-based feedback approach was apparent even with the seed test suite.

Bug#1536205 of FreeMind was detected using a test case from the seed suite. It caused a NullPointerException when *reverting* back from a newly created FreeMind map to its previously saved version. The test case contained two system-interaction events: e_1 – *Create a new FreeMind map*, and e_2 – *Revert*. FreeMind starts with a default map; event e_1 creates a new map with one node; event e_2 reverts the map back to the previously saved version. These events are related using predicate **Case 2** (described in Section 3.3), *i.e.*, $e_1 \xrightarrow{2(1)} e_2$. When executed together, they modify the map object; executed individually, e_2 does not change the state, as there is no saved map.

The crash occurred because the event handlers of e_1 and e_2, contained in the `NewMapAction.class` and `RevertAction.class`, respectively, improperly handled the map object's instance variable `file` used to keep track of the file corresponding to the map. A new map object, created by e_1 has no associated file; the variable `file` remains `null`. A subsequent *Revert* event invokes `createRevertXmlAction(file)` that in turn invokes `file.getAbsolutePath()`. With `file` being `null`, a NullPointerException is thrown.

This example reinforced the intuition linking the run-time state resulting from the execution of events to interactions among event-handler code. The ESIG test cases were more interesting and provided additional insights. The remainder of this section describes the ESIG test cases in detail and the causes of the crashes they detected.

Crash 1: One test case, which executed an event in a modal window followed by the window's termination event, then two events in another modal window followed by that window's termination event, caused a ConnectException in GanttProject; the sequence was attempting to export the current project and publish it on an FTP server. The test case contained five events: e_1 – *Set FTP server URL*, e_2 – *Save settings*, e_3 – *Choose export file type*, e_4 – *Choose to publish on FTP server* and e_5 – *Export and publish project.* Event e_1 is a system-interaction event in a modal window titled `Settings`; e_2 is a termination event in the same window; e_3 and e_4 are system-interaction event in another modal window titled `Export` and e_5 is a termination event in that window. The relationships between the events are $e_1 \xrightarrow{1(3)} e_3 \xrightarrow{2.1(2)} e_4$. Setting an incorrect FTP server URL with e_1 and saving it with e_2, exporting the current project to a *Raster Image* file (for e_3) and choosing to publish it on an FTP server at the same time (e_4), causes e_5 to try to publish the project on the FTP server with a incorrect URL, resulting in a crash.

The crash occurred because the event handlers for e_1, e_2, e_3, e_4, and e_5, contained in `NetworkOptionPageProvider.class`, `SettingsDialog.class`, `ExportC-hooserPage.class`, `ExportChooserPage.class`, and `ExportFinalizationJobIm-p.class`, respectively, made different assumptions about the FTP server URL. Event e_5, performed by clicking `OK` in the `Export` window, exports the current project into a file and publishes it on an FTP server. During publishing, it obtains the FTP server settings, including the URL, which it (incorrectly) assumes to be a valid address. An invocation of `openConnection` on the invalid URL causes the crash.

Lessons Learned: This example demonstrated that event handlers interact in complex ways. Because event handlers may have been developed by different programmers, they may have made incorrect assumptions about the validity of shared objects, leading to integration problems. Moreover, Context 3 (described in Section 3.3), which handles interactions among events contained in multiple windows, is extremely useful because achieving the combined effect of events in a modal window requires the execution of the window's termination event.

Crash 2: An event sequence that executed across two *cascading* (one opened by the other) modal windows, followed by the parent modal window's termination event, caused a NullPointerException in GanttProject. The sequence tried to change the type of a file to be imported after it had selected the name of the file from a list. Developers of the software had expected that users would select the type first, and then select the file name. The sequence contained five events: e_1 – *Choose import file type* T_1, e_2 – *Choose import file*, e_3 – *Click* `OK` *to close* `FileChooser` *window*, e_4 – *Choose import file type* T_2, e_5 – *Click* `OK` *to close import file window.* Events e_1 and e_4 are system-interaction events in a modal window titled `Import`; event e_5 is the window's termination event. Event e_2 is a system-interaction event in the `FileChooser` modal window; event e_3 is the window's termination event. Note that another event is used after e_1 to open the `FileChooser` window; it is not shown here because it is not a part of the EIG. The relationships between the events are $e_1 \xrightarrow{3(2)} e_2 \xrightarrow{3(2)} e_4$. The test case selects a file type (e_1) and the file name (e_2), but then changes the type (e_4) without changing the file name. Executing the termination event results in a crash.

The crash was caused because the event handlers of the events e_1, e_2, e_3, e_4, and e_5 contained in `ImporterChooserPage.class`, `FileChooserPage.class`, `JFileChooser.closeWindow()`, `ImporterChooserPage.class`, and `ImportFileWizardImpl.onOkPressed()`, respectively, failed to keep track of the relationship among a file's name, its type, and the file suffix in the name. GanttProject records the import file's type in `myProject` and its name in `myState`. `ImportFileWizardImpl.onOkPressed()` examines the selected type and assumes that the file name will have the correct suffix. If, however, the filename does not have the assumed suffix, an object `Open` remains `null`. Execution of `Open.load(file)` results in a NullPointerException.

Lessons Learned: This example reinforced the original motivation for developing new techniques for model-based GUI testing. Developers often cannot predict how users will use the software. They typically test the software for a small number of obvious, predictable use cases. Developers need to use automated techniques to test their software for a larger number of unpredictable event sequences.

Summary: The above crashes (and the ones not presented here) illustrated several important points: (1) The event handlers for events are typically implemented in multiple classes. Static analysis that is limited to intra-class analysis fails to reveal problems with interacting events. (2) With the increasing flexibility of new user interfaces, programmers must take steps to ensure that their software works correctly for a large input space. They should check the validity of objects whenever possible before use; text fields in particular should be restricted to the smallest input domains possible. For example, an autosave-frequency text-field in jEdit caused a crash after a negative number was entered. The developers had simply checked whether the length of the entered text was non-zero, which is clearly inadequate. (3) Most of the test cases that revealed the crashes did not add to the code coverage (statement and edge) of the seed test suite. They were able to detect the faults because of the permutations of events that were executed on the GUI.

4.3 Study 2 of ESIG-Based Approach: Digging Deeper via Seeded Faults and In-House Applications

The previous study raised some important questions. One fundamental question that comes to mind pertains to the cause(s) of the added effectiveness, *i.e.*, *Is the added effectiveness an incidental side-effect of the events, event interactions, and lines-of-code that the ESI test cases cover and their length; or is it really due to targeted testing of the identified ESI relationships?* The empirical study presented in this section is designed specifically to address the question of how the fault-detection effectiveness of the suite obtained by the feedback-directed technique compares to that of other "similar" suites, where similarity is quantified in terms of statement coverage, event coverage, edge coverage, and size (number of test cases).

This question were answered by selecting four pre-tested GUI-based applications, and generating and executing 2-way EIG-based and 3-way ESIG-based test suites on them. Additional test suites are generated that are similar to the ESIG-based suite in terms of the aforementioned characteristics and are at least 3-way interacting, and their fault-detection effectiveness is compared. Fault-detection effectiveness is measured on a per-test-suite basis in terms of the number of faults detected. The faults were studied, pinpointing reasons for why some of them remained undetected by the new technique.

4.3.1 Preparing the Subject Applications and Test Oracles

Four open-source applications, called the TerpOffice suite, consisting of Paint, Present, SpreadSheet and Word, have been selected for the study.[9] Table 4.4 shows key metrics for TerpOffice. These applications are selected very carefully for a number of reasons. In particular, to minimize threats to external validity, the selected applications are non-trivial, consisting of several GUI windows and widgets. For reasons described later, artificial faults were seeded in the applications – this required access to source code, bug reports, and a CVS development history. To avoid (the often difficult) distinction between GUI code and underlying "business logic," GUI-intensive applications were selected, *i.e.*, most of the source-code implemented the GUI. Finally, the tools implemented for this research, in particular for reverse engineering, are well-tuned for the Java Swing widget library – the applications had to be implemented in Java with a GUI front-end based on Swing components. As is the case with all empirical studies, the choice of subject applications introduces

[9]Detailed specifications, requirements documents, source code CVS history, bug reports, and developers' names are available at `http://www.cs.umd.edu/users/atif/TerpOffice/`.

some significant threats to external validity of the results; these (and other) threats are noted in Chapter 7.

Subjects	**Windows**	**Widgets**	**LOC**	**Classes**	**Methods**
Paint	16	301	11,803	330	1,253
Present	11	322	10,847	292	2,057
SpreadSheet	9	176	5,381	135	746
Word	26	617	9,917	197	1,380
TOTAL	62	1,416	38,398	954	5,436

Table 4.4: TerpOffice Applications

For the purpose of this study, a GUI fault is a mismatch, detected by a test oracle, between the "ideal" (or expected) and actual GUI states. Hence, to detect faults, a description of ideal GUI execution state is needed. This description is used by test oracles to detect faults in the subject applications. There are several ways to create this description. First is to manually create a formal GUI specification and use it to automatically create test oracles. Second is to use a capture/replay tool to manually develop assertions corresponding to test oracles and use the assertions as oracles to test other versions of the subject applications. Third is to develop the test oracle from a "golden" version of the subject application and use the oracle to test other versions of the application. The first two approaches are extremely labor intensive because they require the development of a formal specification and the use of manual capture/replay tools; the third approach can be performed automatically and has been used in this study.

Several faults were seeded in each application. However, in order to avoid fault interaction and to simplify the mapping of application failure to underlying fault, multiple versions of each application were created; each version was seeded with exactly one fault. Hence, a test case detects a fault i if there is a mismatch between version i (*i.e.*, the version that was created by seeding fault i) and the original.

The process used for fault seeding was similar to the one used in earlier work [40, 57]. Details are not replicated here. In summary, during fault seeding, 12 classes of known faults were identified, and several instances of each fault class were artificially introduced into the subject program source code statements that were covered by the 2-way test cases, thereby ensuring that these statements were part of executable code. Care was taken so that the artificially seeded faults were similar to faults that naturally occur in real programs due to mistakes made by developers. The faults were seeded "fairly," *i.e.*, an adequate number of instances of each fault type were seeded. Several graduate students were employed to seed faults in each subject application; they created 263, 265, 234, and 244 faulty versions for Paint, Present, SpreadSheet, and Word, respectively.

4.3.2 Generating and Executing the ESIG-Based Test Suite

The reverse engineering process was used to obtain the EIGs for the original versions of each application. The sizes of the EIGs, in terms of nodes and edges, are shown in Table 4.5. These numbers are important as they determine the number of generated test cases and their growth in number as test-case length increases.

The EIGs were then used to generate all possible 2-way test cases. The numbers generated were exactly equal to the number of edges in the EIGs – it was quite feasible to execute such numbers of test cases in little more than a day on the

	Paint	Present	SpreadSheet	Word
#EIG Nodes	300	321	175	616
#EIG Edges	21,391	32,299	6,782	28,538
#ESIG Nodes	102	50	45	75
#ESIG Edges	233	233	197	204

Table 4.5: ESIG vs. EIG Sizes

	Paint	Present	SpreadSheet	Word
Total Faults	263	265	234	244
2-way EIG-detected Faults	147	139	139	183
3-way ESIG-detected Faults (only new faults)	47	52	39	36

Table 4.6: ESIG vs. EIG Fault Detection

50-machine cluster. The test cases were executed on their corresponding "correct" applications; the GUI state was collected and stored.

While new software versions were being obtained (via fault seeding as discussed in Section 4.3.1), the 2-way EIG-based test suites and GUI state were used to obtain all possible 3-way ESIG covering test cases. The sizes of the ESIGs are shown in Table 4.5. The table shows that the ESIGs are much smaller than the corresponding EIGs. Due to the small number of nodes and edges, the number of 3-way covering test cases was 2531, 2080, 2069, and 2345 for Paint, Present, SpreadSheet, and Word, respectively.

The 2-way EIG- and 3-way ESIG-based test cases were then executed on the fault-seeded versions of the applications. The number of faults detected is shown in Table 4.6. Note that the last row reports the number of "new" faults detected by the ESIG suite. This table shows that ESIG-based suites are able to detect a large number of faults missed by the EIG.

4.3.3 Developing "Similar" Suites

As mentioned earlier, this study required the development of several new test suites. To minimize threats to validity, the suites needed to satisfy a number of requirements, discussed next.

Previous studies have shown that statement, event, and EIG-edge coverage, and size (number of test cases) play an important role in the fault-detection effectiveness of a test suite [56]. For example, a small test suite that covers few lines of code is more likely detect fewer faults than another larger suite that covers many more lines. To allow fair comparison of fault-detection effectiveness, test suites that have the *same statement, event*, and *edge coverage*, and *size (number of test cases)* as that of ESIG-based test suites are needed.

Previous studies have also shown that long test cases (number of EIG events) fare better than short ones in terms of the number of faults that they detect [60]. To ensure that the new suites do not have any unfair disadvantage, all their test cases have at least 3 EIG events (note that all the ESIG test cases have only 3 ESIG/EIG events).

Because the above criteria (same statement, event, and edge coverage, and size) may be met by a large number of test suites (with varying fault-detection effectiveness), the process of generating different suites and comparing them to the ESIG-based suites needed to be repeated several times. In this study, 700 test suites were generated per application and their fault-detection effectiveness was compared to that of the ESIG suite.

GUI test cases are expensive to execute (*e.g.*, delay for timing, etc) – each test case can take up to 2 minutes to execute (on average, each requires 30 seconds). The 700 suites each for Paint, Present, SpreadSheet, and Word, contained 1,054,064, 860,324, 850,808, and 974,235 test cases, respectively. Each of these 3,739,431 test cases, needed to be run on each fault-seeded version, would have taken several years on the available 50-machine cluster, an impractical task. Other researchers, who have also encountered similar issues of practicality, have circumvented this problem by creating a *test pool* consisting of a large number of test cases that can be executed in a reasonable amount of time [9]. Each test case in the pool is executed only once and its execution attributes *e.g.*, time to execute and faults detected are recorded. Multiple test suites are created by carefully selecting test cases from this pool. Their execution is "simulated" by combining the attributes of constituent test cases using appropriate functions (*e.g.*, *set union* for faults detected). This research will also employ the test pool approach to create a large number of test suites. Note that the test-pool-based approach introduced some threats to validity, which are noted in Chapter 7.

Finally, to avoid introducing any human bias when generating these test cases, a randomized guided mechanical process was used. A related approach was employed by Rothermel *et al.* [47] to create sequences of commands to test command-based software. In their approach, each command was executed in isolation and test cases were "assembled" by concatenating commands together in different permutations. Because GUI events (commands) enable/disable each other, most arbitrary permutations result in unexecutable sequences. Hence, the EIG model was used to obtain only executable sequences.

Test cases were generated in batches of increasing lengths, measured in terms of the number of EIG events. It was required that each EIG edge be covered by at least N test cases of a particular batch. Moreover, each fault-seeded statement was covered by at least M test cases of the overall pool. The test case generation process started by generating (using a process described in the next paragraph) the batch of length-3 test cases until each EIG edge was covered by at least N test cases; they were all executed and their statement coverage was evaluated; the next (and all subsequent) batch was generated ONLY IF each fault-seeded statement was not yet covered by at least M test cases.

The process of generating each batch of length i test cases begins by initializing a `frequency` variable for each EIG edge to *zero*. Then for each event in the EIG, form a list of all outgoing edges; select the edge that has the lowest `frequency`, breaking ties via random selection. Follow the selected edge to its destination event; repeat the `frequency`-based selection and follow-the-edge process until the desired length is obtained; go to the next EIG event. Stop when all EIG events have been covered and all `frequency` $\geq N$.

The above algorithm was guaranteed to stop because all faults had been seeded in lines that were executable by the 2-way test cases; the count for each statement would ultimately reach M and stop. Finally, all the ESIG-based test cases were added to the pool.

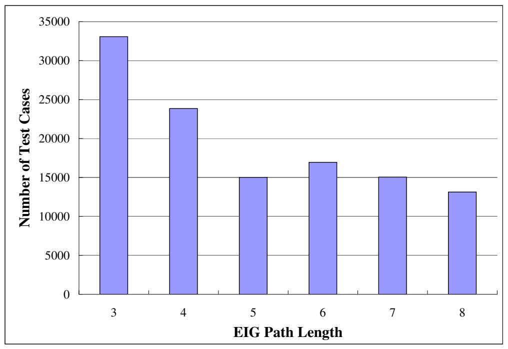
In this study, $N = 10$ and $M = 15$. This choice was dictated by the availability of resources. As described earlier, all the test cases needed to be executed on the fault-seeded versions of their respective application. Even with the 50-machines cluster running the test cases in parallel, the entire process took over four months.

The total number of test cases per application is shown in Column 2 of Table 4.7. The length distribution of the test cases is shown as a histogram in Figure 4.7. As expected, longer test cases were able to cover more EIG edges than the short ones; hence fewer long test cases were needed to satisfy the coverage requirements.

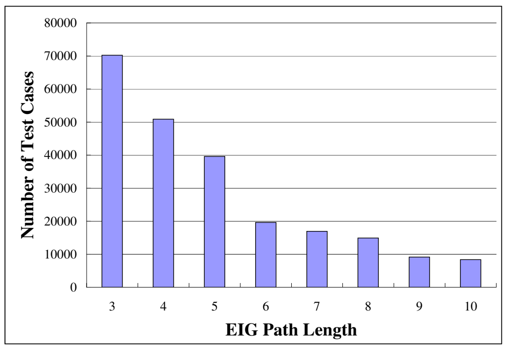
After all the runs had completed, several matrices per application were obtained: (1) the fault matrix, which summarized the faults detected per test case and (2) for each coverage criterion (event, edge, statement), a coverage matrix, which summarized the coverage elements covered per test case.

This test pool was then used to obtain coverage-adequate suites. For example, event-adequate suites were obtained by maintaining sets of test cases that covered each ESIG event. Test cases were selected without replacement from each set and duplicates eliminated, ensuring that each event was covered by the resulting suite. A similar process was used for edge and statement coverage. The process was repeated 100 times to yield 100 suites. The average size of the suites is shown in Columns 3–5 of Table 4.7.

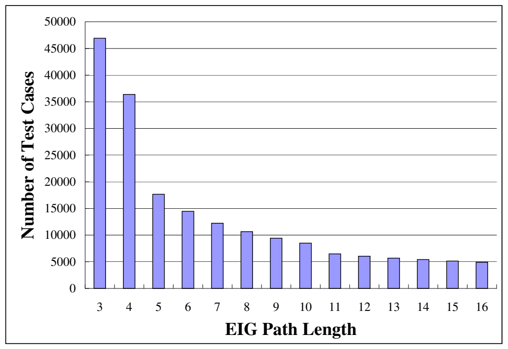
Finally, T_R was constructed using random selection without replacement ensuring that the final size of T_R was the same as that of the ESIG suite. A total of 100 such suites per application were obtained. Similarly, each of the suites T_E, T_I, T_S were augmented with additional test cases, selected without replacement at random from the pool to yield T_{R+E}, T_{R+I}, T_{R+S}, respectively. The sizes of all these



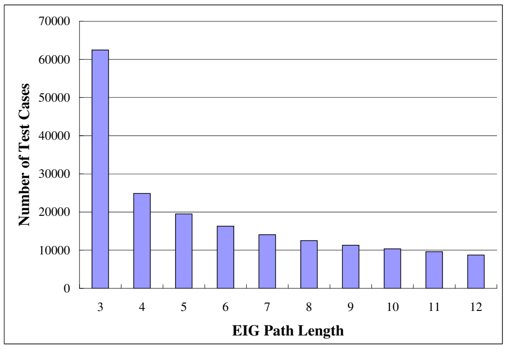
Paint



Present



SpreadSheet



Word

Figure 4.7: Histograms of Test Case Lengths in Pool

suites, as expected, was equal to the size of the ESIG suite.

Note that the fault-detection effectiveness of each test suite can be obtained directly from the fault matrix of the test pool without rerunning the test cases. The results are shown in Figure 4.8 as distributions. The box-plots provide a concise display of each distribution, each consisting of 100 data points. The line inside each box marks the median value. The edges of the box mark the first and third quartiles. The whiskers extend from the quartiles and cover 90% of the distribution; outliers are shown as points beyond the whiskers. Visual inspection of the plots

	Test Pool	T_E Event	T_I Edge	T_S Stmt.	T_R, T_{R+E} T_{R+I}, T_{R+S}
Paint	119,583	103	190	123.64	2531
Present	231,680	50	264	18.24	2080
SpreadSheet	191,966	45	173	14.08	2069
Word	192,042	84	248	30.35	2345

Table 4.7: Test Pool and Average-Suite Sizes

shows that the fault-detection effectiveness of the ESIG-generated test suite (shown as an asterisk) is better than that of individual similar-coverage and similar-sized suites.

A *single sample t-test* was used to test for the significance of the difference between the observed fault-detection of the ESIG suite and the mean of each distribution. The null hypothesis is that the two values are not significantly different; the alternate hypothesis is that they are significantly different. Note that a separate test is needed per (mean of each distribution, fault-detection of the ESIG suite value) pair. The resulting p-values were all more than 0.99. Hence the null hypothesis was rejected and the alternate hypothesis was accepted; there is a significant difference between fault-detection of the ESIG suite and the mean fault-detection of each of the "similar" suites. Test cases that make up ESIG suite are better at detecting faults compared to test cases that cover essentially the same events, edges, and statements. This result helps to answer the primary question raised in this study.

4.3.4 Discussion

Now the details of why the ESIG-based test suites were able to detect more faults than other test suites is presented. Specifically three related issues are going

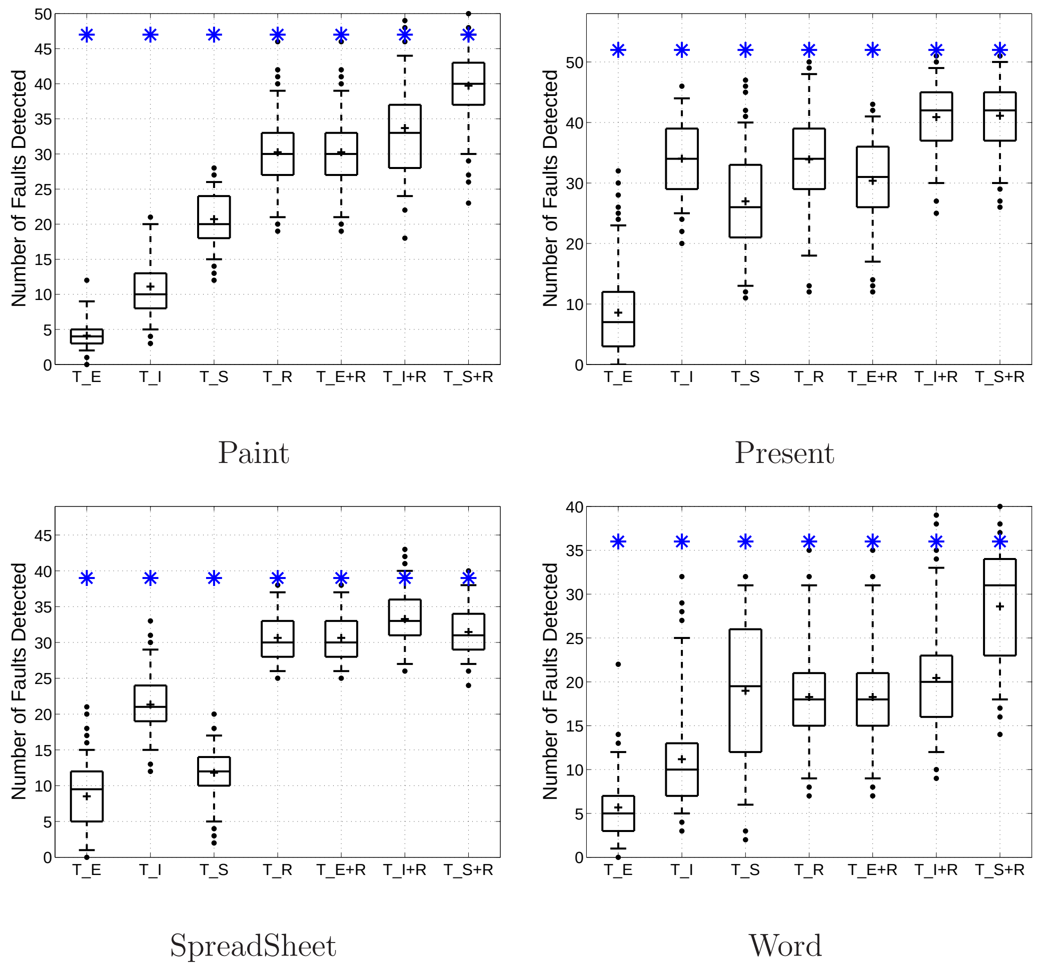


Figure 4.8: Fault Detection Distribution

to be considered: reachability, manifestation, and number of test cases. Note that the first two issues are related to the RELAY model [46, 51] of how a fault causes a failure on the execution of some test. *Reachability* is defined as the coverage of the statement in which a fault was seeded, and *manifestation* as the fault leading to a failure so that the GUI-based test oracle could detect it. As observed in [46], both are necessary conditions for fault detection. The data in Figure 4.8 indicates that the ESIG-based test suites were able to outperform their coverage-adequate equivalent counterparts. Hence, they must have been more successful than their counterparts at the combination of reachability and manifestation of several faults. This behavior is due to the nature of the ESI relationship, which is based on observed GUI state, and hence the software's output. Executing test cases that focus only on ESI events increases the likelihood that a fault will be revealed on the GUI, and hence detected by the GUI-based test oracle. Although this behavior is not attempted in great detail here (indeed this is a direction for future research), some quantitative data is presented to show evidence of this phenomenon by studying the test cases in the pool.

Figure 4.9 shows data of the number of test cases that cover a fault-seeded statement, and the number of test cases in the pool that detect it. The figure is divided into four parts, one for each subject application. Each part is a 3-column table (long tables are wrapped); Column 1 is the seeded fault number, corresponding to a statement in which the fault was seeded; Column 2 is the number of test cases that detected the fault; Column 3 is the number of test cases that did not detect the fault even though they executed the fault-seeded statement; the fault number entry is shaded if at least one ESI test case detected it. For example, the statement that contained Fault 21 of Paint was executed by a total of 856 test cases, of which 11 detected it; the fault was detected by at least one ESI test case. On the other hand, Fault 127 of Paint was not detected by any ESI test case; it was however detected by 42 of the total 42+267 test cases. Hence, a statement-coverage adequate test suite would have a probability of $\frac{42}{42+267}$ of detecting this fault (assuming that faults are independent). The data in Figure 4.9 is in fact sorted by this probability, showing the "hardness" of the fault for statement-coverage adequate test suites. This data helps

Paint					
158	1	1247	**248**	3	115
159	1	1173	**255**	3	115
65	3	721	**256**	3	115
36	4	869	261	3	115
52	5	830	**262**	3	115
64	5	772	42	10	381
33	6	856	**47**	21	762
34	7	843	46	10	359
90	6	722	93	21	748
179	1	120	98	21	730
180	1	118	68	21	729
246	1	117	54	11	379
249	1	117	45	12	383
191	1	116	30	4	127
192	1	116	31	4	127
89	7	773	43	12	373
211	1	110	67	23	707
212	1	110	**124**	3	89
50	10	934	97	27	771
101	24	2206	41	15	402
102	23	2093	87	5	127
85	8	720	73	28	703
22	9	796	72	31	744
77	9	735	44	15	339
57	4	311	86	11	180
21	11	845	14	1	15
51	13	874	120	15	220
56	5	331	**32**	10	122
19	13	850	**253**	9	109
20	14	891	119	19	216
177	2	125	**259**	11	107
96	12	711	59	33	314
55	6	354	58	36	337
100	12	704	123	25	207
247	2	116	128	39	270
250	2	116	127	42	267
260	2	116	**134**	853	3273
76	14	781	**178**	28	99
26	2	108	**229**	28	98
18	18	920	**200**	357	372
66	15	764	**136**	747	446
69	14	688	**137**	557	260
37	17	834			
74	16	755			
75	15	707			
92	15	698			
71	15	696			
99	17	769			
125	2	90			
126	2	90			
84	17	762			
48	16	717			
80	16	714			
94	16	705			
83	16	704			
82	17	743			
79	17	719			
88	1	42			
81	16	664			
70	18	746			
91	18	738			
251	2	82			
252	2	82			
17	24	978			
78	19	762			
95	19	761			
245	3	115			

Present					
214	1	262	**94**	45	479
215	21	4242	157	46	484
212	4	520	222	46	484
33	2	259	**218**	91	912
73	4	257	**127**	48	476
61	6	255	177	49	481
34	6	250	102	44	431
30	7	254	151	19	169
62	7	254	**106**	50	425
71	7	250	57	26	207
72	7	250	58	30	230
80	7	249	59	30	230
32	8	253	105	54	411
36	8	248	**154**	133	862
38	8	248	**198**	107	601
162	8	247	**197**	115	593
192	9	258	**199**	118	590
216	10	282	182	45	217
56	8	225	**196**	124	584
29	9	252	76	46	215
161	131	3119	**115**	96	428
40	25	580	**89**	103	421
8	177	4086	48	53	210
167	121	2780	**47**	54	209
66	11	250	99	98	377
75	11	250	**159**	682	2479
166	138	3112	55	58	205
194	23	517	**152**	715	2446
165	139	3111	123	106	357
178	12	260	118	60	202
185	12	255	122	63	199
43	27	568	**211**	169	355
6	198	4065	**224**	168	209
5	177	3630	**93**	301	223
9	206	4057	**90**	310	214
7	208	4055	**202**	313	211
41	30	575	**117**	314	210
85	220	4043	**201**	322	202
12	223	4040	**203**	332	192
16	32	573	**200**	333	191
78	228	4035	**147**	933	325
187	29	508	**148**	936	322
104	26	439			
17	34	571			
229	75	1216			
221	32	498			
96	32	492			
137	15	229			
153	267	3996			
95	33	491			
184	35	495			
52	18	245			
227	37	490			
111	37	487			
186	37	487			
150	13	168			
82	39	491			
158	39	491			
226	39	491			
225	41	489			
128	41	483			
156	57	668			
220	42	488			
108	38	437			
176	44	486			
219	84	919			
79	45	485			

SpreadSheet		
226	12	1130
232	13	1048
33	172	13445
34	179	13438
163	129	9679
233	14	1037
164	133	9675
218	35	2429
180	67	4512
15	25	1681
200	25	1681
214	37	2427
220	37	2427
160	69	4510
161	70	4509
133	12	772
182	71	4508
230	17	1053
47	2	123
48	2	123
17	28	1678
28	311	18223
231	19	1051
8	57	3150
208	77	4224
192	83	4496
234	19	1025
32	19	1010
39	87	4481
210	82	4219
2	206	10485
54	393	19389
209	89	4212
221	67	3140
20	36	1670
204	69	3138
211	93	4208
41	34	1512
38	105	4463
191	74	3100
45	3	122
130	20	786
5	84	3123
53	53	1916
224	10	300
37	20	576
55	68	1074
225	201	3006
56	76	1066
36	44	569
44	17	164
18	237	1469
229	738	404
30	745	397
228	746	396
42	1030	516
43	1030	516
31	746	362
132	615	191
131	618	188
78	546	157
79	546	157
75	1155	330
136	1161	324
135	614	170
134	615	169
76	1088	296
77	544	137

Word		
156	126	2336
209	68	1238
157	131	2331
158	130	2146
88	4	65
93	4	65
210	74	1185
211	74	1185
212	70	1072
155	167	2295
159	175	2101
102	5	57
30	7	75
66	6	63
67	6	63
79	6	63
80	6	63
83	6	63
101	6	56
91	7	62
76	7	55
103	7	55
81	8	61
85	8	61
25	10	72
29	10	72
114	12	72
57	11	58
97	11	58
116	14	70
117	17	65
118	20	64
120	21	62
121	21	62
160	54	153
115	27	57
119	30	54
32	76	100
6	122	129
24	56	28
122	60	24

Figure 4.9: Test Cases Covered Faulty Statements and Their Fault Detection.

better interpret the results of Figure 4.8. First, the ESI test suites did detect many of the seeded faults. Second, they did better than T_S because they detected many of the "hard" faults (this was most apparent in Paint and SpreadSheet). Third, some faults were detected by many of the test cases that executed the statement. For example, Fault 136 in Paint was detected by 747 of the total (747+446) test cases that executed the statement in which it was seeded. The fault was seeded in the handler of a termination event that closes the `Attributes` window and applies the attributes (if any have changed) to the current image on the main canvas. The seeded fault caused the image size to be computed incorrectly, resulting in an incorrectly sized image whenever at least one attribute in the `Attributes` window is modified by the user. Because there are many permutations of modifying the attributes, a large number of test cases are able to detect this fault (12 in the ESIG test suite and 735 in the rest of the test pool). In general, statement coverage adequate test suites do really well for these types of faults that can be triggered in many different ways.

Fourth, several faults were detected by very few test cases. Fault 34 of Paint is an example. This fault flips the conditional statement in an event handler of a type of curve tool. The condition is to check whether the curve tool is currently selected; if yes, then the curve stroke is set according to the selected line type for the curve tool. Due to the fault, the curve stroke is incorrectly set, resulting in an incorrect image to be drawn on the main canvas only when the event sequence: $<$ *Select CurveTool*; *Select Line Type*; *Draw On Canvas* $>$ is executed. If the first two events are not executed, then *Draw On Canvas* does not trigger the fault even if the statement containing the seeded fault is executed. Hence, although there are many test cases that cover the faulty statement (850), only 7 test cases in the test pool detected the fault. One of them is the ESI test case; ESI-relationships were found between the three events. In general, statement coverage adequate test suites do not do well for faults that are executed by many event sequences but manifest as failures in very few cases.

The size of a suite seems to play a very important role in fault-detection. Indeed, the T_R test suites, which were the same size as the ESIG-based suites, did better (in most cases) than their coverage-adequate counterparts. This behavior is probably an artifact of the density of the fault matrices. A large number of test cases are successful at detecting many "easy" faults. Even if test cases are selected at random, given adequate numbers, they will be able to detect a large number of these easy faults. For example, given 192,042 test cases in the pool for Word and the size of T_R being 2345, the probability that Fault 24, which is detected by more than 84 test cases in the pool, is detected by at least one test case in T_S is 0.49 – this is quite high. In Figure 4.10, the probability that a random suite of its corresponding ESI-suite size would detect a particular fault is shown. This data shows that many of these difficult faults are detected by at least one ESIG-based test case, improving their fault-detection effectiveness. Moreover, 16 faults in Word have a detection probability of more than 0.25. This number is much larger for the other three applications, helping to understand why T_R and the other suites that included randomly selected test cases did so well.

Finally, to examine why some of the faults were not detected, each fault was

Paint			
158	0.0212	41	0.2745
159	0.0212	44	0.2745
179	0.0212	120	0.2745
180	0.0212	74	0.2899
246	0.0212	**48**	0.2899
249	0.0212	80	0.2899
191	0.0212	94	0.2899
192	0.0212	83	0.2899
211	0.0212	81	0.2899
212	0.0212	37	0.3049
88	0.0212	99	0.3049
14	0.0212	84	0.3049
177	0.0419	82	0.3049
247	0.0419	79	0.3049
250	0.0419	**18**	0.3196
260	0.0419	70	0.3196
26	0.0419	91	0.3196
125	0.0419	78	0.3340
126	0.0419	95	0.3340
251	0.0419	119	0.3340
252	0.0419	**47**	0.3619
65	0.0622	93	0.3619
245	0.0622	98	0.3619
248	0.0622	68	0.3619
255	0.0622	102	0.3886
256	0.0622	67	0.3886
261	0.0622	101	0.4016
262	0.0622	**17**	0.4016
124	0.0622	123	0.4143
36	0.0820	97	0.4388
57	0.0820	73	0.4507
30	0.0820	**178**	0.4507
31	0.0820	**229**	0.4507
52	0.1014	72	0.4848
64	0.1014	59	0.5064
56	0.1014	58	0.5371
87	0.1014	128	0.5659
33	0.1205	127	0.5929
90	0.1205	**200**	0.9995
55	0.1205	**137**	1.0000
34	0.1391	**136**	1.0000
89	0.1391	**134**	1.0000
85	0.1573		
22	0.1751		
77	0.1751		
253	0.1751		
50	0.1926		
42	0.1926		
46	0.1926		
32	0.1926		
21	0.2097		
54	0.2097		
86	0.2097		
259	0.2097		
96	0.2264		
100	0.2264		
45	0.2264		
43	0.2264		
51	0.2428		
19	0.2428		
20	0.2588		
76	0.2588		
69	0.2588		
66	0.2745		
75	0.2745		
92	0.2745		
71	0.2745		

Present			
214	0.0090	**156**	0.4020
33	0.0179	157	0.4073
73	0.0354	222	0.4179
212	0.0354	218	0.4335
34	0.0527	**127**	0.4916
61	0.0527	**177**	0.5313
30	0.0612	**102**	0.5599
62	0.0612	**151**	0.5793
71	0.0612	106	0.5869
72	0.0612	**57**	0.6051
80	0.0612	58	0.6156
32	0.0696	**59**	0.6191
36	0.0696	**105**	0.6456
38	0.0696	**154**	0.6551
56	0.0696	**198**	0.6643
162	0.0696	**197**	0.6733
29	0.0780	**199**	0.6933
192	0.0780	**182**	0.6987
216	0.0862	**196**	0.7120
66	0.0944	76	0.7146
75	0.0944	**115**	0.7803
178	0.1026	**89**	0.7823
185	0.1026	**48**	0.7975
150	0.1106	**47**	0.7975
137	0.1265	**99**	0.8324
52	0.1498	**159**	0.8441
151	0.1575	**55**	0.8469
215	0.1725	**152**	0.8626
194	0.1873	**123**	0.8663
40	0.2019	**118**	0.8722
57	0.2090	**122**	0.9101
104	0.2090	**211**	0.9339
43	0.2161	**224**	0.9390
187	0.2301	**93**	0.9407
41	0.2371	**90**	0.9412
58	0.2371	**202**	0.9453
59	0.2371	**117**	0.9500
16	0.2507	**201**	0.9505
96	0.2507	**203**	0.9979
221	0.2507	**200**	0.9984
95	0.2574	**147**	0.9998
17	0.2641	**148**	0.9998
184	0.2707		
111	0.2837		
186	0.2837		
227	0.2837		
108	0.2902		
82	0.2965		
158	0.2965		
226	0.2965		
128	0.3091		
225	0.3091		
220	0.3153		
102	0.3276		
176	0.3276		
79	0.3336		
94	0.3336		
182	0.3336		
76	0.3396		
157	0.3396		
222	0.3396		
127	0.3514		
177	0.3572		
106	0.3630		
48	0.3800		
47	0.3856		
105	0.3856		

SpreadSheet	
47	0.0214
48	0.0214
45	0.0320
224	0.1027
226	0.1219
133	0.1219
232	0.1314
233	0.1408
230	0.1683
44	0.1683
231	0.1861
234	0.1861
32	0.1861
130	0.1949
37	0.1949
15	0.2373
200	0.2373
17	0.2617
41	0.3082
218	0.3157
20	0.3230
214	0.3303
220	0.3303
36	0.3793
53	0.4370
8	0.4609
180	0.5162
221	0.5162
55	0.5215
160	0.5266
204	0.5266
161	0.5317
182	0.5368
191	0.5516
56	0.5612
208	0.5659
210	0.5888
192	0.5933
5	0.5977
39	0.6105
209	0.6189
211	0.6351
38	0.6796
163	0.7530
164	0.7635
33	0.8451
34	0.8564
225	0.8869
2	0.8928
18	0.9235
28	0.9657
54	0.9859
77	0.9973
78	0.9973
79	0.9973
135	0.9987
132	0.9987
134	0.9987
131	0.9988
229	0.9997
30	0.9997
228	0.9997
31	0.9997
42	1.0000
43	1.0000
76	1.0000
75	1.0000
136	1.0000

Word	
88	0.0480
93	0.0480
102	0.0596
66	0.0711
67	0.0711
79	0.0711
80	0.0711
83	0.0711
101	0.0711
30	0.0824
91	0.0824
76	0.0824
103	0.0824
81	0.0936
85	0.0936
25	0.1156
29	0.1156
57	0.1264
97	0.1264
114	0.1371
116	0.1580
117	0.1885
118	0.2179
120	0.2274
121	0.2274
115	0.2823
119	0.3083
160	0.4850
24	0.4975
122	0.5216
209	0.5664
212	0.5769
210	0.5972
211	0.5972
32	0.6070
6	0.7767
156	0.7874
158	0.7976
157	0.8001
155	0.8716
159	0.8836

Figure 4.10: Probability of Detecting Faults by Random Test Cases.

	Ignored Widget Properties	**Non-GUI Failure**	**Longer Sequence**	**Masked Error**	**Crash**	**Total**
Paint	0	0	1	6	0	7
Present	13	0	4	0	0	17
SpreadSheet	9	2	8	5	3	27
Word	5	0	4	11	0	20

Table 4.8: Undetected Faults Classification

manually examined. It is determined that:

1. several of the faults were in fact manifested as failures on the GUI but the test oracle was not capable of examining these parts of the GUI,
2. very few faults caused failures in non-GUI output,
3. several of the undetected faults require even longer sequences,
4. the effect of several faults was masked by the event handler code even before the test oracle could detect it,
5. some faults crashed their corresponding fault-seeded version.

The numbers of these faults are shown in Table 4.8. The large number of "Ignored Widget Properties" shows the need to improve the test oracles for future work.

This controlled study showed that the automatically identified ESI relationships between events generate test suites that detect more faults than their code-, event-, and event-interaction-coverage equivalent counterparts. Moreover, several of the missed faults remained undetected because of limitations with the automated GUI-based test oracle, and others required even longer sequences.

4.4 Summary

This chapter presented the feedback-directed ESIG-based approach to generate test cases that test multi-way interactions among GUI events. It was based on analysis of feedback obtained from the run-time state during initial seed suite execution. The approach was studied via two independent studies on eight GUI applications. The results of the first study showed that the test cases generated using feedback were useful at detecting serious and relevant faults in the applications. The second study related the effectiveness of the ESIG-based test suite and compared it to the EIG-based suites. It also showed that the added effectiveness of the ESIG suite is due to targeted testing of the identified ESI relationships, not an incidental side-effect of the size of the suite, nor the additional events and code that it covers.

However, as one can see from the studies, although better than the exhaustive approach and EIG-based approach, the number of test cases required for the ESIG-based technique also grows exponentially with the length for most applications, making it difficult to test 5-way and above interactions. Another observation from the studies is that there are certain number of unexecutable test cases in the seed suite generated from the EIG (shown in Table 4.9). Careful examination reveals that many unexecutable seed suite test cases are due to disabled widget that could not be executed.

These two weaknesses are addressed in the next chapter by a new approach that significantly improves upon the ESIG-based approach by generating test cases "in batches." The first batch consists of all possible 2-way covering test cases,

Length	FreeMind	GanttProject	jEdit	OmegaT
2-way	45,026	46,848	24,194	7318

Table 4.9: Unexecutable Test Cases

generated automatically using the existing EIG model of the GUI. This batch is executed and the observed execution behavior of the GUI, captured in the form of widgets and their properties, is used to selectively extend some of the 2-way test cases to 3-way test cases via the ESI relations. The new 3-way test cases are subsequently executed, GUI execution behavior is analyzed, and some are extended to 4-way test cases, and so on. In general, the new "alternating approach" (called ALT) executes and analyzes i-way covering test cases, identifying sets of events that interact in interesting ways with one another (and hence should be tested together), and generates (i+1)-way covering test cases for members of each set. Hence ALT generates "longer" test cases that expand the state space to be explored, yet pruning the "unimportant" states. A side-effect of the batch-style nature of this new approach is that certain aspects of GUI test cases that are revealed only at run-time and impossible to infer statically, *e.g.*, infeasible test cases, are also used to enhance the next batch.

Chapter 5

Alternating Test Case Generation and Execution

This chapter presents ALT, the second feedback-directed test case generation approach. It generates GUI test cases in batches, by leveraging GUI run-time information from a previously run batch to obtain the next batch. Each successive batch consists of "longer" test cases that expand the GUI state space to be explored, yet prune the "unimportant" states. The "alternating" nature of ALT allows it to enhance the next batch by leveraging ESI relationships and enable-disable relationships between GUI events that are revealed only at run-time and non-trivial to infer statically.

An overview of ALT is first presented through the "`Radio Button Demo`" GUI example that was used to illustrate the ESIG-based approach earlier. Then, the ALT test case generation algorithm is discussed followed by an empirical study.

5.1 Overview of ALT

The first three steps of applying ALT for GUI testing are as same as those used in the ESIG-based technique described in Section 4.1. Recall that **(1)**, the EIG model of the GUI is created, **(2)** 2-way covering test cases are generated and executed, and **(3)** ESI relationships between event pairs are identified automatically. Three relations that were found earlier are: $e_2 \xrightarrow{5.1(1)} e_6$, $e_6 \xrightarrow{5(1)} e_2$, and $e_3 \xrightarrow{12(1)} e_5$.

The subsequent steps (starting at step #4) of ALT are as follows:

(4) *Generate 3-way test cases.* The first two ESI relationships are used to extend two of the 2-way covering test cases $< e_2; e_6 >$ and $< e_6; e_2 >$ to $< e_2; e_6; e_2 >$ and $< e_6; e_2; e_6 >$, respectively. This is due to the nature of these particular relationships – the ending event in one ESI relationship is the starting event in the other. In this example, $e_2 \xrightarrow{5.1(1)} e_6$ and $e_6 \xrightarrow{5(1)} e_2$ are connected by e_6, and $e_6 \xrightarrow{5(1)} e_2$ and $e_2 \xrightarrow{5.1(1)} e_6$ are connected by e_2, therefore, they are used to generate 3-way test cases.

ALT uses the "enabling" ESI relationship $e_3 \xrightarrow{12(1)} e_5$ to augment all the 2-way covering test cases that started with e_5 but remained unexecutable earlier. The new 3-way test cases are obtained by appending a prefix $< e_3 >$ to *all* the 2-way covering test cases that start with e_5, thereby yielding: $< e_3; e_5; e_1 >$, $< e_3; e_5; e_2 >$, $< e_3; e_5; e_3 >$, $< e_3; e_5; e_4 >$, $< e_3; e_5; e_5 >$, $< e_3; e_5; e_6 >$, and $< e_3; e_5; e_7 >$. These test cases give a tester an opportunity to observe the effect of e_5, previously unexecuted, on other all events that can potentially follow e_5.

(5) *Execute the new 3-way test cases, obtain new ESI relations, and generate 4-way test cases.* All the GUI states after each event are recorded. This step computes new ESI relations by splitting each 3-way covering test case $< e_x; e_y; e_z >$ into two parts: $< e_x; e_y >$ and e_z; the former is conceptually treated as a single *macro event* E_X and used as input to the existing predicates; the resulting ESI relation is now between E_X (which is really the event sequence $< e_x; e_y >$) and event e_z. The "splitting" of the test case is designed in the above fashion very carefully so that the E_X part would already have been executed in the earlier batch, thereby requiring no new execution.

Consider the event sequence $< e_3; e_5; e_6 >$. This is rewritten as $< E_X; e_6 >$, with E_X being $< e_3; e_5 >$; the semantics of E_X can be imagined as "enter a custom color in an *enabled* text-field w_5"; the ESI predicates are applied. So, E_X influences e_6. Event e_6 alone from the initial state renders an empty circle in the `Rendered Shape` area. However, executing E_X before e_6 changes its behavior, yielding a filled circle instead. Hence, predicate **Case 5.1** (described in Section 3.3) applies; $< e_3; e_5 > \xrightarrow{5.1(1)} e_6$.

Because $< e_3; e_5 > \xrightarrow{5.1(1)} e_6$ and $e_6 \xrightarrow{5(1)} e_2$ (as computed earlier), $< e_3; e_5; e_6 >$ is extended to the 4-way test case $< e_3; e_5; e_6; e_2 >$.

None of the other 3-way test cases are extended because the predicates do not apply.

(6) *Execute the new 4-way test cases, obtain new ESI relations, and generate 5-way test cases.* The sole 4-way test case $< e_3; e_5; e_6; e_2 >$ is rewritten as $< E_X; e_2 >$; hence the semantics of E_X are now "enter a custom fill color and create the shape." Note that due to the nature of the splitting, E_X has already been executed earlier; hence its resulting state is already available for analysis.

It is determined that $< e_3; e_5; e_6 > \xrightarrow{5(1)} e_2$. And as it is already known that $e_2 \xrightarrow{5.1(1)} e_6$, only one 5-way covering test case is generated $< e_3; e_5; e_6; e_2; e_6 >$.

(7) *Execute the new 5-way test case, obtain new ESI relations, and generate 6-way covering test cases.* No new ESI relations are found; hence ALT terminates.

In all, 37 two-way, 9 three-way, 1 four-way, and 1 five-way test cases were generated in this example.

Let's informally examine how the 48 test cases executed the code of the simple

```
                  RBExample::CircleAction(ActionEvent evt) {
☑☐☐☐                  currentShape = SHAPE_CIRCLE;
☑☐☐☐                  if (created) {
☑☐☐☐                      imagePanel.setShape(currentShape);
☑☐☐☐                      imagePanel.repaint(); } }
```

e_1's Event Handler

```
                  RBExample::SquareAction(ActionEvent evt) {
☑☑☑☑                  currentShape = SHAPE_SQUARE;
☑☑☑☑                  if (created) {
☑☑☑☑                      imagePanel.setShape(currentShape);
☑☑☑☑                      imagePanel.repaint(); } }
```

e_2's Event Handler

```
                  RBExample::ColorAction(ActionEvent evt) {
☑☑☑☑                  colorText.setEditable(true);
☑☑☑☑                  currentColor = getColor();
☑☑☑☑                  if (created) {
☑☐☐☐                      imagePanel.setFillColor(currentColor);
☑☐☐☐                      imagePanel.repaint(); } }
```

e_3's Event Handler

```
                  RBExample::NoneAction(ActionEvent evt) {
☑☐☐☐                  colorText.setEditable(false);
☑☐☐☐                  currentColor = COLOR_NONE;
☑☐☐☐                  if (created) {
☑☐☐☐                      imagePanel.setFillColor(currentColor);
☑☐☐☐                      imagePanel.repaint(); } }
```

e_4's Event Handler

```
                  RBExample::CreateAction(ActionEvent evt) {
☑☑☑☑                  if (color.isSelected()) {
☑☑☑☑                      currentColor = getColor(); }
☑☑☑☑                  imagePanel.setFillColor(currentColor);
☑☑☑☑                  imagePanel.setShape(currentShape);
☑☑☑☑                  imagePanel.repaint();
☑☑☑☑                  created = true; }
```

e_6's Event Handler

Figure 5.1: Some Source Code for the "`Radio Button Demo`" GUI - Part 1.

```
            RBExample::ResetAction(ActionEvent evt) {
☑☐☐☐          square.setSelected(true);
☑☐☐☐          none.setSelected(true);
☑☐☐☐          colorText.setText(‘‘black’’);
☑☐☐☐          colorText.setEditable(false);
☑☐☐☐          currentShape = SHAPE_NONE;
☑☐☐☐          imagePanel.setShape(currentShape);
☑☐☐☐          currentColor = COLOR_NONE;
☑☐☐☐          imagePanel.setFillColor(currentColor);
☑☐☐☐          imagePanel.repaint(); }
```

e_7's Event Handler

```
            ImagePanel::paintComponent(Graphics g) {
☑☑☑☑          clear(g);
☑☑☑☑          Graphics2D g2d = (Graphics2D)g;
☑☑☑☑          if (currentShape == SHAPE_CIRCLE) {
☑☑☑☐              if (currentColor == COLOR_NONE) {
☑☑☐☐                  g2d.setPaint(Color.black);
☑☑☐☐                  g2d.draw(circle); }
☑☑☑☑              else {
☑☑☑☑                  g2d.setPaint(currentColor);
☑☑☑☑                  g2d.fill(circle); } }
☑☑☑☑          else if (currentShape == SHAPE_SQUARE) {
☑☑☑☐              if (currentColor == COLOR_NONE) {
☑☑☐☐                  g2d.setPaint(Color.black);
☑☑☐☐                  g2d.draw(square); }
☐☐☑☑              else {
☐☐☑☑                  g2d.setPaint(currentColor);
☐☐☑☑                  g2d.fill(square); } } }
            ImagePanel:setFillColor(int inputColor) {
☑☑☑☑          switch (inputColor) {
☐☑☑☑              case COLOR_BLACK:
☐☑☑☑                  currentColor=Color.black;
☐☑☑☑                  break;
☐☑☑☑              case COLOR_RED:
☐☑☑☑                  currentColor=Color.red;
☐☑☑☑                  break;
☐☑☑☑              case COLOR_GREEN:
☐☑☑☑                  currentColor=Color.green;
☐☑☑☑                  break;
☑☑☑☑              default:
☑☑☑☑                  currentColor=Color.gray; } }
```

The ImagePanel Class

Figure 5.2: Some Source Code for the “`Radio Button Demo`” GUI - Part 2.

application. Figure 5.1 and 5.2 show the event-handler code as well as some helper methods. The event handlers of e_2, e_3 and e_6 have been shown in Section 3.2; more are given here with statement coverage information. The statement coverage is summarized as a vector of 4 checkboxes ☑☑☑☑ associated with each statement. The first box is checked if any of the 2-way test cases executed the corresponding line of code; similarly, the second box is for 3-way test cases; third for 4-way, and fourth for 5-way test cases. For example, in the `ImagePanel` class code, lines 16 and 17 were executed only by 4- and 5-way test cases.

There are several points to note about the code and statement coverage. First, each event has a programmer-defined event handler (w_5, which requires no custom functionality, is the exception). Second, the code is implemented in two classes `RBExample` and `ImagePanel` – any code-based analysis must account for interactions across classes. As shown in the previous chapter, several failures are due to incorrect interactions across classes. Third, event handlers interact either directly or indirectly by using shared variables (*e.g.*, `currentShape`, `created`, `currentColor`) or via method calls (*e.g.*, `setFillColor()`). Detecting such interactions at the code level, especially across classes, is non-trivial. Fourth, while many statements are covered by all types of test cases (*e.g.*, Lines 2-4 in the `ImagePanel` class are executed by 2-, 3-, 4-, and 5-way test cases), a few statements that are guarded by a series of conditional statements are executed by very few test cases (*e.g.*, Lines 16 and 17 in the `ImagePanel` class are executed by the sole 4-way and 5-way test case but was missed by the other 46 test cases.) Finally, although not evident by statement coverage, the 4- and 5-way test cases are able to exercise several combinations of control-flow that are only partially covered by the 2- and 3-way test cases.

The above discussion of code coverage is in no way meant to be a formal analysis of the code-covering ability of the ALT test cases. However, it helps to highlight some important aspects of GUI testing that may be investigated in future research.

5.2 The ALT Algorithm

The steps of ALT are now formalized by presenting an algorithm. Intuitively, the algorithm takes an i-way covering test suite as input, in which each test case is fully executable, splits each of its i-way covering test cases $< e_1; e_2; \ldots; e_i >$ into two parts: (1) a macro event $E_X = < e_1; e_2; \ldots; e_{i-1} >$ and (2) the last event e_i. If E_X and e_i are related via an ESI relationship, then for each event e_x that e_i is ESI related to, a new (i+1)-way covering test case $< e_1; e_2; \ldots; e_i; e_x >$ is added to the suite. An extra step handles previously unexecuted events. This approach preserves the property of the earlier ESIG-based test cases that each pair of adjacent events are related via an ESI relation. It imposes a stronger condition that each preceding sequence starting from the first event is also ESI-related to its subsequent event. Moreover, the alternating approach allows the detection of new ESI relations between newly generated sequences and newly enabled events.

Several helper functions are assumed to be available: (1) $FindState(S_0, E_i)$ that returns the state of the GUI after event sequence E_i has been executed on it, starting in state S_0, (2) *isRelated*(S_0, S_1, S_2, S_3) that returns `TRUE` if at least

```
PROCEDURE::$ALT(T_i)${
//$T_i$ is the $i$-way covering test suite.
//$T_{i+1}$ is the output $(i+1)$-way covering test suite.
  $S_0$= GUI's Initial state; $T_{i+1} = \phi$;                          1
  foreach $test\ case\ tc \in T_i$ do                                    2
    $E_X = SubSequence(tc, 1, Length(tc)-1)$;                            3
    $e_j = Last(tc)$;                                                    4
    $S_1 = FindState(S_0, E_X)$;                                         5
    $S_2 = FindState(S_0, < e_j >)$;                                     6
    $S_3 = FindState(S_0, tc)$;                                          7
    if $isRelated(S_0, S_1, S_2, S_3)$                                   8
      foreach $e_x \in pairESI(e_j)$ do                                  9
        $newtc = < E_X; e_j; e_x >$;                                     10
        $T_{i+1} = Union(T_{i+1}, newtc)$;                               11
    if wasNeverExecuted[$e_j$]                                           12
      foreach $e_x \in pairEIG(e_j)$ do                                  13
        $newtc = < E_X; e_j; e_x >$;                                     14
        $T_{i+1} = Union(T_{i+1}, newtc)$;                               15
    wasNeverExecuted[$e_j$] = FALSE                                      16
  return $T_{i+1}$;                                                      17
}
```

Figure 5.3: The ALT Algorithm

one of the ESI predicates evaluates to `TRUE`, (3) $pairESI(e_i)$ that returns the set of all events that are ESI-related to e_i, (4) $pairEIG(e_i)$ that returns the set of all events that have an incoming edge from e_i in the EIG, (5) *Last*(*tc*) that returns the last event in test case *tc*, (6) $SubSequence(tc, first, last)$ returns a subsequence of *tc* starting at $first$ and ending at *last*, (7) *Length*(*tc*) returns the number of events in *tc*, and (8) *Union*(T_i, *tc*) adds *tc* to T_i avoiding duplicates. Also, an array `wasNeverExecuted`, indexed by each event, is set to `TRUE` if the event was disabled in the GUI's initial state S_0; otherwise it is set to `FALSE`.

The algorithm is shown in Figure 5.3. It takes the i-way test suite (T_i) as input and returns the (i+1)-way test suite. Each test case is broken into two parts (lines 3–4). If the first "*Length*(*testcase*) − 1" events (E_X) of the test case yield a state that is related via the ESI relationship (determined by the *isRelated* predicate), to its last event (e_j) (Line 8), then this test case is a good candidate for extension by a new event with all events to which it is ESI related (Lines 9–11). If the last event (e_j) has never been executed before but is made executable by E_X, then it is re-executed to compute new ESI relations (Lines 12–15). The output is the new i+1-way covering test suite.

The algorithm is invoked for T_2, which is obtained from the EIG. Each subsequent invocation with an i-way covering test suite (T_i) as input yields the (i+1)-way covering suite (T_{i+1}). Testing can be stopped once the testing goals have been met (or the testing team runs out of resources) or ALT returns an empty test suite. This can be if BOTH of the following happen:

1. no new ESI relations are found (*i.e.*, *isRelated*(S_0, S_1, S_2, S_3) returns `FALSE` on Line 8) or e_j is not ESI-related to any other event (*i.e.*, $pairESI(e_j)$ returns an empty set in Line 9).
2. e_j has already been executed in an earlier batch or was enabled in S_0.

This algorithm is fairly conservative in the number of test cases that it generates. Lines 8-9 provide a strict condition to test case extension, *i.e.*, not only must E_X by ESI-related to e_j, event e_j must also be ESI-related to at least one or more events, *i.e.*, $pairESI(e_j)$ returns a non-empty set. Moreover, it is observed in the experiments that most events have been executed by the second iteration of the algorithm; hence, Lines 12–15 are rarely executed beyond T_3. Because ALT is intended to be one of many algorithms that a tester should have in the "testing tool-box," having fewer test cases from ALT would help a test designer to conserve resources that may be redirected to other testing techniques, thereby yielding a "healthy" mix of test cases from several techniques.

One final point to note is the use of the function $FindState(S_0, E_i)$. This function maintains a lookup-table to return its output; the table is populated during test case execution; it is important that all entries exist. Entries corresponding to the three invocations of this function on Lines 5–7 are guaranteed to exist – for the invocation on Line 5, E_X was executed in a previous batch, for Line 6, e_j is a single event, whose resulting state was stored during the execution of the 2-way test cases, for Line 7, tc was executed in the current batch.

5.3 Empirical Study of ALT Approach

The test cases obtained from the ALT algorithm can be generated and executed automatically on the GUI. As done earlier in Section 4.2, the fully automatic "crash testing" was applied in this study.

5.3.1 Research Questions

This process provided a starting point for a feasibility study to evaluate the ALT test cases and compare them to the ESIG-generated test cases. The following questions needed to be answered to determine the usefulness of the overall feedback-directed process:

C5Q1: How many test cases does ALT generate? How does this number compare to the EIG- and ESIG-based approaches?

C5Q2: How many faults are detected by ALT? Of the faults detected in this study, which are detected by ALT and which by the ESIG-based approach? Why does one approach detect a particular fault whereas the other one misses it?

5.3.2 Process and Results

This study was conducted using the same four GUI-based OSS applications used in Study 1 described in Section 4.2. The fully-automatic crash testing process was executed on them and the cause (*i.e.*, the *fault*) of each crash in the source code was determined. More specifically, the following process was used for this study:

1. Choose software subjects with GUI front-ends.
2. Generate and execute the 2-way covering test suite.
3. Obtain the ESI relationships.
4. Generate new test suite using the algorithm of Figure 5.3. If the test suite is empty then stop; else execute it and report crashes. Repeat until ALT returns an empty test suite.

Step 1, 2 and 3 in the process have already been completed as part of the study reported in Section 4.2. Subsequent steps are described:

STEP 4: Execution of ALT algorithm. The initial set of ESI relations was used to obtain the 3-way test cases. The number of test cases is shown in Table 5.2. Note that the number for EIG and ESIG have been shown in Figure 4.4 in the logarithmic format. Here detailed numbers are given for clearer comparison. These test cases

Subject Application	i-way Suites				
	2	3	4	5	6
FreeMind	614	204	86	3	-
GanttProject	710	617	109	63	1
jEdit	591	419	54	38	-
OmegaT	469	310	11	-	-

Table 5.1: ESI relationships

were executed and the algorithm was invoked again. This process continued until ALT returned an empty test suite. Table 5.1 shows the number of ESI relations obtained *from* each of the i-way suites, for $i = 2 \dots 6$ (the data for 2-way test suites has been shown in Table 4.2). For example, only one ESI relation was obtained from the 6-way suite of GanttProject. A "-" indicates that there is no such an entry. As the numbers show, the ESI relations decrease with each iteration, thereby helping to terminate the ALT algorithm. This differed across applications: the experiment went as high as 7-way covering test cases for GanttProject and 4-way covering test cases for OmegaT. From these results, one can see that the total number of EIG-generated test cases is simply too large (so large that they are represented using the "exponent" notation to fit in the table). The 3-way ESIG-generated test suites are manageable; 4-way and beyond becomes quite large. The parenthesized ESIG entries are shown for comparison only – it was infeasible to execute such large numbers of test cases; the others were generated and executed. On the other hand, the ALT approach generates a reasonable number of test cases that goes down with each test suite iteration. This helps to answer **C5Q1**.

Both ALT and the ESIG-approach were successful at detecting faults in the applications, except OmegaT (only 2 faults were detected by the 2-way covering test

	i-way Suites				
	3	4	5	6	7
FreeMind					
EIG	$1.72e8$	$9.56e10$	$5.31e13$	$2.95e16$	$1.64e19$
ESIG	10,208	$(122, 426)$	$(1, 690, 861)$	$(21, 857, 767)$	(353,090,927)
ALT	10,208	2821	11	2	-
GanttProject					
EIG	$4.94e6$	$7.17e9$	$2.09e12$	$6.07e14$	$1.77e17$
ESIG	3070	14,742	27,933	$(63, 994)$	$(125, 362)$
ALT	3070	2229	226	34	4
jEdit					
EIG	$9.17e7$	$4.14e10$	$1.87e13$	$8.42e15$	$3.80e18$
ESIG	7572	84,488	$(1, 024, 424)$	$(10, 225, 602)$	(105,931,205)
ALT	7572	1258	738	171	-
OmegaT					
EIG	$7.65e6$	$1.51e9$	$2.97e11$	$5.85e13$	$1.15e16$
ESIG	2335	8935	42,859	$(219, 415)$	$(1, 135, 743)$
ALT	2335	1440	-	-	-

Table 5.2: Test Cases Generation

cases for this application). These results are shown in Table 5.3. Each detected fault is shown as a checkbox ☑, which is checked if the fault was detected; otherwise it is unchecked. To allow easy comparison, the checkbox vector is shown (for the same faults in the same order) for both ALT and ESIG. For example, faults 1, 2, and 3 in GanntProject were detected by both ESIG and ALT. Faults 4 and 5 were not detected by ESIG; they were however detected by ALT, fault 4 by the 3-way test suite and fault 5 by a 5-way covering suite. Clearly, ALT detected all the faults that ESIG detected and some more using many fewer test cases. This helps to partly answer **C5Q2**.

Now more details are provided for faults 4 and 5 of GanttProject, and Fault 3 of jEdit. These faults were not detected by the ESIG-based approach because several events required a complex chain of enabling events, which could only be

Subject Application	Technique	i-way test suite		
		3	4	5
FreeMind	ESIG	☑☑	-	-
	ALT	☑☑	-	-
GanttProject	ESIG	☑☑☑☐	-	☐
	ALT	☑☑☑☑	-	☑
jEdit	ESIG	☑☑	☐	-
	ALT	☑☑	☑	-
OmegaT	ESIG	-	-	-
	ALT	-	-	-

Table 5.3: Fault Detection

detected by alternating between test execution and generation.

Fault 4 in GanttProject results in a NumberFormatException. It is detected by a 3-way test case <e_1: *Create new task*; e_2: *Set general task property*; e_3: *Set non-integer value in task duration*>. Event e_3 causes GanttProject to crash because it expects an integer to be entered for the duration text-field in the task property window. However, if a non-integer value is set, GanttProject redraws the task shown in its schedule panel; the method `getLength()` invokes `Integer.parseInt(durationFi-eld1.getText().trim())` which throws a NumberFormatException.

In the GUI, event e_1 enables e_2, and the sequence $< e_1; e_2 >$ enables e_3. During ALT test case generation, none of the 2-way test cases that started with e_2 and e_3 executed; however, the test case $< e_1; e_2 >$ executed, indicating that e_1 enables e_2. Lines 12–14 of the algorithm used this information to extend all 2-way covering test cases that contained e_2 by appending the prefix e_1 to them; one important test case was $< e_1; e_2; e_3 >$.

In the first iteration of ALT, all 2-way covering test cases that started with e_3 remained unexecuted. Moreover, $< e_2; e_3 >$ was also unexecuted. Hence, by this iteration, ALT did not know how to execute e_3. In the second iteration, once the above-generated 3-way covering test case $< e_1; e_2; e_3 >$ was executed, it was used to determine that $< e_1; e_2 >$ enables e_3. Lines 12–14 used this information to obtain new test cases for the third iteration.

The above 3-way test case was the shortest and only sequence needed to reveal this fault starting in state S_0; none of the 2- and other 3-way test case could have detected it.

Fault 5 in GanttProject results in a NullPointerException. It is detected by a 5-way covering test case <e_1: *Create new task*; e_4: *Custom columns*; e_5: *Add columns (with a name)*; e_6: *Select newly created column in column table*; e_7: *Delete column*>. Once again, the enabling relationship is complex – e_1 enables e_4, $< e_1; e_4 >$ enabled e_5, $< e_1; e_4; e_5 >$ enables e_6 and e_7. One important thing to note is that it cannot be detected by any other 5-way or lower test case.

Fault 3 in jEdit results in a NullPointerException. It is detected by the 4-way covering test case < e_1: *Download QuickNotepad plugin*; e_2: *Select QuickNotepad plugin*; e_3: *Install QuickNotepad plugin*; e_4: *Choose QuickNotePad file*>. After installing the QuickNotepad plugin, jEdit allows the user to open a file by entering its path in a text-field. The user is free to enter any string in this text-field, including an incorrect path or the name of a non-existing file. Hence, when opening a non-existing file in QuickNodePad (e_4), the NullPointerException is thrown. In this test case, e_1 enables e_2, $< e_1; e_2 >$ enabled e_3; hence the $< e_1; e_2; e_3 >$ part of the test case was generated by Lines 12–14 of the ALT algorithm. Finally, $< e_1, e_2, e_3 > \xrightarrow{5(2)} e_4$; Lines 8–11 of the ALT algorithm add the event e_4. In this example, it is clear that the combination of the *enabling* and ESI parts of ALT was important to obtain the test case.

5.3.3 Discussion

This study demonstrated that ALT test cases are able to detect all the ESIG-detected faults, as well as some additional faults, using fewer test cases. Among the three faults that were discussed, one important thing to know is that the test cases that detected them were the shortest sequences needed to reveal the faults. Moreover, the ESIG-based approach could not detect them because of its inability to handle disabled events. An alternative algorithm, based on a random walk of the EIG, would have a very low probability of generating the fault-revealing test cases, for example, $\frac{1}{4.94e6}$ probability for Fault 4 of GanttProject. (Recall that the total number of 3-way sequences from the EIG is 4.94e6 for GanttProject.)

The event handlers in the fault-revealing test cases were distributed across multiple classes. For example, for GanttProject, e_1 was in the `NewTaskAction` class; $\{e_2, e_3, e_4\}$ were in `GanttDialogProperties`; $\{e_5, e_6, e_7\}$ were in `GanttTreeTable`. Similarly, for jEdit, e_1 was in the `PluginManager` class, $\{e_2, e_3\}$ in `PluginList`, and e_4 in `BeanShell`. As mentioned earlier, interactions across classes are difficult to infer statically; the run-time-state-based techniques are agnostic to how the event handlers are distributed.

5.4 Summary

This chapter presented the second feedback-directed test case generation approach that generates multi-way covering test cases by alternating generation and execution. It is based on analysis of the run-time state of GUI widgets obtained from a previous test batch to obtain a new batch; the process cycles through test case generation, execution, and analysis. The existing 2-way covering test cases are used as a starting point for GUI state collection. Subsequently-generated-and-executed test cases are used for the analysis, iteratively yielding additional test cases; no extra test cases are needed. The approach was demonstrated via an empirical study on four OSS applications. The results of the study showed that the test cases generated using the GUI state were useful at detecting serious faults in the applications; the alternating nature of the technique helped to detect complex enabling relationships between events.

Chapter 6

Exploration of Covering Array Sampling-Based Test Case Generation

As mentioned earlier, the graph-traversal-based test case generation algorithm `GenTestCases` was used for the EIG and ESIG-based approaches. The algorithm is straightforward; all possible paths of a specific length are generated from the graph model. As seen in Chapter 4 and Chapter 5, for the EIG model, it was practical to generate 2-way test cases. For the ESIG model, multi-way test cases up to length 5 were generated; longer test cases were impractical. To improve this situation, this chapter explores a new GUI test case generation technique that reduces the number of test cases, yet provides the desired t-way (*e.g.*, t=2, 3, 4, 5) coverage.

A naive approach to achieve this goal is to generate longer sequences of test cases by simply concatenating shorter sequences together. This will reduce the number of test cases (in the case of using length-nine sequences, one might reduce the number of test cases of a 3-way covering by one third), but it may cause other unintended problems. There are three primary technical challenges that must be overcome. First, as events execute differently in different states, the same event, e_i, found in two different locations in a sequence may behave differently depending on which sequence of events has previously been executed. If concatenating shorter sequences together, it is no longer guaranteed that all of the original t-way sequences behave independently; their behavior depends on their start state. The consequence is that not *all* t-way events from a given state may be tested. This will reduce the observed t-way coverage. Second, when combining sequences, it must be considered that GUI events have complex dependencies (*e.g.*, one enables/disables another) and strict structural constraints (*e.g.*, the event *Printer Properties* in the `Print` window can be executed only after the `Print` window is open), one cannot simply concatenate different sequences together to obtain a single executable test case. Third, one should not enforce a restriction on sequence lengths, such that they must be a multiple of the covering strength. For instance, if testing 3-way coverings, it should still be possible to test sequences of length seven or eight. Concatenation makes this impossible.

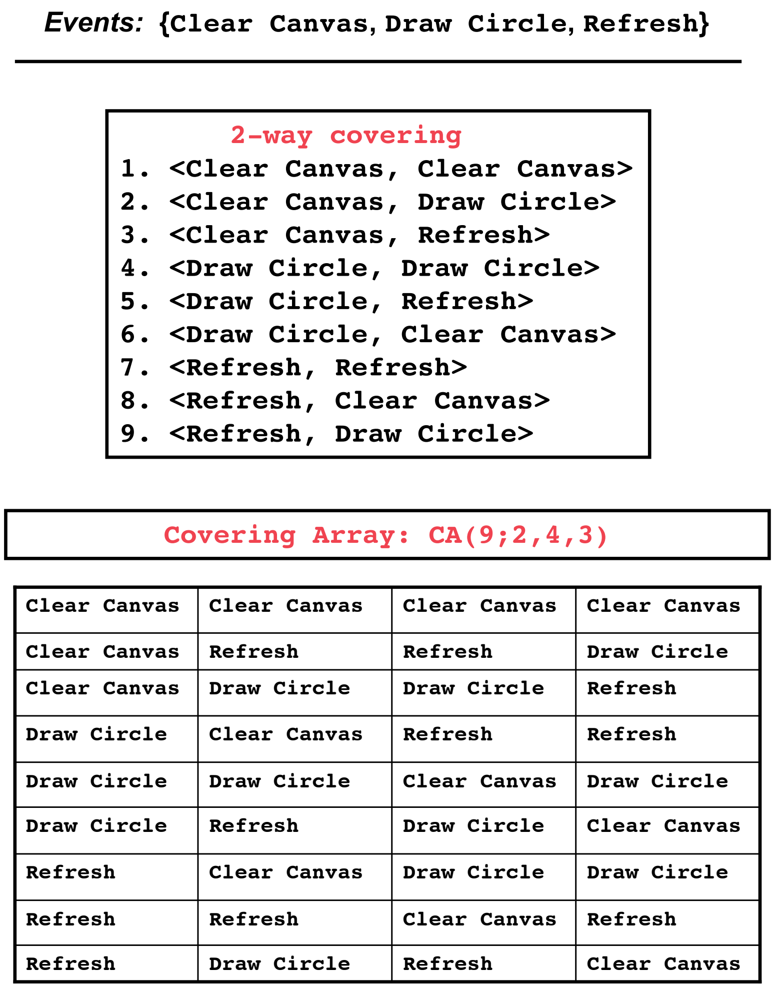
The ideas from combinatorial interaction testing [14, 15] are borrowed and *covering arrays* [15] are leveraged to develop a new automated technique for generating GUI test cases. The key motivation behind using covering arrays is to generate longer sequences that are systematically sampled at a particular coverage strength. The use of covering arrays (described in Section 6.1) solves the first and third technical challenges; the original t-way coverage is maintained, and any length sequence greater than t can be used. Furthermore additional coverage is gained by testing all t-way sequences from a variety of start states. Covering arrays ensure that a given 2-, 3-, 4-, or t-way relationship is maintained between GUI events in all possible combinations of t-locations in the sequence. This forces a balance in the sample of longer sequences. Traditionally, covering arrays have been used to minimize the number of test cases needed to test a software unit while maintaining the coverage of a given number of interactions between parameters or configurations [15]. They have not, to date, been effectively used for sequences which maintain state.

In order to use covering arrays effectively, and to avoid the issue raised by the second technical challenge, the EIG model is used to eliminate the need for ordering relationships between GUI events. This model, along with other inputs, is then used to obtain the covering arrays, which guarantee that given 2-, 3-, 4-, ..., t-way interaction relationships are maintained between GUI events. The rows of the arrays are mapped back to the GUI's original input space, in which ordering relationships are reinserted and used to generate test cases.

6.1 Background on Covering Arrays

A covering array (written as $CA(N; t, k, v)$) 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 [15]. In other words, any subset of t-columns of this array will contain all t-combinations of the symbols. This definition of a covering array is used to define the GUI event sequences. [1] To see how this can be applied to GUI event sequences, suppose the test sequence length is four and each location in this sequence can contain exactly one of three events (*Clear Canvas, Draw Circle, Refresh*) as is shown in Figure 6.1. Testing all combinations of these sequences requires 81 test cases. One can instead sample this system, including all

[1]A more general definition for a covering array, a *mixed level covering array* can be defined that does not use a single v, but instead allows each location in the array to have a different number of symbols. This type of array is not necessary for the problem, because there are the same number of events in each of the k positions.



Events: {`Clear Canvas`, `Draw Circle`, `Refresh`}

```
      2-way covering
1. <Clear Canvas, Clear Canvas>
2. <Clear Canvas, Draw Circle>
3. <Clear Canvas, Refresh>
4. <Draw Circle, Draw Circle>
5. <Draw Circle, Refresh>
6. <Draw Circle, Clear Canvas>
7. <Refresh, Refresh>
8. <Refresh, Clear Canvas>
9. <Refresh, Draw Circle>
```

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

Clear Canvas	Clear Canvas	Clear Canvas	Clear Canvas
Clear Canvas	Refresh	Refresh	Draw Circle
Clear Canvas	Draw Circle	Draw Circle	Refresh
Draw Circle	Clear Canvas	Refresh	Refresh
Draw Circle	Draw Circle	Clear Canvas	Draw Circle
Draw Circle	Refresh	Draw Circle	Clear Canvas
Refresh	Clear Canvas	Draw Circle	Draw Circle
Refresh	Refresh	Clear Canvas	Refresh
Refresh	Draw Circle	Refresh	Clear Canvas

Figure 6.1: 2-way Covering and Covering Array

sequences of shorter size, perhaps two. This sequence is modeled as a $CA(N; 2, 4, 3)$ (lower portion of Figure 6.1). The *strength* of given sample is determined by t. For instance, t is set to 2 in the example and all pairs of events between all four locations are included. If examining any two columns of the covering array, one can find all nine combinations of event sequences at least once. In this example there are 54 event sequences of length two which consider the sequence location. This can be compared with testing *only* the nine event sequences which would be used in the prior generation technique for a 2-cover (see top portion of Figure 6.1)

The number of test cases required for the t-way property is N. In the example, a $CA(9; 2, 4, 3)$ can be generated. Because the primary cost of running the test case is the setup cost, many more event sequences are covered for almost the same cost as the 2-way cover. In general there is no guarantee that the size of N will be the same as a shorter sequence, but it will grow logarithmically in k rather than exponentially as does the number of all possible sequences of length k [14].

Covering arrays have been used extensively to test input parameters of programs [10, 14, 18] as well as to test system configurations [31, 59]. Other uses of covering array sampling have been suggested, such as testing software product line families [16] and databases [13]. A special type of a covering array (an orthogonal array) developed from Latin squares has been previously used to define GUI test cases by White [53]. However, this work used covering arrays in a stateless manner to define subsets of the input parameter combinations. Bryce *et al.* used covering arrays to test a flight guidance system also modeled with state variables [11]. However, in this study only event sequences of length one were considered. In this technique, covering arrays are used to sample long event sequences, where events must consider state (determined by location in sequence and all prior events).

There are many tools and algorithms for constructing covering arrays. Because a covering array simply states that it must cover each t-set *at least* once, finding a minimal set of test sequences with the covering array property is an optimization problem. Given the constraints that all of the t-sets must be covered, the aim is to find the minimal number of sequences for which this property will hold. There are both mathematical constructions for covering arrays [23], as well as computational techniques that include both greedy [14] and meta-heuristic search [15]. For the purposes of the feasibility studies, one of the meta-heuristic search techniques, *simulated annealing*, is used because it provides the additional flexibility of feeding in “already covered” t-sets, building covering arrays of any size t, and generally (compared to other computational tools) produces covering arrays of comparable or smaller size.

6.2 Covering Array-Based Test Case Generation

The new technique explored in this chapter samples long event sequences from given event groups systematically and contains required t-way coverage. The event sequences are executed, and in the cases where this criterion fails, the feedback on the failed test sequences are obtained to help re-generate new sequences for testing. Unlike the feedback of GUI run-time information used earlier for ESI relationship identification, the feedback here is about the execution, such as the successfully executed partial event sequences and its event interaction coverage information. The technique uses a 7-step process; each of the steps is described next.

(1) *Generate GUI EIG.* This step is the same as in the two test case generation approaches presented earlier in Chapter 4 and Chapter 5. Details are not repeated here.

(2) *Partition GUI Events.* From the ESIG-based approach, one observation is that the ESIG may constitute multiple disjoint sub-graphs. Each sub-graph contains GUI events that are functionally related to each other. The studies also showed that the events in each sub-graph need to be tested more thoroughly (more faults were detected by longer test cases involving these events). Therefore, once the EIG has been generated, events are partitioned into functionally related units. The events within each unit constitute the events for a single model that is used to generate test sequences. Events that are not contained within the same group are not tested together. This part of the process is currently done manually from domain knowledge that can be used to determine which events are likely to be included in similar functionality. In future work, this process may be automated through the use of historical data on similar domains or ESI relationships. The output is a model that lists the specific event groups as well as the number of events per group.

(3) *Identify Constraints.* This step creates the abstract event model. Once the event graphs and groups have been identified, constraints are specified on events such that the generated event sequences are executable. This is necessary because some events may not be executable without a set of prior set-up events or must occur only after another event has been fired. The abstract constraint model creates aggregate events for these. For instance, from the example events in Figure 6.1, the *Draw Circle* event may require that an event *Set Ink Color* occurs first. Although this event may not have been of interest in the original EIG graph, it is needed to reach the event *Draw Circle* and the new aggregate event has these two events concatenated together. This is retained as a single abstract event *Draw Circle* in the model. The output of this phase is the full set of aggregate events which can be expanded in later steps into test sequences. This ends the modeling stage of the test case generation.

(4) *Generate Covering Array Sample.* There are four inputs to this step of the process. The first is k, which determines the length of the abstract sequences (*i.e.*, those that may need the insertion of other events to become executable). The second is the number of abstract events per location, v, with a list of the v abstract events that are to be tested. The third is the strength of the desired covering array, t. Finally, a set of already covered interaction t-sets is optionally passed as an input. The need for this optional parameter is explained later. Using these parameters a covering array is generated using one of the known covering array generation algorithms. The covering array contains N rows of abstract events. These are passed to the next phase for translation into executable test cases. The bottom portion of Figure 6.1 is an example of the output for this stage.

(5) *Generate Executable Test Cases.* The input to this step is the covering array. The abstract event sequences from step **(3)** are expanded in this step to generate executable test cases. Returning to the example, the second row of the covering array has 4 abstract events. In order to execute this, the last event *Draw Circle* must be expanded to make these events into executable calls. The executable test vector is now <*Clear Canvas*; *Refresh*; *Refresh*; *Set Ink Color*; *Draw Circle*>. Furthermore, the test vector are transfered to actual calls to given program by translating them into a scripting language for the replay tool used to execute test case. The output of this step is a set of actual test cases.

(6) *Execute Test Cases.* The test cases are executed. During this stage data is collected via the automated oracle to determine which test cases detect faults. These are sent to output as an error report. Unexecutable test cases (those that fail to execute successfully because they specify an invalid event sequence) are also recorded. The location in the sequence is recorded as well. The output of this stage is an error report, as well as a list of which test cases failed to execute to completion. In the scenario described above, if the last test case fails after *Draw Circle* , it is recorded as a *fail* and the failing location would be 4.

(7) *Determine Missing Coverage.* The input to this step is the set of test cases and the last successfully executed location in each test case. This data is analyzed to produce a list of *covered* t-sets. The algorithm that performs this task enumerates all of the covered t-sets in each row using the last passed location as its stopping point.

The output of this stage is fed back into the covering array generation (Step **(4)**) as the last optional parameter, mentioned above. This parameter is used to re-generate a covering array sample that covers the previously untested combinations. For instance, if during the execution of the covering array shown in Figure 6.1, the last test sequence fails to execute after the second event, all pairs of events tested by all sequences prior to this are covered, as well as the pair of events *Refresh, Draw Circle* in location 1, 2. There are four other event combinations that were not tested in this test (*i.e.*, *Refresh, Refresh* in location 1, 3, *Refresh, Clear Canvas* in location 1, 4, etc.) Steps **(5)**, **(6)**, and **(7)** are then repeated for the new test cases. This iterative process continues as many times as is needed to either complete the desired coverage, or until no more faults are found.

The iterative process is also summarized in Figure 6.2. There are seven primary steps in this process, four of which occur inside of the feedback loop. Given the automated test process and complex interactions between events, it is not always

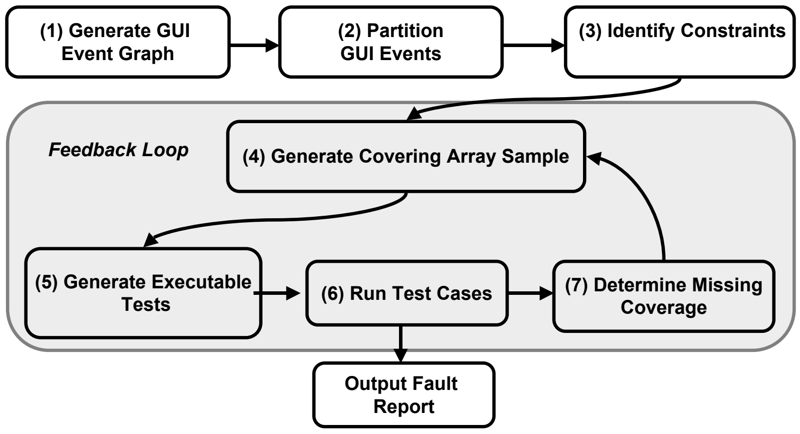

(1) Generate GUI Event Graph
(2) Partition GUI Events
(3) Identify Constraints
Feedback Loop
(4) Generate Covering Array Sample
(5) Generate Executable Tests
(6) Run Test Cases
(7) Determine Missing Coverage
Output Fault Report

Figure 6.2: Test Generation Process Using Covering Array Sampling

possible to predict these failures ahead of time. Therefore, after executing test cases, additional test sequences are needed to be re-generated to satisfy the coverage decided upon in earlier steps of this process.

6.3 Feasibility Study of the Covering Array-Based Approach

A feasibility study was conducted to determine the effectiveness of covering array sampling for GUI testing on faults that can only be detected by long event sequences. This is compared with the previous EIG-based technique, *i.e.*, testing *all possible* 2-way coverings for shorter sequences. Effectiveness of the test suites is measured both in terms of the cost to generate and to run, as well as through fault-detection effectiveness.

6.3.1 Research Questions

More specifically, the study was designed to address the following two research questions:

C6Q1: How does the fault-detection effectiveness of t-way covering array sampling compare with that of the *same* and *stronger* t-way coverage on shorter sequences?

C6Q2: Is there a cost difference in generating and running covering array samples for long sequences, and generating and running a larger number of shorter t-way sequences?

6.3.2 Study Subject

For this study, Paint in TerpOffice suite was used. Note that it was also used in Study 2 for the ESIG-based approach (described in Section 4.3). Moreover, the same type of test oracle was used in this study, together with the fault-seeded versions of Paint.

Because this study is concerned primarily with the fault-detection effectiveness of event sequences *longer than two* in order to detect faults that can only be detected by longer event sequences, only the seeded faults which were not detected by the existing 2-way covering EIG test cases were used in this study. Of the original 263 seeded faults, 115 fall into this category.

6.3.3 Process

All the components in the algorithm addressed in Section 6.2 have been implemented in an automated testing procedure. Next, the details for each step of the process that are specific to the study are described.

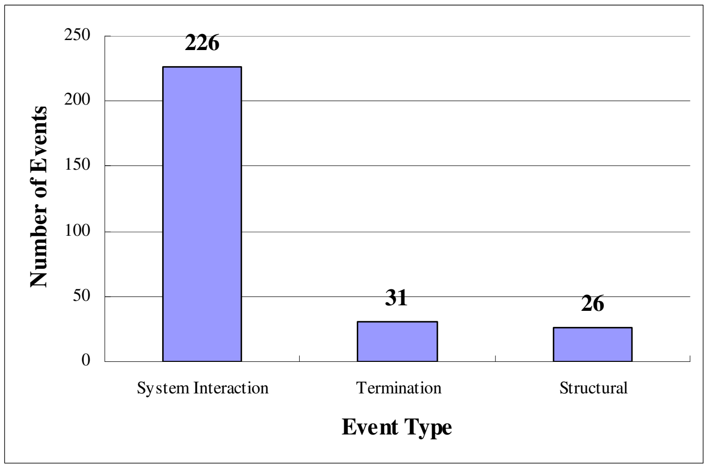


Figure 6.3: Classification of GUI Events in Paint

STEP 1: Create EIG model. The previously obtained EIG model for Paint was used here. Recall that the EIG contains *system-interaction* events (those that interact with the underlying system rather than manipulate the GUI's structure) and *termination* events (those that close windows) [40]. Other *structural* events, such as those used to open windows and open menus, are omitted from the EIG to improve efficiency. In order to avoid ordering relationships between events due to GUI structure, only system-interaction events were used for covering arrays. Some of the other events may be inserted back into the final event sequences in **STEP 5** if they are needed to reach particular system-interaction events.

Figure 6.3 shows the number of the different event types. One can see that a large number (226) of the GUI events fall under the system-interaction category. All of these events are used for test case generation.

STEP 2: Partition the GUI events. This step was done manually and took two hours to complete. The partitions for the system-interaction events are shown in Figure 6.4. The six partitions represent events related to different functionality. The

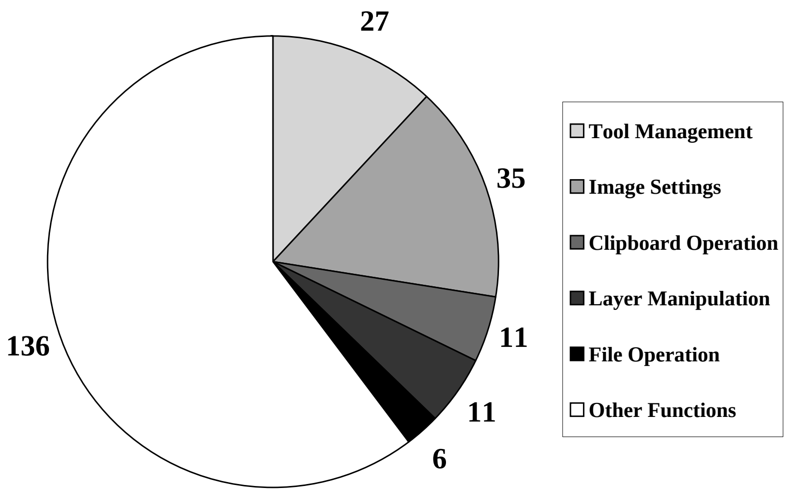


Figure 6.4: Partition of System-Interaction Events in Paint

numbers of events in each of the first five groups are also shown in Table 6.1. The six groups identified are as follows: (1) Group 1: *Tool management.* This includes events such as selecting the *Line Tool* and dragging the mouse on the canvas; (2) Group 2: *Image settings.* The events in this group make up the events that affect the size and style of images such as *Choose Background Color* and *Stretch the Image to Size*; (3) Group 3: *Clipboard operation.* This includes events such as *Copy* and *Paste*; (4) Group 4: *Layer manipulation.* This includes events such as *Add a New Layer* and *Move to Front Layer*; (5) Group 5: *File operation*, which includes events related to the filesystem such as *Open File* and *Save Current Image File*; and (6) Group 6: *Other functionality.* Events in this partition are relatively independent of each other. They include events to scroll the canvas, or events for which functionality has already been represented by other system-interaction events in other partitions. For example, in window `Open`, there are many system-interaction events used to navigate to the file that will be opened, such as *Choose Directory* and *Up one Level.*

This can also be done using the *Input File Name to text-field* event. When the text-field event is given a complete file name together with its absolute path, there is no need to perform other events related to file searching in this window.

STEP 3: Identify constraints. In this step, the events from each group were examined, and the additional sequences of events needed to make the longer sequences executable were determined. For example, in GROUP 1, the event *Select All* is performed by clicking the corresponding menu item. However, the menu item will not be ready for clicking until it has first been opened in the `Edit` menu. Another example is seen in GROUP 2. The events to set image properties are in different windows. The path between these events, which includes events to close the first window and events to open the second window, must be added to the final event sequences to make test cases executable.

STEP 4: Generate long test cases using covering array sampling. The study focuses on the first five groups, because events within each group are functionally related. Covering array samples are constructed for each of these groups independently. Different coverage criteria have been chosen for each group based on the number of events. Through previous studies, it has been determined that abstract sequences of length 10 are about the maximum length sequences that run without failure. Therefore, the sequence length has been fixed at ten for this study. Each of the 10 locations in the event sequence can contain any one of the events from this group. This follows the model shown in Figure 6.1. The strength (t) for each covering array is determined using a heuristic that generates the highest covering array strength, for each group, that is complete in a single overnight test run. This number is approximately 20,000. For instance, in Group 1, there are 27 unique events; therefore, the highest sampling strength would be two because a covering array built for t=3 will exceed the 20,000 test cut-off. But in Group 5, there are only six events, so covering arrays of strength t=4 are able to be generated.

Table 6.1 provides data on the covering array sampling for each group. The #Events row shows the total number of events in each group, the Event Space row, shows the number of sequences that would be required to cover all 10-way combinations of the group. The CA row gives the covering array definition that was used for each group and the #Test Cases row provides the number of test cases in the final covering array sample generated. The covering arrays were created using a simulated annealing program [15]. The user CPU generation times on a Dual 2.7 GHz PowerPC G5 and 1.5 GB of RAM running Mac OS X, are shown in Table 6.3 as Build CA. The times vary from .73 hours to 6.6 hours. Group 3 and 4 have the same covering array parameters, therefore only a single covering array was generated and mapped to each of the appropriate groups. For this reason, the time for Group 4 is shown in parentheses.

Groups	Description	#Events	Event Space	CA	#Test Cases (N)
1	Tool Mgt.	27	27^{10}	$CA(N; 2, 10, 27)$	1055
2	Image Settings	35	35^{10}	$CA(N; 2, 10, 35)$	1783
3	Clipboard Ops.	11	11^{10}	$CA(N; 3, 10, 11)$	2870
4	Layer Manip.	11	11^{10}	$CA(N; 3, 10, 11)$	2870
5	File Ops.	6	6^{10}	$CA(N; 4, 10, 6)$	3428

Table 6.1: Test Cases Generated by Covering Array Algorithm

STEP 5: Generate executable event sequences. Each test case is mapped back to executable event sequences through the abstract event model identified in step three. An example of generating test cases from GROUP 1 follows. One possible test sequence from the covering array is the vector of abstract events: *<81;98;102;260;231;235;81;103;229;102>*. The behavior of Paint while executing this test case follows. First it reverts the previous action by clicking the *Undo* (event #81). Then, it enlarges the size by performing the *Large* (event #98) and sets the *Draw Opaque* (event #102) on the image. Next, it selects `Northeast` (event #260) in the `Emboss` window. To reach this event, corresponding open window events are added before it. It rotates the image *270 degrees* (event #231). Note, this event is in the `Flip/Rotate` window. To reach it, termination events to close the `Emboss` window and open window events for `Flip/Rotate` window must be inserted before this event. After that, it *Stretches* the image to a certain size (event #235). Again, this event is in a different `Stretch/Skew` window and necessary reaching events need are inserted. Finally, it *Cancels* the last operation (event #81), sets the image *Brightness* (event #103), rotates the image by *90 degrees* (event #229) and sets the *Draw Opaque* (event #102) image again. The final length of the executable sequence is 21.

In summary, the necessary non-system interaction events and termination events which are used to reach the first events in each test case or which reside on the path between two system-interaction events in the test case are added to transform the generated test cases to executable event sequences. The actual test case execution file is an XML file containing the events and required parameters. Table 6.2 shows the distribution of the *actual* event numbers in the generated long test cases. As can be seen, all of the test cases in GROUP 1 have actual event lengths

Actual Length	1	2	3	4	5
10-19	1055	0	2739	2851	0
20-29	0	335	131	19	264
30-39	0	1405	0	0	2637
$\geq$ 40	0	43	0	0	527

Table 6.2: Event Sequence Length Distribution

less than 20, while Group 5 has over 500 sequences that are longer than 40 events in length.

STEP 6: Execute test sequences. The test cases are executed on the original version of Paint. The validity of test cases are identified and the execution time is recorded. Table 6.3 shows number of successfully executed test cases for each group and their execution time in hours. The time is broken down into the time to obtain oracle information and the time spent in verification. To gather oracles for the experiments, all test cases are run on the original, non-faulty version of Paint. This collects the expected GUI run-time states as each test executes.

Groups	1	2	3	4	5	Tot
#Success	1055	1774	2010	1321	13	6173
Percent t-way Cov.	100	99.9	88.7	67.3	1.8	NA
Build CA (h)	.73	1.2	2.9	(2.9)	6.6	14.3
Oracle Collect(h)	3.3	12.4	16.3	8.5	9.6	50.2
Verific.(h)	52.4	530.5	225.4	44.3	1.8	854.4

Table 6.3: Test Case Execution

The verification phases takes longer to finish than the oracle phase. This is because test cases are run on multiple versions of the program. It has been determined through code coverage analysis during the oracle run which test sequences potentially traverse more than one fault. For these test cases, they were ran on multiple versions of the program, each containing only one of the potential faults.

This prevents interactions that might cause missing a fault, however, it requires additional execution time.

A slight further time degradation is seen in the verification stage, because the tool used to run GUI test cases in **GUITAR** is configured so that if the execution of a test case is stuck, the executer waits for the maximum allowable execution time before exiting. This time is proportional to the actual number of events (Table 6.2), which may exceed the abstract test length of 10. The test cases were executed on the 50-machine cluster used in previous studies.

STEP 7: Regenerate test cases to cover missing event interactions. In the first two groups almost all test cases executed successfully to completion, *i.e.*, nearly 100 percent coverage modeled by the covering array is achieved. In the last three groups, however, there are failed test sequences that required a return to **STEP 4** in the feedback loop (Generate Covering Array Sample). For groups three and four, covering arrays were re-generated by identifying which of the 3-sets were already tested These were passed back into the covering array algorithm as an optional argument. This was not applied to the second group during the feasibility study because only 9 test cases failed providing almost 100 percent of interaction coverage. In a real test scenario, one would, however, re-generate to be sure that there are no missed interaction t-sets. In the last group, Group Five, one can see that there was a very strict ordering requirement between GUI events. This may be related to the large number of additional events needed for reachability. In this set of test sequences, only a small set of the test cases ran to completion. It can be seen that only 1.8% of the 4-way coverage was achieved. Because the probability of

Group	3	4	Total
#Regenerate Test Cases	881	1536	2417
#Success	298	222	520
Percent Accum. t-way Cov.	96.1	77.7	NA
Build CA (h)	.99	2.9	3.9
Oracle Collection (h)	6.3	3.8	10.1
Verification(h)	28.1	5.5	33.6

Table 6.4: Regenerated Test Cases

generating new valid test cases was very low, no more sequences were re-generated for this partition. Future work may involve further constraints and better abstract models to avoid these types of problems.

Table 6.4 shows the lengths of the sequences and achieved accumulated coverage after re-generating and running more sequences as well as the additional run times in hours. In both of these cases, it is unable to achieve 100 percent coverage after 2 iterations of testing. Because the second set of test sequences in Group 3 and 4 did not uncover any new faults, the feedback loop and re-generate of test sequences terminated.

Control Group: As a control group, another test suite $T(orig)$ that uses the EIG algorithm to generate longer test cases of complete t-way coverage was run. It includes two separate test suites $T(same)$ and $T(stronger)$. $T(same)$ includes test cases with the corresponding t-way coverage for each group as the $T(cov)$. For $T(stronger)$, due to the enormous number of test cases that is generated for each group in full t-way coverage, test cases that were only one-way longer than the coverage criteria used in the covering array sampling generation was generated. For instance, in Group 2, $T(stronger)$ is a 3-cover (all 3-sequences) and for Group 3

Groups	1	2	3	4	5
Length (t-way)	3	3	4	4	5
#Test Cases	19,683	42,875	14,641	14,641	7776
#Success	19,682	42,806	13,157	11,321	281
Percent t-way Cov.	99.9	99.8	89.9	77.3	3.6
Oracle Collect(h)	39.1	136.6	46.5	32.7	31.3
Total Hours in Oracle Collection = 348.4					
Verific.(h)	253.3	1911.4	425.4	142.1	25.9
Total Hours in Verification = 2758.0					

Table 6.5: $T(stronger)$ Test Case Execution Time

$T(stronger)$ is a 4-cover. Table 6.5 shows the length of the $T(stronger)$ test cases for each group as well as the number of test cases.

6.3.4 Results

This section discusses the results in finding faults using the covering array sampling process as they relate to the two research questions.

C6Q1. Table 6.6 shows the fault-detection effectiveness for the covering arrays, sequences, $T(cov)$, using both the original arrays ($T1$) and regenerated arrays ($T2$), vs. two sets of $T(orig)$ test sequences, $T(same)$ and $T(stronger)$. Fault-detection effectiveness is measured as the number of unique faults detected divided by the total number (115). One can see that the number of faults detected by $T(cov)$ is much higher than that detected by the $T(stronger)$. The fault-detection effectiveness increased by 17%. For all groups, the covering array-based test cases detected the same or more faults using a much smaller number of test cases. $T(same)$ had a very poor fault-detection effectiveness so it is not included in further analysis. Figure 6.5 shows the cumulative fault coverage by test case for $T(cov)$ vs. $T(stronger)$.

The x-axis shows the number of the test case, while the y-axis shows the number of new faults detected by each successive test case. Although the default order of the covering arrays (or generated $T(stronger)$) arrays was used, this data seems to be compelling. Even if the $T(stronger)$ test cases are re-ordered to improve their early fault detection, in order to obtain complete coverage, 99,616 test cases (vs.14,423) must be run. At the end of this time, there are 20 faults that $T(orig)$ did not uncover. On the other hand, the covering array test cases, detect the same number of faults as $T(orig)$ after only 145 test cases and reach their maximum fault finding after only 9,570 test cases.

Figure 6.6 shows similar data but only for Group 1 because this is the group that uncovers the most faults. Figure 6.7 shows the *density* of test cases that are able to detect each individual fault in Group 1. The x-axis lists each of the 69 faults that are found by the covering array sample. The y-axis is the count of test cases that find the specific fault. Although there are several cases where $T(stronger)$ (lighter bar) shows a higher density of test cases detecting a particular fault, this is not the case for the majority of faults even though $T(stronger)$ has more than 18 times the number of test cases.

Test Suite	$T(orig)$		$T(cov)$	
	$T(same)$	$T(stronger)$	$T1$	$T2$
Group 1	0	49	69	NA
Group 2	0	4	4	NA
Group 3	6	8	8	8
Group 4	0	0	0	0
Group 5	0	5	5	NA
Total(unique)	6	62	82	
% of Detection	5.22%	53.91%	71.3%	

Table 6.6: Fault-Detection Effectiveness

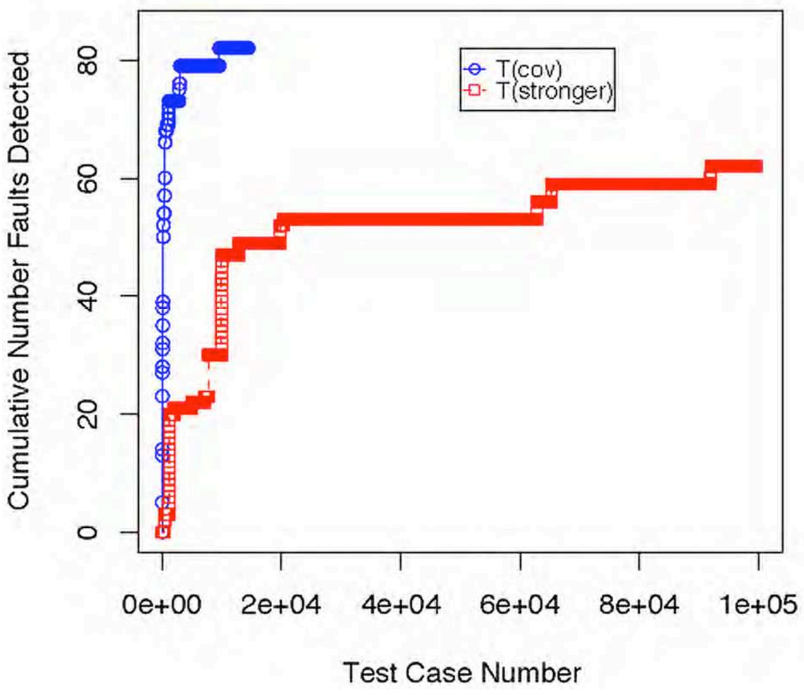


Figure 6.5: Total Cumulative Fault Coverage

C6Q 2. Figures 6.5 and 6.6 show the additional number of test cases that must be run for $T(stronger)$. However this isn't a direct measure of test execution time. The execution time tables (Tables 6.3, 6.4, 6.5) which show execution time in hours clearly indicate that the execution of the covering array test cases saves considerable time. Although individual test cases in $T(cov)$ contain more events (*i.e.*, they are longer), the overriding factor is the size of the test suite. For each test case, time must be spent starting up the Java Virtual Machine, initializing states, etc. The execution times are compared for collecting the oracles, because in this case the program is run a single time. The total time across all covering arrays (including the re-generation steps) is 78.5 hours while the time to run $T(stronger)$ with no re-generation took 348.4 hours, more than four times as long. In days, the covering

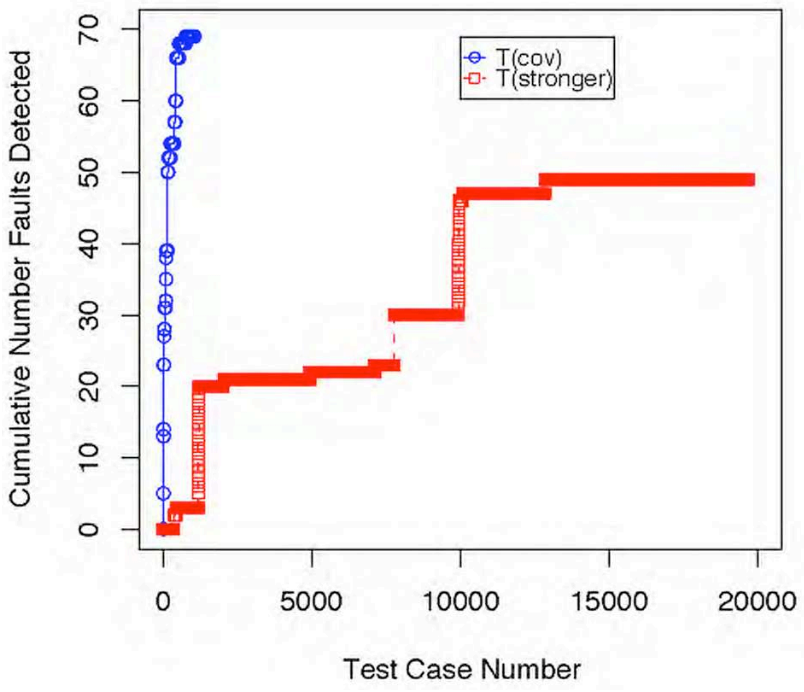


Figure 6.6: Cumulative Fault Coverage in Group 1

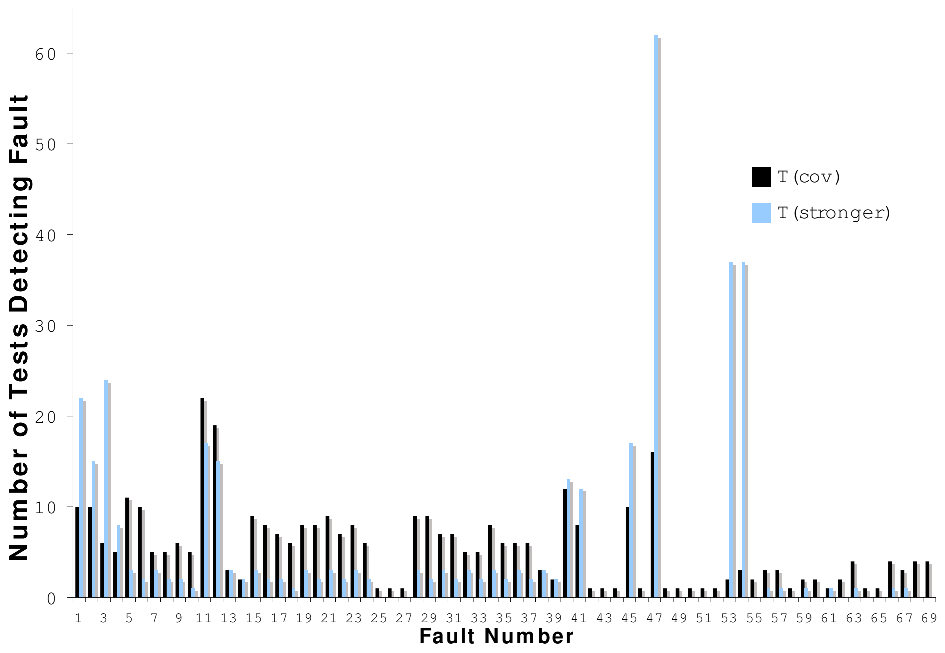


Figure 6.7: Density of Test Cases Detecting Found Faults for Group 1

array test execution took just under 3.5 days, while $T(stronger)$ took over 14 days to run. There is some non-trivial time required to construct the covering arrays. These times are shown in Tables 6.3 and 6.4. The overall generation (and re-generation) time for the covering arrays required 18.2 user CPU hours hours although 2.9 hours of this is "artificial" because only a single covering array requiring 2.9 hours of CPU user time was generated and used for both Group 3 and 4. The time required to manually partition the events into groups (2 hours in this study) must be done for both the $T(cov)$ and $T(orig)$ methods.

6.3.5 Discussion

The results obtained in the feasibility study are encouraging. They show that using t-way covering array sampling on longer sequences of events improves fault detection over shorter $(t+1)$-way coverings. In addition there is a large cost savings in terms of reducing the cost of test execution. In this section some insights that are obtained from this feasibility study are discussed, which provide direction for making this testing method applicable to a wider range of GUI applications.

Fault Finding Analysis: To understand why the longer covering array-based test cases detected more faults, faults that were detected only by the covering array samples were analyzed. One set of faults (86, 87, 88) were detected by several test cases in the covering array sample but never detected by the $T(orig)$ test suite. These faults are all found in the handler for the event *Select Eraser Tool* which corresponds to the method `eraserActionPerformed`. The faults incorrectly change an “==” to an “!=” in three different conditions in this handler to check `curZoom`, a property that decides what type of cursor is to be used for the eraser tool. If `curZoom` == `zoom2`, for example, the eraser cursor's size will be set to one size, but when `curZoom` == `zoom1`, it will be set to a different size. When the condition results are incorrectly returned, and the eraser tool is used, different results will occur. One of the test cases from the covering array sample in GROUP 1 which detects this fault contains the following *abstract* event sequence: <*Text Tool, Line Tool, Fill with Color, Select Double Zoom, Eraser Tool, Fill with Color, Line Tool, Move Mouse in Canvas, Show Tool Bar, Ellipse Tool*> Because no 3-way covering test cases from $T(stronger)$ in GROUP 1 detected these faults, detection requires more than three events. First the test case must reach the faulty statements in the code. Two *actual* events are needed for this: 1) *Select a Zoom tool* (There are four possibilities which correspond to `zoom1`, `zoom2`, `zoom3`, `zoom4` for `curZoom` in the code.); 2) *Select Eraser Tool*. The order of these two events can not be changed. Simply reaching this code is not enough for detection, however. The faulty behavior needs to show itself in the GUI for detection. In this case, an image is needed where the *Eraser Tool* can be applied, and the wrong eraser will wipe out a different part of the image. By checking the resulting image on the canvas, one can detect the fault. Here, at least two more events are needed: one for setting up an image and the other for using the eraser tool (*Fill with Color* and *Move Mouse in Canvas* in the test case). Therefore, the shortest sequence that can detect this fault would be a length-four sequence. In the experiments, the GUI is started with no image (a white canvas). In the detecting test case the *Fill with Color* event fills the empty canvas with the default color, which is black. After selecting the *Eraser Tool*, it moves the mouse on the canvas with the eraser tool, and a black area will be removed (turned white). As the type (therefore the size) of eraser is incorrectly set by the fault, the resulting image is different from the expected one and the test case detects the fault.

Unexecutable Sequences: It has been seen that some of the test suites did not run to completion. For this reason, the feedback loop to allow test re-generation was added. However, for some groups, there are simply too many ordering constraints to make even re-generation feasible. GROUP 5 is an example of this. In both types of test suites more than 96% of the test cases failed to execute to completion. One need is to develop better methods of detecting and defining temporal type constraints which can then be enforced when building the covering array sequences. Recent work on incorporating constraints into constructing covering array samples may be useful for making this step feasible [17].

Cost of the Process: In the current feasibility study, a simulated annealing algorithm was used to construct the test cases. This algorithm often takes longer than other types of construction methods, although it often produces smaller covering arrays [15]. A cost-benefit equation to determine which algorithm is to be used for covering array construction should be added to the process. Sometimes the time to generate the covering arrays may be tolerated in the overall testing time, but other times this may be a bottleneck and faster generation techniques that create more test sequences may be desired. In the feasibility study the covering array construction time added a 30% overhead. Strategies that may help to reduce these costs include parallelism in covering array construction and maintenance of repositories of covering arrays for the common GUI parameters. For instance, in this study, two groups have identical parameters and therefore the (real) cost was reduced by almost three hours. Another cost savings may be found in the manual partitioning of groups (**STEP 2**). This may not scale as programs get larger. This process may be automated using ESI relationship identification used in ESIG-based approach to keep the cost of the partitioning step reasonable in future work.

6.4 Summary

This chapter explored a new technique to generate longer GUI test cases than previously generated from graph-traversal-based approaches that guarantee certain event interaction coverage. The technique uses covering arrays to generate long event sequences that are systematically sampled at a particular coverage strength. This technique is novel in its use of covering arrays to sample sequences for state-based, event-driven systems, an abstraction of the GUI that enables efficient use of covering arrays. The feasibility study demonstrated that the new technique greatly reduced the number of test cases. At the same time, it was able to detect faults that were previously undetected by shorter test cases.

Chapter 7

Conclusions and Broader Impacts

This research developed a new automated, feedback-directed model-based GUI test case generation technique. In this technique, the novel concept of *event semantic interaction* (ESI) relationships was introduced, and identification of the ESI relationships among GUI events was formalized using predicate evaluation on GUI execution feedback (GUI run-time state). The ESI relationships were used to direct GUI multi-way test case generation. Two feedback-directed test case generation approaches were developed.

In the ESIG-based approach, ESI relationships are computed from EIG 2-way covering test cases. The ESIG is created by annotating EIG edges with these ESI relationships, and a graph-traversal algorithm is used to obtain multi-way test cases involving only ESI-related GUI events. The ALT approach alternates test case generation and execution. As in the ESIG-based approach, test case generation begins with a seed test suite of EIG 2-way covering test cases. New test cases are generated in batches, where in each batch 1) more ESI relationships (*i.e.*, between event sequences and events) are computed from test cases generated in the previous batch; 2) the new ESI relationships are used to generate one-step-longer test cases; 3) new enable-disable relationships among events are identified and used to generate test cases involving previously disabled and unexecutable events.

Empirical studies demonstrated that test cases generated from both of the feedback-directed test case generation approaches significantly improved the fault-detection effectiveness of GUI testing. These approaches were able to detect previously undetected faults with reasonable cost in terms of number of test cases.

Finally, a covering array-based test case generation technique was explored to generate test cases. It generates long test cases that systematically sample the input space with a particular event interaction coverage strength. A feasibility study shows that the covering array test cases improve fault-detection effectiveness with small number of longer test cases compared to shorter ones.

As is the case with all empirical studies, the studies in this research are subject to threats to validity. First is the selection of subject applications and their characteristics. Eight Java applications in two sets have been used in the studies. Although they have different types of GUIs, this does not reflect the wide spectrum of possible GUIs that are available today. Moreover, the applications are extremely GUI-intensive, *i.e.*, most of the code is written for the GUI. The results may vary for applications that have a complex back-end, are not developed using the object-oriented paradigm, or have non-deterministic behavior. Second, the use of the test pool approach in Study 2 of the ESIG-based approach (described in Section 4.3) was due to practical limitations. It is expected that the repetition of the same test case across multiple test suites will have an impact on some of the results. Third, the algorithms used to create test cases ensures that each event (the first event in the test case) is executed in a known initial state; the choice of this state may have an effect on the results.

Fourth, the Java API allow the extraction of only 12 properties of each widget; only these properties were used for obtaining the ESI relationship via GUI state; widgets may have additional properties that are not exposed by the API. Hence, the set of ESI relationships may be incomplete. Fifth, two approaches were used to generate test cases – ESIG-based graph-traversal approach and ALT. Other techniques (*e.g.*, using capture/replay tools and programming the test cases manually) may produce different types of test cases, which may show different execution behavior. Sixth, a threat is related to the measurement of fault-detection effectiveness in Study 2 described in Section 4.3; each fault was seeded and activated individually. Note that multiple faults seeded and activated simultaneously can lead to more complex scenarios that include fault masking, thereby affecting the measurement of fault-detection effectiveness.

7.1 Contributions

The feedback-directed GUI test case generation technique makes the following contributions:

- The idea of using feedback-directed test case generation for GUI testing was presented.
- The ESI relationship was formalized using predicates on GUI run-time state.
- Two automated GUI test case generation approaches were developed using the feedback-directed technique.

- A covering array-based test case generation technique was explored.

7.2 Broader Impacts

This research has presented several exciting opportunities for future research. The following research directions may be explored:

Better understand of the ESI relationship. The studies in this research showed that ESI relationships helped to generate test cases that improved fault-detection effectiveness in GUI testing. It is hypothesized that this improvement is caused by the linking of events that, in some sense, are functionally related; executing them together causes the revelation of problems due to shared objects. Therefore, as discussed in Section 5.1, a better understanding of the subtle nature of the ESI relationship may further improve test case generation.

Classify GUI events. Study 1 in Chapter 4 showed that certain events in the GUIs dominate the ESI relationship. These events may need further study and classification to help identify the core part of the GUI's functionality and group events that may need to be tested with more complex test cases.

Simplify ESI identification predicates. Four contexts were identified, each with twelve cases of ESI relationships. In the future, these contexts and cases may be simplified and, if possible, combined. The current special treatment of termination events, which led to an additional three contexts, may be revised. One possibility is the revision of the EIG model; the elimination of all termination events from this model may be explored. This revision may also lead to the definition of new, fundamentally different cases for the ESI relationship.

Refine ESI identification. Several events are ESI-related because of multiple predicates. Currently predicates are not "counted" for each relation; in the future, "strengths" to ESI relations may be assigned based on how many predicates are `TRUE` for each pair of events.

Enrich execution feedback. The feedback currently obtained at run time is in the form of GUI widgets. Mechanisms like reflection in modern programming languages may be used to obtain additional feedback from non-GUI objects. The definition of state, in terms of a set of objects with properties and values, is general; it may be applied to any executing object. Some of the twelve cases may be adapted for non-GUI objects. The run-time state information is currently collected using the Java Swing API for standard Swing widgets. Future work may incorporate a customized API for application-specific widgets into feedback collection and analysis. Another straightforward way to enhance the feedback is to instrument the software for code coverage and run-time invariant collection. This feedback may be used to generate new types of test cases.

Combine dynamic analysis and static analysis. The analysis summarized in Section 4.2 led to a deeper understanding of the relationship between real GUI events and the underlying code in fielded GUI applications. This may lead to new techniques that combine dynamic analysis of the GUI and static analysis of the event handler code. For example, the code for related events may be given to a static-analysis engine that could examine the code for possible interactions that are only apparent at the code level, *e.g.*, data-flow relationships.

Apply the feedback-directed technique to other applications. Some of the challenges of GUI testing are also relevant to testing of web applications and object-oriented software. One way to test these classes of software is to generate test cases that are sequences of events (*e.g.*, web user actions or method calls). The technique developed in this research has already been used by other researchers to prune the space of all possible event interactions to be tested for Ajax-based web applications [2]. Similar extensions may be explored for object-oriented software.

Bibliography

[1] JUnit, Testing Resources for Extreme Programming. http://junit.org/news/extension/gui/index.htm.

[2] Alessandro Marchetto, P. T., and Ricca, F. State-based testing of Ajax web applications. In *Proceedings of the 1st International Conference on Software Testing, Verification, and Valication* (April 9–11, 2008), pp. 121–130.

[3] Apfelbaum, L. Automated functional test generation. In *Autotestcon '95 Conference* (1995), IEEE.

[4] Barnett, M., Grieskamp, W., Nachmanson, L., Schulte, W., Tillmann, N., and Veanes, M. Towards a tool environment for model-based testing with AsmL. In Petrenko and Ulrich [45], pp. 252–266.

[5] Belli, F. Finite-state testing and analysis of graphical user interfaces. In *ISSRE* (2001), IEEE Computer Society, pp. 34–43.

[6] Blackburn, M. R., Busser, R., Nauman, A., and Chandramouli, R. Model-based approach to security test automation. In *13th International Symposium on Software Reliability Engineering (ISSRE)* (Annapolis, MD, 2002).

[7] Blackburn, M. R., Busser, R., Nauman, A., Knickerbocker, R., and Kasuda, R. Mars polar lander fault identification using model-based testing. In *ICECCS* (2002), IEEE Computer Society, pp. 163–.

[8] Boyapati, C., Khurshid, S., and Marinov, D. Korat: automated testing based on java predicates. In *ISSTA '02: Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis* (2002), pp. 123–133.

[9] Briand, L. C., Labiche, Y., and Wang, Y. Using simulation to empirically investigate test coverage criteria based on statechart. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering* (2004), IEEE Computer Society, pp. 86–95.

[10] Brownlie, R., Prowse, J., and Phadke, M. S. Robust testing of AT&T PMX/StarMAIL using OATS. *AT&T Technical Journal 71*, 3 (1992), 41–47.

[11] Bryce, R. C., Rajan, A., and Heimdahl, M. P. E. Interaction testing in model-based development: Effect on model-coverage. In *APSEC '06: Proceedings of the XIII Asia Pacific Software Engineering Conference* (2006), pp. 259–268.

[12] Campbell, C., Grieskamp, W., Nachmanson, L., Schulte, W., Tillmann, N., and Veanes, M. Model-based testing of object-oriented reactive systems with spec explorer., May 2005.

[13] Chays, D., Dan, S., Deng, Y., Vokolos, F. I., P. G. F., and Weyuker, E. J. AGENDA: A test case generator for relational database applications˙ Tech. rep., Polytechnic University, 2002.

[14] Cohen, D. M., Dalal, S. R., Fredman, M. L., and Patton, G. C. The AETG system: an approach to testing based on combinatorial design. *IEEE Transactions on Software Engineering 23*, 7 (1997), 437–444.

[15] Cohen, M. B., Colbourn, C. J., Gibbons, P. B., and Mugridge, W. B. Constructing test suites for interaction testing. In *Proceedings of the International Conference on Software Engineering* (May 2003), pp. 38–48.

[16] Cohen, M. B., Dwyer, M. B., and J.Shi. Coverage and adequacy in software product line testing. In *Proceedings of the Workshop on the Role of Architecture for Testing and Analysis* (July 2006), pp. 53–63.

[17] Cohen, M. B., Dwyer, M. B., and Shi, J. Interaction testing of highly-configurable systems in the presence of constraints. In *International Symposium on Software Testing and Analysis* (July 2007). to appear.

[18] Dunietz, I. S., Ehrlich, W. K., Szablak, B. D., Mallows, C. L., and Iannino, A. Applying design of experiments to software testing. In *Proceedings of the International Conference on Software Engineering* (1997), pp. 205–215.

[19] Farchi, E., Hartman, A., and Pinter, S. S. Using a model-based test generator to test for standard conformance. *IBM Systems Journal 41*, 1 (2002), 89–110.

[20] Ferguson, R., and Korel, B. The chaining approach for software test data generation. *ACM Trans. Softw. Eng. Methodol. 5*, 1 (1996), 63–86.

[21] Gallagher, M. J., and Narasimhan, V. L. Adtest: A test data generation suite for Ada software systems. *IEEE Trans. Software Eng. 23*, 8 (1997), 473–484.

[22] Gupta, N., Mathur, A. P., and Soffa, M. L. Automated test data generation using an iterative relaxation method. In *SIGSOFT FSE* (1998), pp. 231–244.

[23] Hartman, A. Software and hardware testing using combinatorial covering suites. In *Graph Theory, Combinatorics and Algorithms: Interdisciplinary Applications* (2005), pp. 327–266.

[24] Hicinbothom, J. H., and Zachary, W. W. A tool for automatically generating transcripts of human-computer interaction. In *Proceedings of the Human Factors and Ergonomics Society 37th Annual Meeting* (1993), vol. 2 of *SPECIAL SESSIONS: Demonstrations*, p. 1042.

[25] Hong, H. S., Kwon, Y. R., and Cha, S. D. Testing of object-oriented programs based on finite state machines. In *APSEC* (1995), IEEE Computer Society, p. 234.

[26] Hovemeyer, D., and Pugh, W. Finding bugs is easy. *SIGPLAN Not. 39*, 12 (2004), 92–106.

[27] Jorgensen, A. A., and Whittaker, J. A. An application program interface (API) testing method. In *STAREast Conference* (May 2000).

[28] Kasik, D. J., and George, H. G. Toward automatic generation of novice user test scripts. In *Proceedings of the Conference on Human Factors in Computing Systems : Common Ground* (New York, 13–18 Apr. 1996), ACM Press, pp. 244–251.

[29] Koopman, P. W. M., Plasmeijer, R., and Achten, P. Model-based testing of thin-client web applications. In *FATES/RV* (2006), K. Havelund, M. Núñez, G. Rosu, and B. Wolff, Eds., vol. 4262 of *Lecture Notes in Computer Science*, Springer, pp. 115–132.

[30] Korel, B. Automated software test data generation. *IEEE Trans. Software Eng. 16*, 8 (1990), 870–879.

[31] Kuhn, D., Wallace, D. R., and Gallo, A. M. Software fault interactions and implications for software testing. *IEEE Transactions on Software Engineering 30*, 6 (2004), 418–421.

[32] Lee, N. H., and Cha, S. D. Generating test sequences from a set of mscs. *Computer Networks 42*, 3 (2003), 405–417.

[33] Lucio, L., Pedro, L., and Buchs, D. A methodology and a framework for model-based testing. In *RISE* (2004), N. Guelfi, Ed., vol. 3475 of *Lecture Notes in Computer Science*, Springer, pp. 57–70.

[34] Memon, A. M. *A Comprehensive Framework for Testing Graphical User Interfaces*. Ph.D. thesis, Department of Computer Science, University of Pittsburgh, jul 2001.

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

[36] Memon, A. M., Banerjee, I., and Nagarajan, A. GUI ripping: Reverse engineering of graphical user interfaces for testing. In *Proceedings of The 10th Working Conference on Reverse Engineering* (November 2003).

[37] Memon, A. M., Nagarajan, A., and Xie, Q. Automating regression testing for evolving GUI software. *Journal of Software Maintenance and Evolution 17*, 1 (Jan. 2005), 27–64.

[38] Memon, A. M., Pollack, M. E., and Soffa, M. L. Hierarchical GUI test case generation using automated planning. *IEEE Trans. Softw. Eng. 27*, 2 (2001), 144–155.

[39] Memon, A. M., and Xie, Q. Using transient/persistent errors to develop automated test oracles for event-driven software. In *ASE '04: Proceedings of the 19th IEEE international conference on Automated software engineering* (Washington, DC, USA, 2004), IEEE Computer Society, pp. 186–195.

[40] Memon, A. M., and Xie, Q. Studying the fault-detection effectiveness of GUI test cases for rapidly evolving software. *IEEE Trans. Softw. Eng. 31*, 10 (2005), 884–896.

[41] Michael, C. C., McGraw, G., and Schatz, M. Generating software test data by evolution. *IEEE Trans. Software Eng. 27*, 12 (2001), 1085–1110.

[42] Micskei, Z., and Majzik, I. Model-based automatic test generation for event-driven embedded systems using model checkers. In *DepCoS-RELCOMEX* (2006), IEEE Computer Society, pp. 191–198.

[43] Miller, W., and Spooner, D. L. Automatic generation of floating-point test data. *IEEE Trans. Software Eng. 2*, 3 (1976), 223–226.

[44] Pacheco, C., Lahiri, S. K., Ernst, M. D., and Ball, T. Feedback-directed random test generation. In *ICSE '07: Proceedings of the 29th International Conference on Software Engineering* (Washington, DC, USA, May 23–25, 2007), IEEE Computer Society, pp. 396–405.

[45] Petrenko, A., and Ulrich, A., Eds. *Formal Approaches to Software Testing, Third International Workshop on Formal Approaches to Testing of Software, FATES 2003, Montreal, Quebec, Canada, October 6th, 2003* (2004), vol. 2931 of *Lecture Notes in Computer Science*, Springer.

[46] Richardson, D. J., and Thompson, M. C. An analysis of test data selection criteria using the relay model of fault detection, 1993.

[47] Rothermel, G., Elbaum, S., Malishevsky, A. G., Kallakuri, P., and Qiu, X. On test suite composition and cost-effective regression testing. *ACM Trans. Softw. Eng. Methodol. 13*, 3 (2004), 277–331.

[48] Rountev, A., Kagan, S., and Gibas, M. Evaluating the imprecision of static analysis. In *Proceedings of the ACM-SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering* (2004), pp. 14–16.

[49] Shehady, R. K., and Siewiorek, D. P. A method to automate user interface testing using variable finite state machines. In *FTCS '97: Proceedings of the 27th International Symposium on Fault -Tolerant Computing (FTCS '97)* (Washington, DC, USA, 1997), IEEE Computer Society, p. 80.

[50] SIRER, E. G., AND BERSHAD, B. N. Using production grammars in software testing. In *PLAN '99: Proceedings of the 2nd conference on Domain-specific languages* (New York, NY, USA, 1999), ACM Press, pp. 1–13.

[51] THOMPSON, M. C., RICHARDSON, D. J., AND CLARKE, L. A. An information flow model of fault detection, 1993.

[52] WHITE, L., AND ALMEZEN, H. Generating test cases for GUI responsibilities using complete interaction sequences. In *ISSRE '00: Proceedings of the 11th International Symposium on Soft ware Reliability Engineering* (Washington, DC, USA, 2000), IEEE Computer Society, p. 110.

[53] WHITE, L. J. Regression testing of GUI event interactions. In *International Conference on Software Maintenance* (1996), pp. 350 – 358.

[54] WHITTAKER, J. A. Stochastic software testing. *Ann. Software Eng. 4* (1997), 115–131.

[55] XIE, Q., AND MEMON, A. M. Automated model-based testing of community-driven open source GUI applications. In *Proceedings of the 22nd IEEE International Conference on Software Maintenance* (2006).

[56] XIE, Q., AND MEMON, A. M. Studying the characteristics of a 'good' GUI test suite. In *Proceedings of the 17th IEEE International Symposium on Software Reliability Engineering (ISSRE 2006)* (Nov. 2006), IEEE Computer Society Press.

[57] XIE, Q., AND MEMON, A. M. Designing and comparing automated test oracles for gui-based software applications. *ACM Transactions on Software Engineering and Methodology 16*, 1 (2007), 4.

[58] XIE, T., AND NOTKIN, D. Mutually enhancing test generation and specification inference. In Petrenko and Ulrich [45], pp. 60–69.

[59] YILMAZ, C., COHEN, M. B., AND PORTER, A. Covering arrays for efficient fault characterization in complex configuration spaces. *IEEE Transactions on Software Engineering 31*, 1 (2006), 20–34.

[60] YUAN, X., AND MEMON, A. M. Using GUI run-time state as feedback to generate test cases. In *ICSE '07: Proceedings of the 29th International Conference on Software Engineering* (Washington, DC, USA, May 23–25, 2007), IEEE Computer Society, pp. 396–405.