A Regression Test Selection Technique for Graphical User Interfaces

Carl Chesser

B.S., Computer Technology, Purdue University, 2005

Submitted to the graduate degree program in the department of Electrical Engineering and Computer Science and the Graduate Faculty of the University of Kansas in partial fulfillment of the requirements for the degree of Master of Science.

Dr. Hossein Saiedian
Professor and Chairperson

Dr. Bo Luo
Assistant Professor

Dr. Perry Alexander
Professor

May 4, 2012

Date Defended

The Thesis Committee for Carl Chesser certifies
that this is the approved version of the following thesis:

A Regression Test Selection Technique for Graphical User Interfaces

____
Dr. Hossein Saiedian
Professor and Chairperson

____
Dr. Bo Luo
Assistant Professor

____
Dr. Perry Alexander
Professor

May 4, 2012
____
Date Approved

Abstract

Regression testing is a quality control measure to ensure that the newly modified part of the software still complies with its specified requirements and that the unmodified part has not been affected by the maintenance activity. Regression testing is an important and expensive activity during the software maintenance process and its purpose is to ensure quality and reliability in modified software. Regression testing selection techniques are focused on the reusability of existing test suites for a modified program from a previous version. Many regression testing selection techniques have been approached for conventional and object-oriented software. There is little discussion about those techniques to be applied for the Graphical User Interfaces (GUIs). This thesis addresses the gap. GUIs have characteristics different from traditional software, and the conventional testing techniques do not directly apply to GUIs. Unlike most previous techniques for selective retest, this thesis focuses on developing an event driven regression testing selection technique for GUIs. It defines an event dependence graph (EDG) to identify the interaction and relationship of the events within GUI components, develops an algorithm to construct the EDG for GUIs, and presents the GUI modeling structure and its selection retest technique. An algorithm is given to determine and generate a modified test suite automatically for GUI based on its original version. Experiments are presented on an implementation of this solution and discusses newly found challenges when

applied to an established GUI application. Finally, feasibility and future areas of research are addressed on the findings during the implementation of the solution.

Acknowledgements

I would like to show my gratitude to my advisor, Dr. Hossein Saiedian, whose guidance and instruction was pivotal in the development of this subject matter. Furthermore, this thesis benefited from previous research conducted by Kevin Yu while working under Dr. Saiedian. I would also like to thank the committee members, Dr. Bo Luo and Dr. Perry Alexander, for their time and consideration in assessing my work in this area. Finally, I would like to thank Cerner Corporation for their financial support in my education and my family and friends who have given me support throughout the time of pursuing this degree.

Contents

Abstract iii

Acknowledgements v

1 Introduction 1
1.1 Problem Statement 2
1.2 Significance 2
1.3 Research Methodology 4
1.4 Organization 5

2 Background 6
2.1 GUI Testing 7
2.2 GUI Regression Testing 9
2.3 GUI Event Modeling 12
2.3.1 Control Flow Graph 13
2.3.2 Program Dependence Graph 13
2.3.3 Event Flow Graph 15
2.3.4 Event Interaction Graph 15
2.4 GUI Regression Test Generation 16
2.5 Related Work 19
2.6 Summary 21

3 Algorithms for Developing an Event Dependence Graph 23
3.1 GUI Components and Event Classification 24
3.1.1 Event Dependence Graph 26
3.1.2 Construction of Event Dependence Graph 29
3.2 Regression Test Selection Techniques for GUIs 33
3.2.1 Motivating Observations for Testing a Modified Program 33
3.2.2 The Test Selection Algorithm 35
3.3 Summary 40

4 Research Results Evaluation and Validation **42**
4.1 Static and Dynamic Analysis of GUI Components 44
4.2 GUITAR Framework Overview 49
4.3 Apache JMeter Application Assessment 49
4.4 Graphical User Interface Ripping Process 51
4.5 Event Dependence Graph Construction 60
4.5.1 Parsing the EDG output from GUI Ripper 60
4.5.2 Analyzing GUI model to identify events in Event Dependence Graph 62
4.6 Test Case Selection Data Model and Process 65
4.7 Feasibility of the Proposed Solution Implementation 69
4.8 Summary 72

5 Contributions and Areas for Further Research **73**
5.1 Summary 73
5.2 Research Contributions 74
5.3 Future Work 75
5.3.1 Accurately Maintaining an Identity of a Modified Event 76
5.3.2 Automated Identification of Changed Events 76
5.3.3 Improving the GUI Ripping Process 77

Bibliography **78**

List of Figures

2.1 Control Flow Graph 14
2.2 Program Dependence Graph 15
2.3 Event Flow Graph 16

3.1 Microsoft Notepad 24
3.2 Pictorial Symbols for the GUI Events 26
3.3 A Copy-Paste Edge in Notepad 28
3.4 Open and Save-As Screens in Notepad 31
3.5 An EDG for Editing a File 32
3.6 Screen Shots of of Find Screen and its Modified Version 39
3.7 EDG for Find Screen and its Modified Version 40

4.1 Overview of Implementation 44
4.2 Initial Edit menu options and Edit menu options after state change . . 57
4.3 Graph of time taken for GUI event parsing 62
4.4 Graph of time taken for EDG construction 64
4.5 GUI Test Case Entity Relationship Diagram 66
4.6 JMeter Search menu 68
4.7 JMeter Search window 69

List of Tables

3.1 Test Cases for Opening a File in Notepad 35

4.1 JFC GUI Ripper Arguments 52
4.2 JMeter GUI Ripping Results 58
4.3 JMeter GUI Ripping Timings (in seconds) 59
4.4 JMeter GUI Ripping Timing Summary (in seconds) 59
4.5 JMeter GUI Parsing Times of Ripping Output (in milliseconds) 61
4.6 Conceptual Example of Data 67

Chapter 1

Introduction

Graphical User Interfaces (GUIs) are normally more complex than the other traditional components of a software. GUIs present a "fluid" interface which can be changed at the whim of the users. GUI testing is different and difficult in that the input is interactive whereas the output may be graphical or may be an event. An especially serious question can be asked in the software maintenance phase where modifications are made to the GUI application: How can a modified GUI application be tested? Regression testing is the process of validating modified software to provide confidence that the changed parts of the software behave as intended and that the unchanged parts of the software have not been adversely affected by the modifications. Therefore, regression testing plays an integral role for the quality and reliability in the software maintenance process, especially in the GUI applications. Although the use of GUIs continues to grow very fast [8, 44], GUI testing has, until the past decade, remained a neglected research area [20]. Recent advances in GUI testing have focused on the development of test case auto generation, test oracles, and coverage criteria for GUI testing [26, 28, 32, 41]. Moreover, the development of regression testing selection techniques for GUIs has not been extensively addressed.

1.1 Problem Statement

It is desired to efficiently apply regression testing to a GUI application when changes are applied. GUI applications are considered complex due to the large series of states which they can potentially represent. With a large frontier of possible application states, many valid testing flows will also exist for that application. The costs of running all possible tests or performing analysis to determine which subset of tests to execute for a given change can be a very costly exercise to perform (in terms of time). It is desired to provide an effective model which represents a GUI application that can be used to determine affected areas of an application from a given change. This model can then support a regression test selection method that is tailored for GUI applications which is capable of selecting the minimal required set of tests which effectively validate the changes applied.

1.2 Significance

Regression test selection is the process of selecting an appropriate subset of the original test suite for the given assumptions. A regression test case selection technique is considered safe if it will never exclude a test case if that test case can reveal faults in the modified software [36]. Regression test selection techniques can be used for analyzing relationships between the test cases and the software entities they cover, using information about changes to select test cases for new versions. Many regression test selection techniques have been presented in recent years [13, 31, 33, 34, 37, 38]. These developed regression test selection techniques are applied in both procedural-language software and object-oriented software and focus on executing every statement of program's code which relates to the applied

change. Therefore, they can be used to test modified class and derived class. However, those regression test selection techniques do not address the adequacy of GUI regression testing for a number of reasons. First, a GUI application is an event driven software. The input for this type of application is an ordered series of events. Based on the series of events utilized, a large range of states in the application are possible. Second, the source code tests may not completely cover the GUI testing because there are many unsolicited events [28] Moreover, the event sequences that must be tested on the GUI are conceptually at a much higher level of abstraction that cannot be obtained from the code. This means that simply focusing on structure of the code to dictate which tests are to be included in the regression test execution is not sufficient without the context of the events which trigger the affected changes. These challenges suggest that we need to develop regression test selection techniques based on events for GUI applications because events are a key characteristic in this type of program. There are several requirements for the development of regression test selection techniques. First, the GUI applications need to be decomposed into some smaller GUI components so that a unit of testing can be performed for each GUI component. This is because there might be an enormous number of possible event permutations of GUIs. Second, we need a way of modeling the GUI's intended behavior so that we can make a comparison between a test suite and its modified version. This thesis defines an event dependence graph, a new event interaction and relationship, for the GUI component, and presents a new regression test selection technique that addresses the regression test selection problem for GUIs by constructing the event dependence graphs for GUI components. This approach has several benefits. First, the technique is currently the only selective retest technique that is an event based test technique for GUI applications. Second, it selects test cases based on events, not code explicitly. Third, it is independent of the method

used to generate tests initially for GUI components. Fourth, it selects every test that may produce different output in the modified GUI components. Finally, it is automatic.

1.3 Research Methodology

An empirical research methodology was applied when evaluating a new event graph model and its effectiveness. First, the investigation of existing and related research on GUI applications, general testing and GUI specific testing, regression testing methods, GUI modeling, and approaches on event based analysis of GUI applications were assessed. During this research, the work of Greg Rothermel, in the area of regression testing methods, and Atif Memon, in the area of GUI modeling, analysis and testing, were key sources of information due to their significant contributions in these fields. In addition to these two key contributors, other sources in areas of GUI testing, test selection methods, and areas of automated GUI testing approaches were evaluated. After investigating these topics and establishing an understanding of the current state of research in these areas, the gap of regression test selection based solely on event state of a GUI application was further examined. A hybrid model was then proposed from the event-flow graph [28] and the program dependence graph [33, 37, 38], to represent the GUI application. Experiments were then applied against this proposed model to determine its effectiveness and feasibility in implementation. Through the experiments applied, several new challenges in the complexity of GUI applications were discovered. In this thesis, these experimental findings helped prove the proposed solution's effectiveness and opened new areas of future to extend the ideas and concepts proposed in this thesis.

1.4 Organization

The organization of this thesis is as follows:

- **Chapter 2: Background** - This chapter provides background information on related research and foundational information which supports the concepts proposed in this thesis. GUI testing, regression testing, GUI event modeling, test generation, and other related topics are presented.

- **Chapter 3: Algorithms for Developing an Event Dependence Graph** - GUI event classification is described and then further extended into a new GUI model which supports the proposed solution for a regression test selection method for GUI applications. In this chapter, the event dependence graph is defined and illustrated through examples. Related algorithms to support the event dependence graph construction and graph traversal for test selection are also presented.

- **Chapter 4: Research Results Evaluation and Validation** - This chapter will take the approach described from Chapter 3, and apply to an established GUI application (Apache JMeter). Implementation details and reasoning behind implementation choices are discussed. Performance measurements are taken throughout the experiments to provide context of relative costs as compared to other stages of the implementation. Tests are then applied on the implementation to evaluate if proposed solution supports the expected behavior. A feasibility section is included which lists findings from the experiments applied.

- **Chapter 5: Conclusion** - In this chapter, the findings from the evaluation (Chapter 4) are summarized and candidates of future research are stated.

Chapter 2

Background

Today's GUI applications give users more control and flexibility, which translates into a dramatic increase in the number of situations that need to be tested during software development processes. Regression testing verifies that previously identified problems have been corrected, and that these "corrections" have not caused problems elsewhere. Thus, GUI applications raise interesting concerns for regression testing [43]. This section discusses our concerns relevant to GUI testing, regression testing and its selective techniques; describes the selection retest techniques for GUI applications. Selective retest techniques reduce the cost of regression testing by reusing existing test cases and identifying portions of the modified GUI components. Selective retest techniques differ from the retest-all technique, which run all test cases in the existing test suite. Section 2.1 provides the general information for GUI applications. Section 2.2 describes general approaches to regression test selection techniques for GUI applications.

2.1 GUI Testing

Graphical User interfaces add a new dimension of complexity to software testing [3, 22]. GUIs have brought considerable benefits to developers. They release the developer from the concerns of interface design in most environments. The GUI design standards impose conventions which make one application look very much like another on the same platform. In addition, GUIs free the user to access system functionality in their preferred way. They have a permanent access to all features and may use the mouse, the keyboard or a combination of both to have a more natural dialogue with the system.

GUI testing is a difficult problem to solve due to the large number of states to be tested. The input space is extremely large because of the different permutations of inputs and events that affect GUIs. The complex GUI objects and event dependencies will increase the complexity of testing. With the complex nature of GUI applications, testing is a challenge that is continually faced when changes are applied to the application. Some of the reasons for these challenges are as follows:

- Event-Driven Software: The event-driven nature of GUIs presents the first serious testing challenge. Because users may click on any pixel on the screen, there are significantly more possible user inputs that can occur. The user has an extremely wide choice of actions. At any point in the application, the users may click on any field or object within a window. They may bring another window in the same application to the front and access that. The window may be owned by another application. The user may choose to access an operating system component directly, e.g. a system configuration control panel [5].

- Unsolicited Events: Unsolicited events cause problems for programmers and

testers due to their originating nature. For example, a message-oriented middleware component might dispatch a message (an event) to remind the client application to redraw a diagram on screen, or refresh a display of records from a database that has changed. Testing of unsolicited events is difficult because the number of test cases may be high and special test drivers may be necessary to generate such events within the operating systems.

- Hidden Synchronization: It is common for window objects to have some forms of synchronization implemented. For example, if a check box is set to true, a text box intended to accept a numeric value elsewhere in the window may be made inactive or invisible. The GUI developers must use the event handling mechanisms to implement the synchronization functionality, so it is challenging to identify all related events that contribute to targeted synchronization points in the application which are desired to test.
- Large Magnitude of Input/Output: GUI applications can take a large range of different forms of input and therefore can produce a large range of output to support feedback when these means of input are applied. With a large range of inputs/outputs to test, it produces significant challenges to thoroughly analyze all of these points to test when a change is applied to the program [42]. More importantly, this threatens an organization's mean of scaling their abilities in performing tests as inventory of GUI applications expand and the frequency of testing increases.

2.2 GUI Regression Testing

A regression test is a comprehensive retest of the entire GUI applications and/or their dependent components after validation that the defects or enhancements were successfully implemented. A regression test should be performed to ensure that the GUI applications still work as designed. This testing is focused on testing areas of the application which relate to the change that was applied to the GUI application [27]. If only portions will be affected by a modification, then only a partial regression test of the affected portion will be necessary. The complete regression testing should be performed when the whole system architecture has been significantly affected by a modification. The GUI regression testing includes two phases: initial phase and critical phase. During the initial phase, while the GUI application is still under developing process, its regression testing is not on critical. Testers may retest modified units, develop test plans, and do limited integration tests, but the bulk of the testing effort awaits inclusion of the final modifications. When modifications are complete, regression testing enters the critical phase, where the final integration and all application test must be selected and executed. The testing circumstances (sufficient test time and personnel) affect the regression test for GUI application in real life scenarios. For example, retest-all technique can simply reuse all existing test cases while ad hoc/random techniques will be applied when time constraints prohibit the use of a retest-all approach. Therefore, it is in the critical phase that which can have large cost implications of the project if it is not performed effectively. Regression test selection techniques can be applied in these phases to minimize the testing execution effort by selecting the minimal set of tests which can safely ensure that no new fault will be missed. A variety of regression test selection techniques have been described in the research literature. Rothermel

& Harrold [34] analyzed and classified these selection techniques.

One approach of regression testing that is identified by Rothermel & Harrold [37,38] is the *retest all* method, which executes all existing test plans during regression testing. This approach is simple and highly effective, as all existing test plans are utilized. This avoids the need to perform analysis for selecting tests that are capable of exposing potential faults which are introduced with the newly changed application. However, this is normally the highest cost option when tests require to be executed through human intervention. If the tests are automated, this is a common approach applied since the cost is relatively low to re-execute the existing tests. Therefore, it has become an attractive option to adopt automated testing for GUI applications due to the power of being able to execute all tests with low costs.

Memon & Xie [29] identify how automated regression testing commonly is applied to GUI applications by either:

- Bypassing GUI components to test business logic: This approach obviously faces challenges in quality assurance as it is scoped to only the business logic components and it is not testing the end-to-end user experience. This approach allows many different well established unit testing frameworks (i.e., *xUnit family frameworks*) to be leveraged; however, this testing approach avoids some of the desired integration aspects of testing the GUI events. Therefore, this form of testing is less desirable when testing the GUI events is the changed component in the software.

- Test GUI components by utilizing an external tool to *record/playback* [14] a testing walkthrough of the application: The *recording* stage of the process is when user is manually executing the software and their steps are being recorded. The *playback* stage is when the test is executed automatically, by

going through the same steps the user performed during the *recording* stage. This approach relies on manual selection of the flows to test event space which is possible, and therefore its effectiveness hinges on the expertise of the testers involved [11, 40, 47].

If a GUI application does not have a full suite of automated tests, it is generally not feasible to follow the *retest all* method. Therefore, GUI applications are favored to go through a different selection process of determining test cases during regression testing. Moreover, due to the difficulty of GUI testing discussed in the previous section, faults can be effectively discovered through testing events as they occur in the entire application, rather than testing an identified event in isolation. Thus, if we want to be sure that we have executed all existing tests that may expose faults in a modified GUI component, the entire application needs to be assessed in terms of its relationship to the modified component. Rothermel & Harrold [37, 38] define a method for selecting tests as "safe" if it selects all tests from the original test set which can expose faults in the modified component of the application. Therefore, it is desired to seek a test case selection method which is safe to produce the most effective test set for regression testing.

The general selective retest process is provided by [37, 38]. As a special case, the following describes the typical GUI selective retest process: Let G be a GUI component, let G' be a modified version of G, and let T be a set of test cases (a test suite) created to test G. A GUI regression testing will proceed as follows:

1. Select $T' \subseteq T$, a set of test cases to execute G'
2. Test G' with T', establishing G' correctness with respect to T'
3. If necessary, create T'', a set of new test cases for G'

4. Test G' with T'', establishing G' correctness with respect to T''

5. Create T''', new test cases and test history for G', from T, T', T''.

As Rothermel & Harrold [37,38] define this process, they also identify several problems that are addressed at each step. Step 1 involves the problem of selecting the test cases (T') from the existing test set (T') to test G', which is known as the regression test selection problem. Step 2 addresses the problem of efficiently testing G' with T', which is known as the test suite execution problem. Step 3 addresses the problem of performing additional testing to ensure that all changes which exist in G' are being covered, which is called the coverage identification problem. Step 4 also addresses the test suite execution problem in performing the additional tests (T'') to fully cover G'. Finally, Step 5 addresses the problem of managing all the test data which is created from T, T', T'', which is known as the test suite maintenance problem. Although each of these problems is significant, this thesis will only restrict our attention to the regression test selection problem for GUI applications.

2.3 GUI Event Modeling

In modeling processes and state of an application, many approaches can apply. For GUI event modeling, other modeling approaches can be evaluated as each approach has relatable attributes which lend themselves to illustrating GUI event flows. In this section, a high level summary of different types of graphs which are supportive of GUI event modeling will be described.

2.3.1 Control Flow Graph

One of the core types of modeling used to illustrate software processing is a control flow graph [33, 37, 38]. A control flow graph provides a simple illustration of the continuity between logical sets of operations [7]. Through its illustration, logic of the application is clear where branching would occur and how the application navigates through its flows. For example, in Listing 1, a simple set of instructions exist.

This set of instructions from Listing 1 can then be illustrated in a generic control flow graph, presented in Figure 2.1. In the control flow graph, the instructions are illustrated as vertices and directed edges flow from those which indicate possible paths in the graph. In addition, the control flow graph establishes to key elements: an entry point (source) and exit point (sink). By having these established in the graph, one can discern where it must begin in the graph and when to terminate in performing graph traversals.

2.3.2 Program Dependence Graph

Another core type of graphing model which can be used to illustrate an application is a program dependence graph [12]. The program dependence graph is somewhat similar to a control flow graph in that it captures the flow of which operations are

```
int courseNum = 899
if (courseNum >= 700) {
   graduateCourse = 1;
} else {
   graduateCourse = 0;
}
```

Listing 1: Simple Set of Procedural Statements

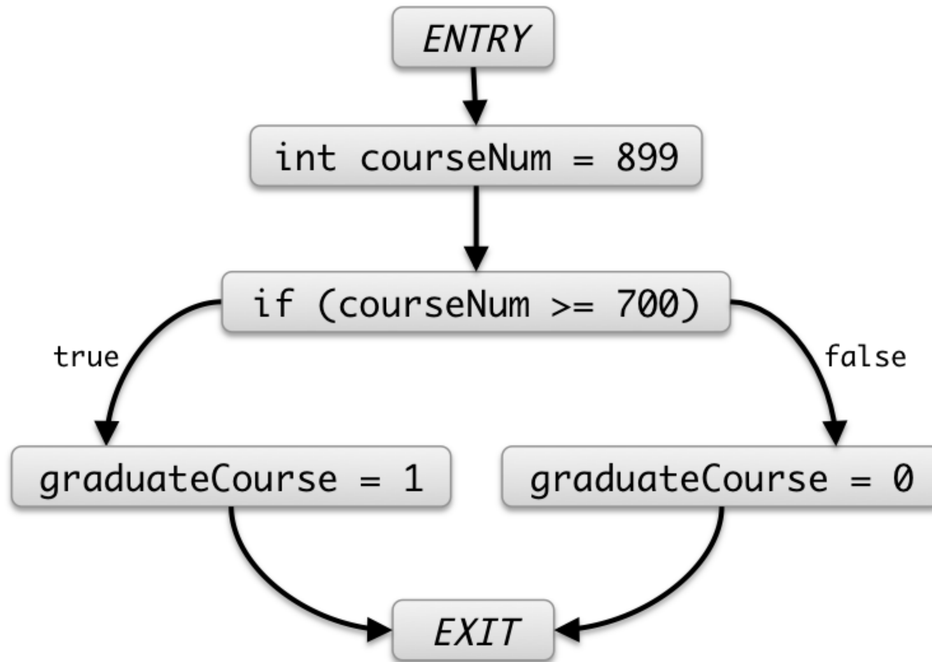

Figure 2.1: Control Flow Graph

performed in the application. Two key types of relationship information is presented in the program dependence graph. The first type is control dependencies and the seconds is data dependencies [30]. Relationships are directed edges in the graph which illustrate the dependency of either control or data for a given operation. This is type of graph is more expensive to construct due to its additional relationship information that it captures. With that cost, additional benefits exist that support different forms analysis of the program. Typical types of analysis that utilize the program dependence graph are motivated to discover optimizations that achievable through parallelism [6, 12]. In Figure 2.2, an example program dependence graph depicts the relationships which exist from the operations provided in Listing 1.

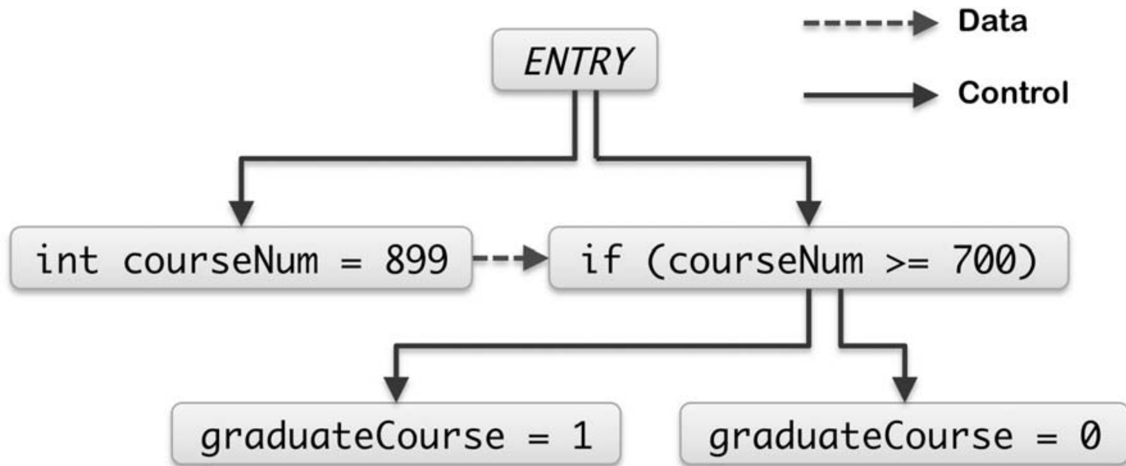

Figure 2.2: Program Dependence Graph

2.3.3 Event Flow Graph

A specific type of modeling which is tailored to illustrate GUI events, is the event flow graph [2, 19, 29, 46]. This graph communicates the flow of events by depicting their sequence in how they can be invoked in the application. Within this graph, all event sequence permutations which are possible in the program are provided. In Figure 2.3, an abstract example of this graph is provided which illustrates a possible event flow when opening a file. This graph illustrates the flow (or possible flow) of events through the directed edges which relate to each event (listed as a vertex). The general layout of this graph is similar to how a control graph is modeled [47].

2.3.4 Event Interaction Graph

The event interaction graph is a more selective type of graph than an event flow graph, in which it only represents the events which can invoke business logic components of the application [29, 49]. Therefore, the events modeled in this graph do not represent GUI control events like minimize or maximize of a window. This re-

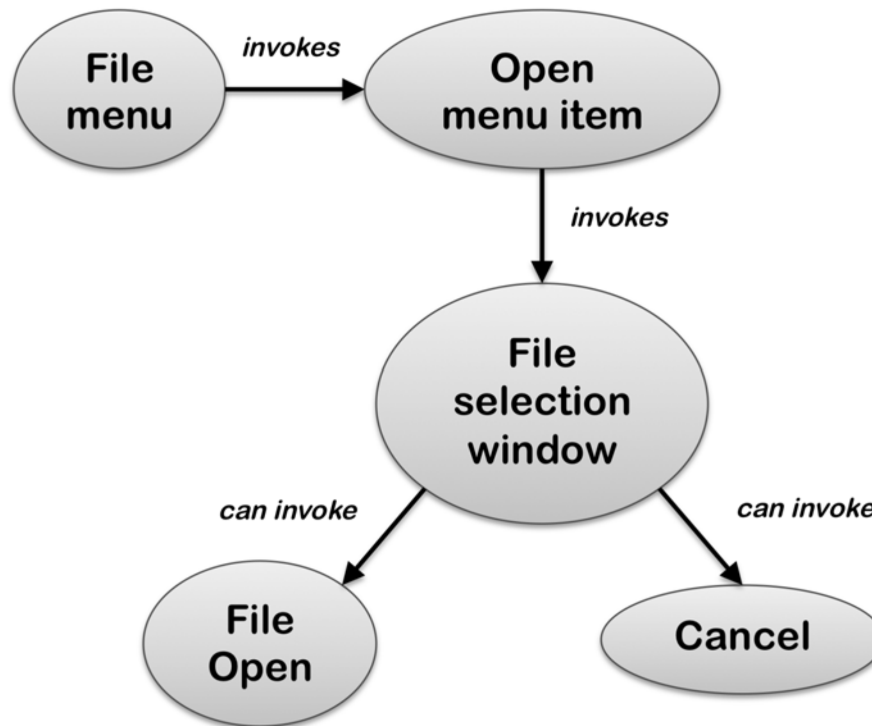

Figure 2.3: Event Flow Graph

stricted type of graph simplifies the views presented by only showing the key event which dictate how business logic is being invoked within the program [47].

2.4 GUI Regression Test Generation

In recent years, automated test case generation has gained more attention due to its relative low cost and its generic assessment to find faults in an application. This section will present some of the recent developments of this area to provide insight of the different approaches being used. Although, these approaches do not lend themselves to a selection method of existing tests, their approaches have similar aspects to the proposed solution of this thesis. Some of these similarities

include the application of reverse engineering to decompose a GUI application to its supporting events and then constructing an abstract model to determine tests to be applied given a set of criteria (what events to test).

Utilizing usage patterns of an application is another way to accurately identify real use-cases of the software when evaluating testing flows. Brooks & Memon [4] propose an approach of capturing the actual usage of a GUI application, known as a usage profile, which guides the testing efforts of what workflows to test. This usage profile captures a sequence of events which a user walks through in a common usage of the application. These events are then presented in event flow graphs which are constructed with two key nodes that serve as a start and end point in the graph for the testing flow. Thus, there is one entry point and one exit point of the graph. Weights are then associated to the edges of the graph, and a probabilistic event flow graph is then defined which provides the ability to determine the highest probable pairs of events that would occur for testing. Test cases are then generated to test for fault detection. Through their empirical study of this approach, they were able to produce a smaller test suite which was more effective in exposing faults in the application than just replaying the user profiles in how they were originally captured.

A challenging area in test case generation has been assessing potential test case flows from the software's static state. Memon and Yuan [50] evaluated a means of doing incremental sampling of an application to assess user feedback during a run-time test of a GUI application. The approach first builds a seed test suite based on an event interaction graph, which is then executed by an automated means. The run-time state is then captured and constructs a new type of graph, called an event semantic interaction graph, which is used to then determine the next set of tests to be executed based on the current run-time state. This approach explores a new way

of doing multi-way interaction testing of GUI events and in their experiments, was able to find new faults in existing open source software projects.

A dynamic test generation technique named "ALT" is proposed by Memon and Yuan [48, 51], which assists in generating test cases for GUI applications based on the run-time information assessed from prior execution. The name of "ALT" is derived from the steps of the process which alternates between running a set of tests and assessing program state to generate the next tests to execute. This style of alternating steps is a principle attribute of the process as it is continually evaluating the run-time state of the application after set of tests have been executed. This assessment of the program's state is essential during run-time, as the possible state of the GUI application cannot easily be determined from the static state of the code. Throughout this process, new tests are created based on the state of the program from the prior set of tests being executed, and unnecessary tests are avoided by only evaluating actual run-time states given the current event sequence. With this technique, the testing oracle evaluates a test result as passing if the program does not unexpectedly terminate. Therefore, the testing oracle in not comprehensive enough to expand beyond seeing invalid states of the program which do not unexpectedly terminate the program. Therefore, the strength of this technique is exposed in its ability to dynamically produce test cases based on the complex run-time state of the GUI application to seek event sequences which will cause the program to unexpectedly terminate.

In recent developments, Cohen, Memon, and Yuan [47] establish methods of utilizing covering arrays [10, 17] of combinatorial testing to assist in incorporating additional context for test case generation for GUI applications. This approach allows for a rich evaluation of event states within long event sequences to produce tests that are highly effective in producing faults. This approach focuses on the

importance of event context, the number of events within the test flow, and the location of the event within overall sequence. When evaluating the events, it considers the required state and dependencies of other events to know what events sequences are truly possible, as all potential event sequences may not be possible given their state dependencies. In their case study, they were able to determine that by increasing the number of unique event combinations and dictating the position of events in the overall sequence, they were able to generate more faults than from earlier methods which used shorter tests.

These advances in GUI test generation assist in building test suites with basic test oracles (test passes if the program does not crash); however, these advances do not solve the problem of selecting existing tests (with specific test oracles) that relate to the changes applied to the existing program.

2.5 Related Work

Some research has been presented in recent literature on regression test selection techniques for the GUI application strictly based on GUI event interactions. Memon, Pollack and Soffa [20] defined GUI event classifications and developed a test case generation system for GUIs. Memon, Soffa and Pollack [28] developed a GUI modeling system for the use of the GUI testing coverage criteria. Cohen, Memon, and Yuan [47] later expressed test case generation means through the construction of event-flow and event-interaction graphs.

There is a close relationship between test case generation techniques for original applications and their modified version. A number of researches have addressed the test case generation for GUIs based on the GUI event interaction techniques [20, 25, 26, 32, 41].

The most commonly used techniques to generate test cases for the modified programs are control flow graph and program dependence graph [13, 31, 34, 37, 38]. Those code-based techniques can be applied to programs for the structure languages as well as objected- oriented languages, classes, derived class, polymorphism and dynamic binding. By using those techniques, the test case generation for the modified application is safe and efficient [37]. Rothermel, Li & Bernett [34] applied the regression test selection techniques to the form- based visual program. In that paper, they developed a cell relation graph and used it to present adequacy criteria. Moreover, they described the differences between the form-based and imperative programming paradigms, and discussed effects that these differences have on strategies for testing form-based programs [35]. However, all of those techniques are based on the code and does not account for the context of events. Because of the complicity of GUI applications (e.g. interactions between GUI program and testers), code based testing techniques are not enough for the accurate of GUI testing [20].

In past years, Memon, Pollack & Soffa (2000, 2001) developed a nearly comprehensive framework for testing GUIs. This framework covers test case generator, test coverage evaluator, test executor, test oracle and regression tester. The adequacy of generated test suite is evaluated by the test coverage evaluator, which is employing event-based coverage criteria developed specifically for the GUI testing framework. In this framework, a test executor automatically executes all the test cases on the GUIs. As test cases have been executed, a test oracle automatically determines the correctness of the GUIs. The test oracle employs a model of the expected states of the GUI in terms of its constituent objects and their properties. If we put all of them together, the test case generator, test coverage evaluator, test executor, and test oracles provide the necessary mechanisms to automatically test GUI applications. However, the accuracy of this testing approach hinges on the effectiveness

of the testing oracle and the correct test cases being generated for the modified version of the program for the approach to be considered safe [1, 45].

In more recent developments, Memon [23] proposed an approach of repairing tests for regression testing, which can utilize an event flow graph to assist in selecting tests which are usable based on the modified version. For tests cases of the existing test suite which are not usable, they will be generated to support the test cases which no longer are capable of fully working against the modified version. This hybrid approach provides benefits of utilizing existing tests of a test suite, but adapts for cases which it cannot handle by generating tests for cases which can longer work for the modified version.

Based on these related works, a common requirement exists after changes are made to a GUI component: a regression tester needs to determine the parts of the GUI that have been modified and select a suitable subset on the test suite. Thus, the GUI test selection techniques presented in this thesis can be applied to the additional testing for the modified parts. This GUI regression test selection technique is general and can be used to test other applications that share the event driven characteristics of GUIs, such as object-oriented, web and reactive software.

2.6 Summary

In this chapter, related research work involving GUI applications and testing are presented. The first area addressed was GUI testing and the complications which they bring forth due to their complex nature. The focus of the their complexity lies in how they are defined by being event driven and the large event space for options of input into the application. This large event space presents challenges in fully testing the application to cover that event space, and accurately testing events

which are unsolicited. GUI regression testing is discussed and general regression test selection approaches are presented which are a means to avoid the *retest all* approach due to the costs that it can impose. Automated testing approaches are presented and explained in how they can be categorized. Additionally, automated testing approaches may not always be feasible for existing applications which automated testing frameworks are not available or extensive manual testing is already invested. A general regression selection approach is then defined to provide context of how a selection process would be applied and what is considered as a safe method in applying such an approach [37, 38]. GUI modeling types such as control flow graphs, program dependence graphs, event flow graphs, and event interaction graphs are explained by providing their core attributes and examples of each. GUI regression test generation approaches, which have had focus in several areas of the related research, were presented and explained with how they relate to the problem of selecting existing test cases. Finally closely related work in the area of test case generation based on a control flow graph and program dependence graph, repairing regression test cases by reusing working tests and generating new ones, and the utilization of testing oracles with test case generation approaches.

Chapter 3

Algorithms for Developing an Event Dependence Graph

Memon, Soffa and Pollack [26, 28] developed a framework for GUI structure and event classification. Based on their framework, the section presents the definition and concept of the event dependence graph (EDG) that extends the event flow graph defined by Memon, Soffa and Pollack [28]. Then Microsoft Notepad software is used as an example (shown as an Microsoft Notepad in Figure 3.1) to illustrate how to construct the event dependence graph. Lastly, an algorithm is provided to construct the EDG.

The important GUI characteristics include the graphical orientation, event-driven input, event interaction and relationship, and hierarchical structure. A GUI component consists of objects (buttons, menus, icon, etc) using metaphors familiar in real life. The users of GUI applications interact with the GUI components by performing events that manipulate the GUI component. GUI events cause deterministic changes to the state of the GUI application that may be reflected by a change in the appearance of one or more GUI components. Moreover, GUIs are hierarchical, and this

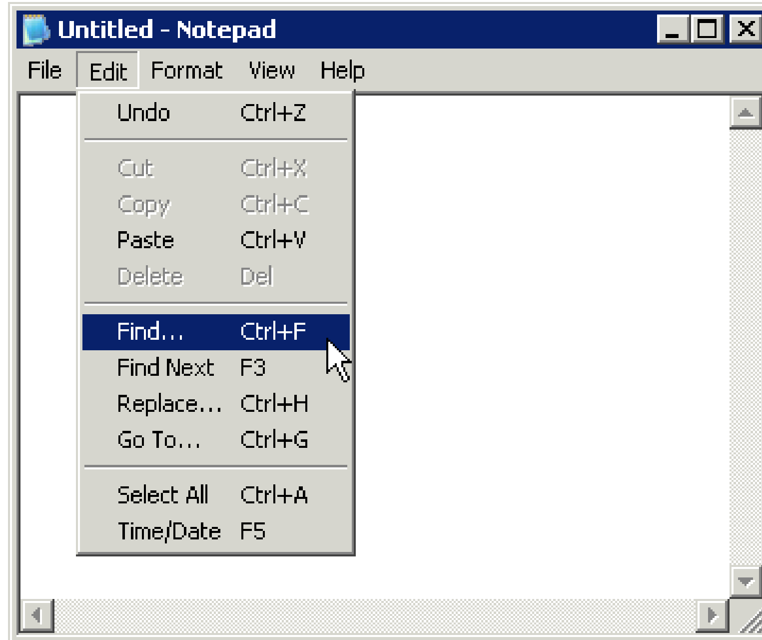

Figure 3.1: Microsoft Notepad

hierarchy may be exploited to identify groups of GUI events that can be tested in isolation [20]. To test GUI software properly, we must test GUI events [28]. GUI Event testing approaches typically invoke sequences of events in varying orders, and then verify that the resulting state of the GUI components manipulated by the events is correct.

3.1 GUI Components and Event Classification

This subsection lists some basic concepts defined by Memon, Soffa & Pollack [26,28]. A modal window is a GUI window, a kind of hierarchical GUI component, which monopolizes the GUI interaction and restricts the focus of the user to a specific range of events within the window until the window is explicitly terminated. The other window in the GUI is called a modeless window that does not restrict the user's focus. A GUI component G [28] is an ordered pair (R_f, R_c), where R_f represents a

modal window in terms of its events and R_c is a set whose elements represent the modeless windows also in terms of their events. Each element of R_c is invoked by an event in either R_c or R_f. An event flow graph (EFG) is a directed graph in which each node represents an event and each edge represents the possible interactions between two nodes (events).

Memon, Pollack & Soffa [20] defined the most commonly used GUI events in GUI applications:

- **Restricted-focus** events that open modal windows: Users have to explicitly close the window by clicking the `OK` or `Cancel` button and return to the original window. The common examples include the Save screen opened by clicking menu `File→Save` in Notepad.

- **Unrestricted-focus** events that open modeless windows that do not restrict user's focus. These events will expand the GUI events that are available for users. This will make GUI applications much more user friendly. The `Find` menu option that opens an unrestricted-focus window is a common example.

- **Termination** events that close modal windows: Every main window or modal window must have termination events that allow users to close the application or window. The OK or Cancel buttons and the '`x`' box on the right top of the window are very common termination events.

- **System-interaction** events that interact with the underlying software to perform some actions: The system-interaction events can be launched in any GUI component like button (the `Find Next` button), and menu option (`Save, Copy, Paste`, etc).

- **Menu-open** events that are used to open a menu list: They group a set of GUI

events available to the users. Unlike system- interaction events, the menu-open events do not interact with the underlying software. Most GUI applications include `File` and `Edit` menu options so users can have more menu selections.

Figure 3.2 lists the five different shapes that represent five defined events. Restricted-focus events (e.g., `Save As` under `File` menu option) are shown as rounded rectangles. Unrestricted-focus events (e.g., `Find` under `Edit` menu option) are shown as rectangles. Termination events (e.g., `OK` button) are represented as hexagons. System-interaction events (e.g., `Copy` under `Edit` menu option) are shown as ellipses. Octagons represent the menu-open events (e.g., `File` menu option).

3.1.1 Event Dependence Graph

This subsection defines the GUI event dependence graph (EDG) and its execution semantics. It also discusses the standard translation from a GUI component to its event dependent graph. Different shapes that represent different types of events are used to represent explicitly the event nodes in the EDG.

Definition 1 *For events v_x and v_y in G, v_x is an event flow dependence predecessor (efd-predecessor) of v_y if $(v_x, v_y) \in E$; and v_y is an event flow dependence successor (efd-successor) of v_x.*

Definition 2 *For each event $v \in V$, we define an event flow predecessor set (efp-set) P of v if $(p, v) \in E$ for any $p \in P$ and p is the efd-predecessor of v. An event flow successor set (efs-set) S of v if $(v, s) \in E$ for any $s \in S$ and s is the efd-successor of v.*

Figure 3.2: Pictorial Symbols for the GUI Events

Definition 3 *For events v_x and v_y in G, v_x is an event control dependence predecessor (ecd-predecessor) of v_y if $(v_x, v_y) \in E$ and the event v_y will be only launched after event v_x; v_y is an event control dependence successor (ecd-successor) of v_x.*

Definition 4 *For each event $v \in V$, we define an event control predecessor set (ecp-set) P of v if $(p, v) \in E$ for any $p \in P$ and p is the ecd-predecessor of v. An event control successor set (ecs-set) S of v if $(v, s) \in E$ for any $s \in S$ and s is the ecd-successor of v.*

Definition 5 *Definition: An event dependence graph Σ for a GUI component G is a 4-tuple $< V, E, B, \Psi >$ where:*

1. *V is a set of vertices representing all the events in the component. Each $v \in V$ represents an event in G.*
2. *$E \subseteq V \times V$ is a set of directed edges between vertices. An edge $(v_x, v_y) \in E$ if and only if the event represented by v_y is efd-successor or ecd-successor of the event represented by v_x.*
3. *$B \subseteq V$ is a set of vertices representing those events of G that are available to the user when the component is first invoked.*
4. *$P(v) \subseteq \Psi$, $P(v)$ is a ecs-set for the event $v \in G$.*

An event dependence graph represents all possible dependencies among the events in a component. The root of EDG is the ENTRY vertex, an event to start the GUI application. Double clicking on an icon is a common example. Every event vertex is reachable from the ENTRY vertex, and the termination events are reachable from every event vertex on the GUI component. If an edge $(v_x, v_y) \in E$, and v_x is only a fd-predecessor of v_y, that means two events v_x and v_y have no interaction with each other. Menu-open events Save and Save As are such examples. Users can click each event by any order. Any such edge is shown as a dash line in the EDG. On the other hand, if v_x is only a cd-predecessor of v_y, then the event v_y can only be launched after event v_x happens. Thus, these events are represented as solid lines.

Example 1: A Copy-Paste EDG in the Notepad. The copy-paste process in the Notepad can be represented as an event sequence. When users want to copy and paste a word or paragraph, they should highlight it first, and then click `Edit` and `Copy` from the menu option, and lastly click `Edit` and Paste from the menu option. During this process, the highlight event is the ecd-predecessor of the Copy event because users have to highlight a word or paragraph they want to copy, then Notepad will store the word or paragraph in memory when the `Copy` event is fired. For the same reason, the `Paste` event is the ecd-successor of the `Copy` event. Therefore, the EDG can be displayed as Figure 3.3, a `Copy-Paste` EDG in Notepad.

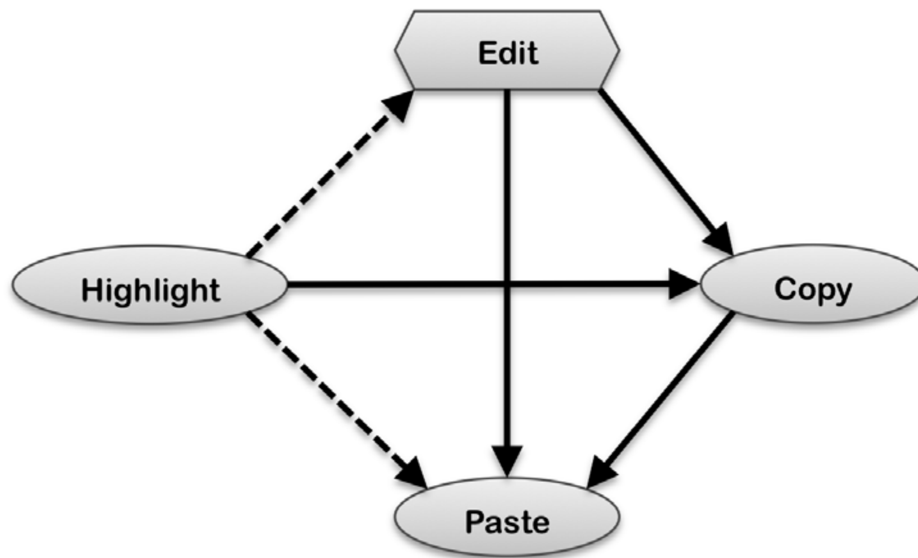

Figure 3.3: A Copy-Paste Edge in Notepad

Figure 3.4 displays two screens, the `Open` screen on the left and the `Save As` screen on the right. In order to edit an existing file, users have to execute a sequence of events. Normally, users have to click the `File→Open` menu option to open the `Open` screen, then select a desired directory in the Look in combo-box, then double click or type the document name, then click the `Open` button to close the screen.

After the file has been modified, users need to click the `Save As` button, select a desired directory in the `Save` in combo-box, type in a new file name, and then click `Save` button. The whole process can be controlled by the EDG in Figure 3.5.

Example 2: An EDG for Editing an Existing File. Figure 3.5 shows the diagram for our second example, an EDG for editing an existing file in the Notepad.

3.1.2 Construction of Event Dependence Graph

This subsection describes how to construct the event dependence graph, which extends the event flow graph defined by [28]. Listing 2 presents an algorithm to construct an event flow graph for a GUI application.

Listing 2 presents an algorithm `ConstructEDG` for constructing EDG. The algorithm takes a GUI component G, and returns Σ, an EDG that contains all events and their dependence relationships. `ConstructEDG` first initializes all events in the V (recall that V represents all events in the GUIs), the initial event set B (also recall that B represents events that are available when a component is invoked), and then constructs EDG (with Initial Event set B) for G. Next, the algorithm calls `FollowDependentSet` for each event node in B. `FollowDependentSet` will construct a set of outgoing edges and set the dependence type into of the EDG . This recursive algorithm contains a switch structure that assigns `FollowDependentSet`(v) according to the type of each event. In this processing, it is assumed that the source code of the GUI application is available to elicit the event information.

If the type of the event v is a `menu-open` event and v is in B then the user may either perform v again, its sub-menu options, any event in B, or the events in ecs-set of v (recall ecs-set includes all events that are event control dependence of v). How-

```
algorithm ConstructEDG(G): Σ
input  G: a GUI component
output Σ: the EDG for G

begin
    Initialize(V)
    Initialize(B)
    Initialize(Ψ)
    for each event v in B do
        FollowDependentSet (v)
    end-for
    Σ  = < V,  E,  B,  Ψ >
end

algorithm FollowDependentSet(v  ∈  V)
begin
    switch (eventType(v))
    case menu-open
        if v  ∈  B
           E  =  E  ∪  {(v,  s)  ∈  E where
               s  ∈ Menuchoices(v) ∪ {v} B ∪ ecs-set(v) is efd-successor}
        else
           E  =  E  ∪  {(v,  s)  ∈  E where
               s  ∈ Menuchoices(v) ∪ {v} ∪ FollowDependentSet(ecp-set (v)) ∪
               ecs-set(v) is efd-successor}
        end-if
    case system-interaction
        if v  ∈  B
           E  =  E  ∪  {(v,  s)  ∈  E where
               s  ∈  B ∪ ecs-set(v) is efd-successor}
        else
           E  =  E  ∪  {(v,  s)  ∈  E where
               s  ∈ ecs-set(v) ∪ FollowDependentSet(ecp-set (v))
               is efd-successor of v}
        end-if
    case exit
        E  =  E  ∪  {(v,  s)  ∈  E where
            s  ∈  B of Invoking component is efd-successor}
    case restricted-focus
        E  =  E  ∪  {(v,  s)  ∈  E where
            s  ∈  B of Invoked component is efd-successor}
    case unrestricted-focus
        E  =  E  ∪  {(v,  s)  ∈  E where
            s  ∈  B ∪ of B Invoked component is efd-successor}
    end-switch
end
```


Listing 2: Algorithm for Constructing EDG

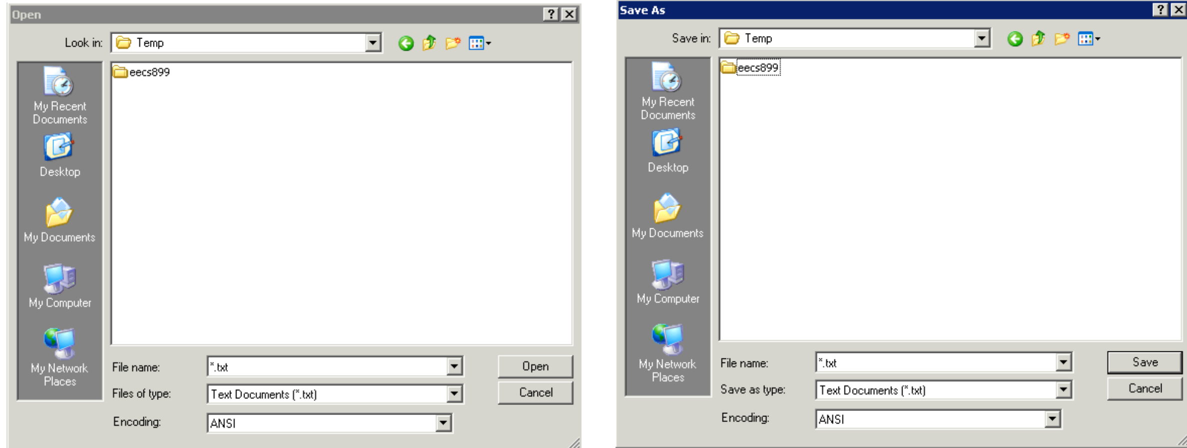

Figure 3.4: `Open` and `Save-As` Screens in Notepad

ever, if v is not in B then the user may either perform all sub-menu options of v, v itself, the event in ecs-set of v, or all events in the `FollowDependentSet(ecp-set(`v`))`. If v is a system-interaction event and v is in B, then after performing v, the GUI reverts back to the events in B or moves to the events in ecs-set of v. Otherwise, if v is not in B, the user may perform the events in ecs-set of v or all events in the `FollowDependentSet(ecp-set(`v`))`.

If v is a termination event, the `FollowDependentSet(`v`)` consists of all events of the invoking component. If v is a restricted-focus event, then only the events of the invoked component are available. Finally, if v is an unrestricted-focus event then the available events include both the events that are available in the invoked component and all events in the invoking component.

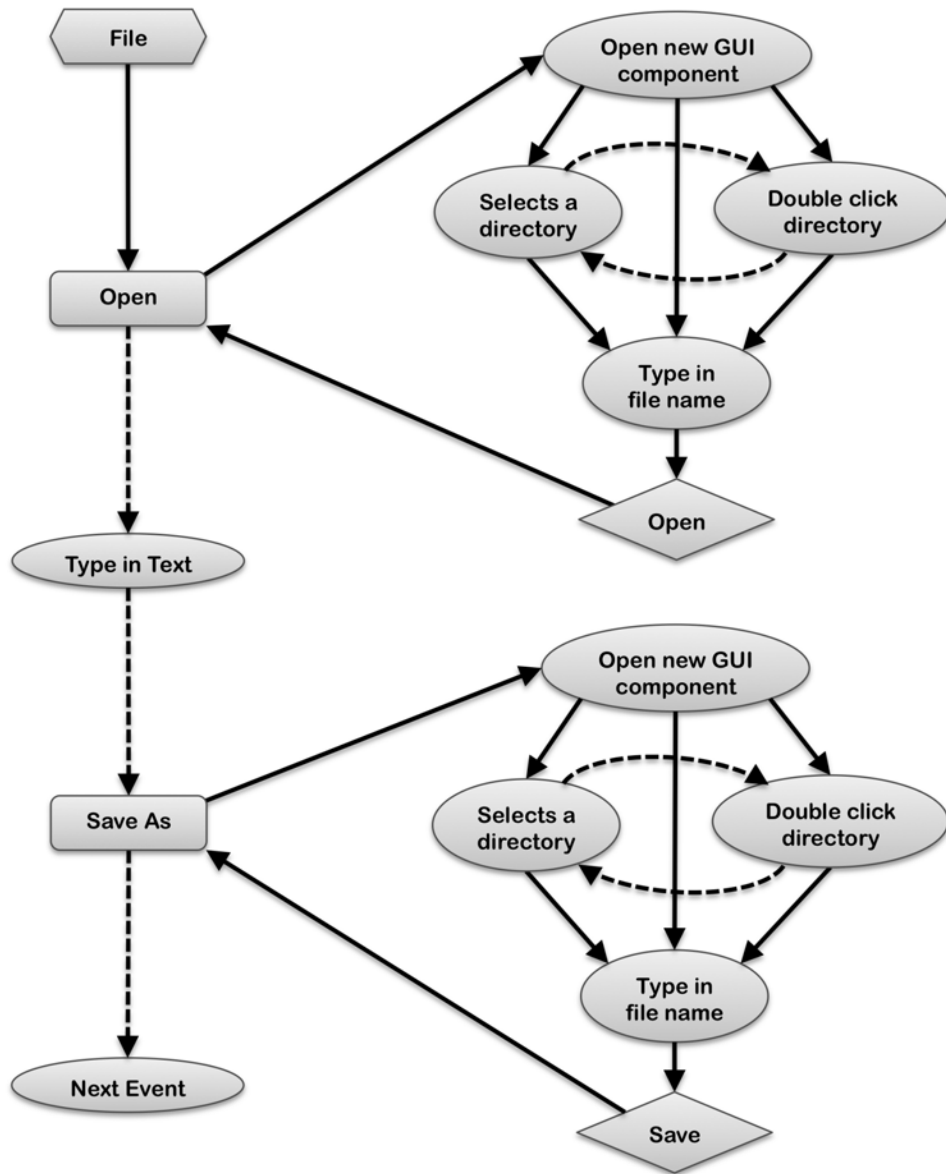

Figure 3.5: An EDG for Editing a File

3.2 Regression Test Selection Techniques for GUIs

We now turn to the problem of testing a GUI component after it has been modified. Rothermel & Harrold [33,37,38] developed general regression testing selection techniques by using control flow graph and program dependence graph for the object-oriented software. Those techniques are applied to GUI applications by using event dependence graph in this section. We will use the information provided in the Sections 2 and 3 to present `EventTestSelections`, the regression test selection algorithm for GUIs. `EventTestSelections` takes a GUI component G, its modified version G', and the test suite T for G, and returns G', a set that contains tests that are modified for G and G', and then an example is used to explain how the algorithm works for the GUI regression testing.

3.2.1 Motivating Observations for Testing a Modified Program

Ideally, after modifying a GUI component, one would like to create an adequate test suite for the modified GUI component (reusing as many old tests as possible), and to run the GUI component on all of the tests in this suite. However, it may be too expensive to run all of the tests. Therefore, a reasonable goal is to run enough tests to guarantee that every affected GUI events of the modified GUI components is exercised. In this case, if a reused test is only known to test unaffected GUI events, that test should not be rerun.

Given the goal discussed above, the following process can be used to test a modified GUI component:

1. A subset of the original test suite is identified for use in exercising events of the modified GUI components.

2. A subset of the tests selected in Step 1 is identified for use in exercising affected events
3. The tester runs the modified GUI component on the tests selected in Step 2; when the component is run, a record is kept of the events that actually were exercised.
4. The tester creates new test cases for the affected events of the modified GUI component that were not exercised by the reused tests.

In order to perform regression testing using an existing test suite, we must have access to test history information, which keeps data on the previous executions of the tests. For the purposes, the test history information includes an event trace: a list of the events executed by each test. Using the event trace, each event in the EDG is associated with the set of tests in T that execute all events during running of the original GUI components. This set is called an event history, whose concept is provided by Rothermel & Harrold [33]. For an event N, N.history represents its event history. Let's use an example to illustrate the event history. The left picture in Figure 3.4 is the screen that allows users to open a file. The users have many ways to open a file `research.txt` in the directory `C:\EECS\899`. Table 3.1 lists five test cases to open the file. Each test case includes a sequence of events. The event double clicking `research.txt` has only $T2$ associated with it while open event has all tests in the test suite associated with it. Thus, the history of event double clicking `research.txt` is $\{T2\}$ and the history of event clicking open button is $\{T1, T2, T3, T4, T5\}$.

The simplest method for selecting tests based on the preceding observations requires a complete mapping of the event nodes in G to the event nodes in G', and information on which nodes in G' enclosed changed nodes. Given such a mapping,

Table 3.1: Test Cases for Opening a File in Notepad

Test Case	Event History
T1	Click Open menu, type text `C:\EECS\899\research.txt` in name field, click `Open` button
T2	Click `Open` menu, select `D` in combobox, double click `EECS` folder, then `899` folder, double click `research.txt` file, click Open button.
T3	Click `Open` menu, select `D` in combobox, type text `EECS\899\research.txt` in name field, click `Open` button.
T4	Click `Open` menu, select `D` in combobox, double click `EECS` folder, type text `899\research.txt` in name field, click Open button.
T5	Click `Open` menu, select `D` in combobox, double click `EECS` folder then 899, type `research.txt` file in name field, click `Open` button.

this method just simply selects all tests associated with the events that enclose changed event nodes. However, such a mapping may be difficult and costly to obtain. We can improve on the simplest method by instead traversing G and G' in preorder and, on reaching an event node whose ecd-successors have changed, selecting all tests that reach that node. Having selected these tests, it is not necessary to proceed further in this traversal; all tests reaching nodes farther in the traversal through this chain of control dependence events have necessarily been selected.

3.2.2 The Test Selection Algorithm

Listing 3 presents `EventTestSelections`, the regression test selection algorithm. This algorithm takes a GUI component G, its modified version G', and the test suite T for G, and returns T', a subset of T to test G'. `EventTestSelections` uses recursive procedure compare to perform synchronous depth-first traversals of G and

G', relying on a "visited" flag attached to each node to avoid revisiting nodes during the traversals. EventTestSelections first marks all EDG event "not visited," and then initiates the graph traversals by calling function EventIdentify with the entry nodes of G and G'.

The EventIdentify procedure selects the tests for a given pair of events in G and G'. Called with a pair of event N and N', EventIdentify first marks them "visited," and then calls function GetRelationship(N,N'). Given any such pair of events N and N', the GetRelationship determines the relationship between two events N and N' in certain status: NoMatch, Added, Equivalent, Modified and Deleted. If N and N' are equivalent, then the EventIdentify function will continue to look for their ecd-successor events and do further compare.

The GetRelationship function determines whether traversals should continue beneath N and N'. Essentially, this approach obviates the need for prior knowledge of modifications, instead locating the changed event as it traverses the graph, and only as needed. Listing 4, The GetRelationship procedure, attempts to establish a "mapping" between a pair of events N and N' and records that mapping in Correspondence. Correspondence is a pair of arrays that list all events as "not examined." As GetRelationship determines mappings between events in G and G' it updates the arrays, recording event nodes that are in G but not G' as "deleted." Event Nodes that are in G' but not G as "added." Event nodes that exist in both G and G' differ as "modified," and event nodes that are the same in both G and G' as "equivalent." In the case of "modified" event nodes, Correspondence also tracks which node in G corresponds to which node in G'. If GetRelationship cannot establish a mapping, it just marks as "no match": all tests through G must be selected. If, however, GetRelationship can establish a mapping, it examines the mapped event nodes. If event nodes are new, modified, or deleted, then GetRelationship returns the cor-

```
algorithm EventTestSelections(G, G', T, Correspondence): T'
input  G, G': base and modified GUI component
       T: a test set used to test G

output T': the subset of T selected for reuse to test G'
       Correspondence: a partial mapping between event nodes in G in G'

begin
   Σ = ConstructEDG(G)
   Σ'= ConstructEDG(G')

   for each event v in Σ and Σ' do
      mark v 'not visited'
   end-for

   for each event N, N' ∈ B, B'
      T' = T' ∪ EventIdentify(N, N')
   end-for
end

procedure EventIdentify (N, N')
input  N, N': events in Σ, Σ'

begin
   mark N and N' 'visited'
   switch (GetRelationship(N, N'))
   case 'Equivalent'
      for each ecd-successor n, n' of N, N' do
         EventIdentify(n, n')
      end-for

   case 'Added'
      T' = T' ∪ {new test set for the event N' in G'}

   case 'Modified'
      T' = T' ∪ N.history // processing can't continue in N & N'

   case 'Deleted'
      for each ecd-successor n of N do
         T' = T' ∪ n.history
      end-for
   end-switch
end
```

Listing 3: `EventTestSelections` Algorithm for Regression Test Selection

responding status, indicating that all tests through N must be selected. However, if neither of these conditions holds, the function returns “no match,” indicating that nothing needs to be changed, and possibly avoids selecting all tests through N.

When `GetRelationship` returns “equivalent,” either there are no changes between two events N and N', or the changes are not sufficient to force selection of all tests through event N. For the later case, `EventIdentify` considers each ecd-successor n of N and n' of N', and calls the recursive function for each child pair. When `GetRelationship` returns “added,” that means N' is a new event that didn’t exist in G. `EventIdentify` will add the related test cases into T'. If its return value is “deleted,” `EventIdentify` will add all test cases in the history for all cds-set for the deleted event. If the return is modified, `EventIdentify` simply adds N.history into T'.

We now consider an example that illustrates the use of `EventTestSelections`. Figure 3.6 shows the picture for `Find Screen` on left and its modified version on right. The modified version of the `Find Screen` includes a new control: a checkbox `Find whole words only` for users to check it when they want to to find only that word. Thus, the right screen reflects the modification. Therefore, developers need to add the checkbox on the screen and modify the `Find Next` button event. All other events on this GUI component remain the same as the left screen.

Figure 3.7 shows the original EDG and its modified EDG shown on the right. The `GetRelationship` function marks the event `Find whole words only` as “added,” and `Find Next` as “modified,” and there is no corresponding event in G for `Find whole words only,` so the new test cases for the new event needs to be added into T'. In this simple example, `Find Next` event is the ecd-successor of event `Type in Word.` `GetRelationship` returns “modified” for event `Find Next` because it should be modified after new event is on the screen. Thus, all test cases in the

```
procedure GetRelationship (R, R′): Relationship Status
input    R, R′: events in EDG G and G′
output   Relationship Status in {NoMatch, Equivalent, Added, Modified, Deleted}

begin
   attempt to match R and R′, locating new, deleted, and modified flags
   if N and N′ are equivalent
      return ‘Equivalent’
   else
      record information on node Correspondence, and on new, deleted,
             and modified nodes in Correspondence
      if N′ modified then return ‘Modified’
      else if N′ is new then return ‘Added’
      else if N is deleted then return ‘Deleted’
      else return ‘No Match’
      end-if
   end-if
end
```

Listing 4: The `GetRelationship` Procedure

history of event `Find Next` in G will be added into T'.

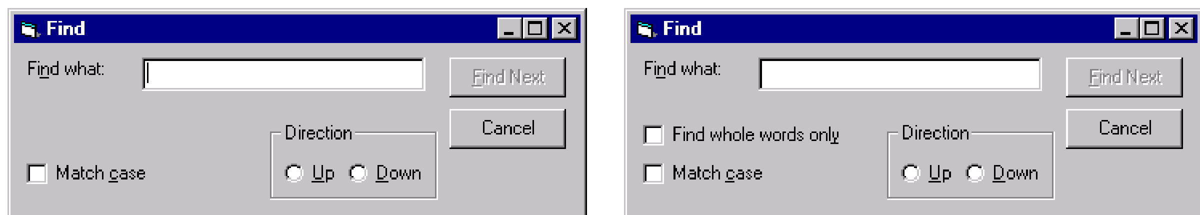

Figure 3.6: Screen Shots of of `Find` Screen and its Modified Version

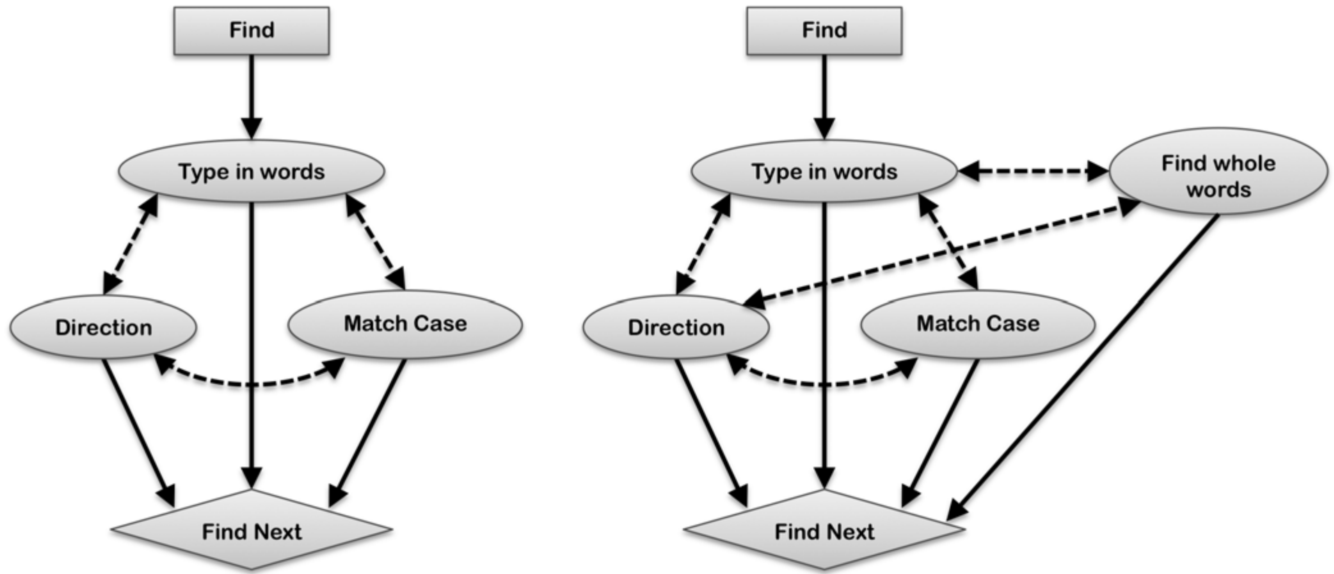

Figure 3.7: EDG for `Find Screen` and its Modified Version

3.3 Summary

In this chapter, the approach of defining and constructing the event dependence graph is presented and the process of regression test selection utilizing this model are explained. Event type classification is presented to first provide an understanding of the different types of events and their importance in the relationships of events. The event types defined are system interactions, unrestricted focus, restricted focus, termination, and menu open. The event dependence graph is then defined which is a hybrid model that has similar characteristics of the control flow graph, program dependence graph, and the event flow graph. From its definition, the algorithm to construct the event dependence graph is presented (`ConstructEDG`). With the event dependence graph defined, the motivation of leveraging a test case selection process against this model is presented. The motivating factors are the core needs which drive regression testing of an evolving software solution. It is explained how test cases are associated to events within the event

dependence graph, and how the entire set of related events is considered the event trace of a test case. Finally the test selection algorithm (`EventTestSelections`), is presented and explains how changed events are utilized to dictate which test cases need to be executed to sufficiently test the impacted events (based off of their relationships in the event dependence graph).

Chapter 4

Research Results Evaluation and Validation

An implementation for the proposed solution will compose of several different components, each of which will have key responsibilities. This implementation is focused on supporting the proposed solution's attribute of being an automatic process of selecting all the relevant tests for a given event change. In order to support an automatic process, each component will perform its responsibilities with minimal human intervention. The details of this implementation will be specific to Java programs, but its structure in design is agnostic to any specific language. The listing of components is as follows:

1. **Data Store** A persistent data store will exist to maintain a relationship of test plans which relate to a given event of an application. This data store only requires storing test plan relationships to events, and does not necessarily store details of the test plans (i.e., test steps, expectations). This will capture all initial test plan / event relationships, as well as, new test plans or events which are created during the maintenance phase of the application.

2. **Ripper** An external component from the GUITAR framework [21, 24, 39] which will execute the GUI application and performing a ripping process which will extract GUI components and construct an event flow graph.

3. **Analyzer** Applies the proposed solution algorithms to determine all the dependent events for a selected event. The selected event can either be selected by the human who is initiating the program or can be elicited during the scannerÕs phase to determine what changes have occurred based comparing two different version of the program (G and G').

4. **Report Generator** With the selected events which are dependent on the event supplied to the analyzer, the related test plans need to be selected from the data store and presented so the consumer is aware of what tests have been selected to be executed.

The relationship of these components are listed in Figure 4.1. The workflow of data through these components begins with the specifications of how to perform the GUI ripping process for the application under test. The GUI ripper then executes its process and collects the GUI component and event information and presents this in an output file that is in an XML format. This output file is then consumed by the analyzer which constructs the event dependence graph based on the GUI component and event information presented by the GUI ripper. The output of the analyzer (file which constitutes the structure of the event dependence graph) can then be consumed by the report generation tool which can then select test cases. This selection process is based on the event dependence graph and the events that have been specified as being changed. The output of the report generator will list these test cases, which a software tester could then use as their test suite for regression testing of change set to the software.

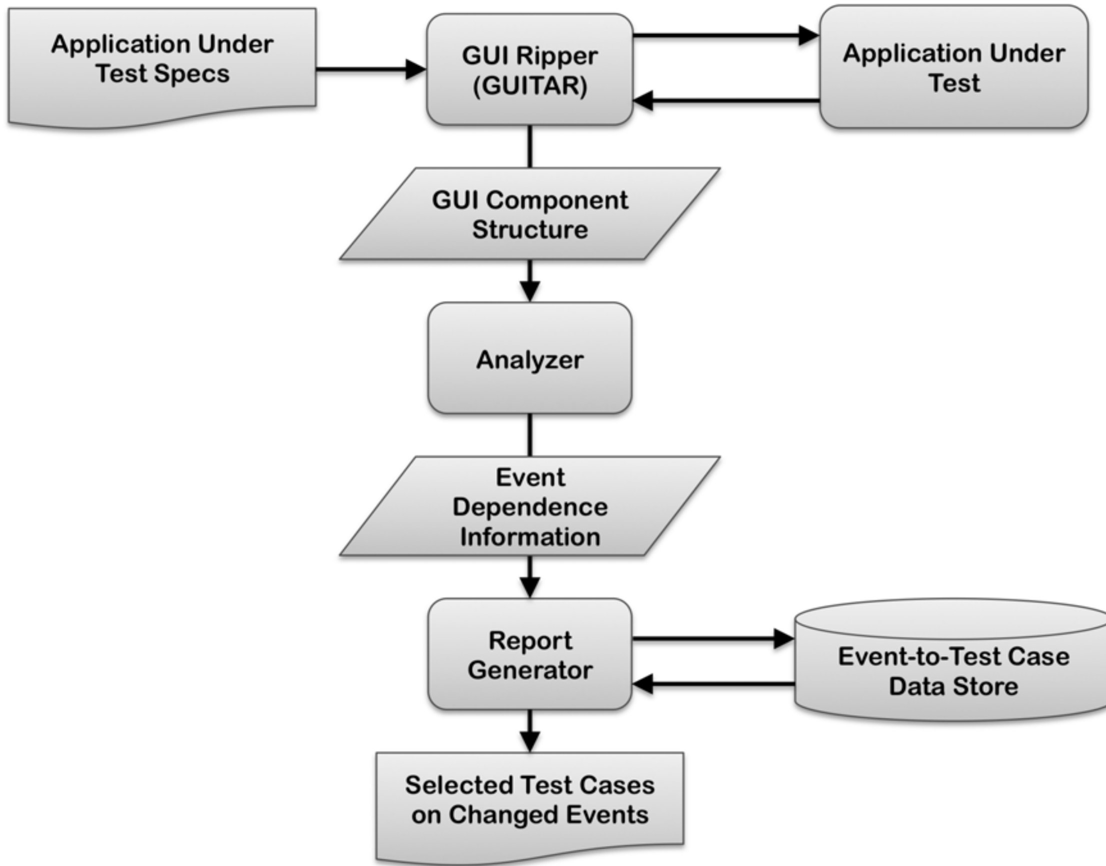

Figure 4.1: Overview of Implementation

4.1 Static and Dynamic Analysis of GUI Components

When assessing how to extract GUI event information from the application under test, two different approaches were considered: static analysis and dynamic analysis. The first approach that was considered was static analysis, since it was expected that the tester would have access to the source code of the application under test. Since the focus of this experiment is using Java technologies, it was assessed whether static analysis could be performed for event information extraction using the compiler API which is present in the Java Standard Development Kit version 6. With this API, it would be capable of extracting types of classes which represent

specific known Java GUI components and action listeners. However, it can be challenging when utilizing this compiler API against a large application with complex structures to determine all the inter-relationships of types. A critical challenge exist with this approach where it is not apparent what all other GUI components are available to be invoked based on the static structure of code that represents the application. For example, it was capable of determining that a given GUI component (i.e., button), when clicked, would open a window. However, it wasn't clearly evident that from that window which was opened, what other GUI components (possibly in the background) were capable of being invoked while that window was present. Therefore, the extensive cross relationships of components would not be easily captured without analyzing this during runtime. State information of the application would also dictate the behavior of these GUI components (i.e., if a button was enabled or not). This further made static analysis challenging in accurately capturing the possible event relationships since the potential state information could not be inferred. From these conclusions when assessing static analysis as an approach for event extraction, it became apparent that dynamic analysis would be a more accurate method for obtaining the desired event information.

When assessing the feasibility of using dynamic analysis of the GUI application, it became apparent that certain attributes must exist in the tooling which is leveraged to perform the analysis. The key attributes were: automated execution, extensive state extraction, and capable of identifying events.

During dynamic analysis, the application is put into motion by starting it in a running state and invoking functionality until it ultimately terminates (or reaches some targeted state). In order to trigger the GUI application to go into a running state and invoke all possible events (so an accurate analysis can be performed) requires an automated tool to efficiently accomplish this. Hence, it is not desirable

to utilize any tooling which requires human intervention to exercise the GUI application in a running state to extract information. The driving force which would navigate the application during this run-time must be a consistent and timely process. If the process was not automated, several challenges would exist:

1. **Variance in execution process** If a manual process was applied every time the GUI application was analyzed for event information, there would be a possibility of variance in how that GUI application was executed. This variance would drastically cause issues in the quality of the analysis, as many decisions are based on that event information which is extracted. If an event is missed, it would be determined that an event was deleted from the application, which cascades into triggering many tests cases to be utilized in testing when that should not be the case. This would degrade the quality of the test case selection, as it would not be accurately restricting the number of tests to be executed from the set which would be utilized from a retest all approach. In order to compare against previous analysis, the same process of performing that analysis must be followed to remove the chances of variance (which could injected into the process if it were performed manually by a human).

2. **Costs increased due to time required** The goal of this solution is to provide a minimal listing of tests that can be performed for regression testing and are considered safe. If the analysis is a lengthy process, it would add costs to the process and ultimately to the overall costs of performing regression testing. Therefore, we must try to avoid any additional costs when performing this analysis. One key challenge that exists is that when performing analysis on the runtime state of an application, the time progression which an application state would reside can be a valuable element in the tests when time

based event triggers exist. For example, if an event fires every five minutes to alert the user of something (i.e., displays a modal window), the analysis process may miss these events. This is a known limitation of performing the analysis, where events that are not invoked by GUI components may not be captured (as they may not easily be triggered). In any case, the analysis will be performed in a consistent manner to ensure comparisons are accurate when performed. Meaning, if a time based event will not be captured due to time taken by the dynamic analysis being performed, then it will not be captured on any subsequent analysis which will be used to compare previous states.

3. **Inaccurate analysis** With performing dynamic analysis, we need to capture state information closely to the time which the event is being invoked. If this were to be applied in a manual fashion (i.e., utilizing a debugger) it would be a very costly and challenging task to ensure state information is accurately paired with the time which the event occurred. By leveraging an automated means, you can accurately listen for the state change of the application when changing its state during the its runtime. This is due to necessary synchronization of checking state with the invocation of the GUI event.

When analyzing the GUI application at runtime, it is necessary to extract a rich amount of information about its state. This is desirable as this information is expanded upon and used to identify each GUI component within the application. With comprehensive applications, a large breadth of events are possible and it is required to be capable of uniquely identifying each one of those events to perform analysis in comparison to the previous event dependence graph. Therefore, identifiable attributes are necessary to be extracted. With those identifiable attributes, their value at the point of extraction (their state) is needed be known, as that data dictates an

event's relation to a given component.

An event is identifiable based on the context of what it relates to and what it supports. Its relationship information to GUI components (i.e., buttons, labels, panels) provide sufficient identifying marks of its source of invocation. The type of event and its resulting behavior provided added uniqueness to how that event can be identified in the graph. For example, the source of the event is a menu-item in the main application window that is called "Search", which when clicked, invokes a "Search" window (restricted focus type of event). All of this information provides an identity for that event which makes it unique within the host program.

The analysis must be capable of identifying events, which means it must be aware of when an event occurs. To achieve this, the process which is analyzing the application must know and listen for specifics in alterations of the application which may not be apparent from the visual state of the program. For example, I may have an event which is triggered based on a mouse hovering over a particular area of window pane. That event may not change anything in terms of visuals which are present in the application, but may trigger some type of system interaction to occur. These types of events make it exceptionally difficult of knowing all the different areas to invoke to trigger such events.

With these supporting reasons to utilize dynamic analysis, it was further investigated into what tooling exists that would support this need. The approach of utilizing runtime information of a Java program to support test case selection is an area which has been investigated in other research [16]. With the rich tooling support of the Java Virtual Machine, several options exist in obtaining runtime information of a program when exercising its functionality. The next section will further explain the GUITAR framework which was chosen to be utilized in this experiment for extracting runtime information.

4.2 GUITAR Framework Overview

Automated event identification is a challenging task. When assessing options to implement this mechanism, the approach of GUI ripping from the GUITAR framework was evaluated [15, 24]. This approach performs its analysis at runtime of the application (rather than static analysis). Therefore, it must exercise the application through the events desired to produce the listing of GUI components of the application. It is desired to leverage this existing tooling in order to extract GUI components of the application to assist in constructing the event dependence graph.

GUITAR is a testing framework which offers several valuable components when evaluating a GUI application to test. As mentioned earlier, it is capable of ripping a GUI application to obtain information of its structure (represented as an XML file with a `*.gui` file extension). It can take this information and build an event flow graph (which is listed as XML in a file that has a `*.efg` file extension). Within this file, is defines the EFG through an adjacency matrix. This file can then be used to generate graphs which are constructed in the Graphviz **.dot* format to illustrate the directed graphs. Furthermore, test generation can be fueled by this data, as well as, replaying the desired events for the test cases.

4.3 Apache JMeter Application Assessment

Apache JMeter is an open-source Java application that allows users to load test their applications [18]. It supports load testing on many of the core server types, such as: RESTful and SOAP web servers, database servers through JDBC, and LDAP servers. It was desired to apply our experiments against this application, as it has several attractive traits:

1. **Identifiable** The application is well established in the development and testing communities and therefore details of the experiments may be well understood based on the audience's general knowledge of the application.

2. **Relevant** The initial version of JMeter (1.0) was released in 1998. This application has a large history of changes and it is clearly in the maintenance phase of its existence. Thus, by using this application in our experiments, it will exhibit similar challenges of analyzing and identifying test cases in a established software solution. The purpose of the experiment is to show the effectiveness of the proposed solution and doing so on a relevant scenario (i.e., analyzing an existing software project to identify test cases based on an event change to the application).

3. **Adequate Complexity** JMeter is presented in a fairly simple GUI layout, but does offer complex functionality. It was desired to experiment with an application that did not have an excessively complex GUI layout so that the description of the steps in the experiment are not plagued with the complexity of the application. In addition, experiments could be easily scoped to individual event flows which are of a digestible size when explaining their relationships in an *EFG* and *EDG*. Furthermore, the application has complex system interactions (i.e., starting processes to load test a remote service) which can present realistic and relevant challenges when performing analysis of of the application at run-time to extract event information.

4.4 Graphical User Interface Ripping Process

The first step to construct the EDG is to extract the event information from an application through the GUI Ripping mechanism in the GUITAR application suite. There are several different flavors of the GUI ripping functionality based on the type of technology being used to support the GUI. In our case, we are testing a Java application and therefore will use the JFC (Java Foundation Class) GUI ripper. The ripper is a command prompt application which takes in several different arguments. In Table 4.1, the specifics of the command line options are listed which were utilized to execute the GUI ripper on JMeter.

When performing the ripping process, it is possible to be an exhaustive process due to all the potential events that are being captured. The output of the ripping process is an XML document which is identified by the GUITAR framework with a *.gui file extension. This file contains all the GUI component information which is organized by containers. GUI windows are core types containers that are the root containers for GUI components. These containers (which are identified as a `Container` element in the XML schema) are nested structures which can further expand to indicate reachable GUI components from parent containers. In each container, a generic name/value pair structure exists (identified as a *Property* element). There can exist variable number of properties one GUI component may have, all of which are hosted in the *Attributes* element of the XML schema. Within each property, some of the key attributes which are obtained:

- **ID** Identifier of the GUI component. This attribute is a valuable piece of information which is further leveraged to identify events.
- **Size** Size is specified by its overall width and height based on pixel spaces.

Table 4.1: JFC GUI Ripper Arguments

Argument	Description
`-c`	Main Java class under test. The first thing which needs to be identified is the *main class* of the Java application. JMeter, which is bundled as a JAR (called `ApacheJMeter.jar`), has this information bundled in the manifest of the JAR. Within the manifest (`MANIFEST.MF`), it identifies the main class as `org.apache.jmeter.NewDriver`
`-cf`	Ripping configuration file. Within this file, we can specify what components we wish to ignore and which components are terminal. For the initial run (to hit expand upon all possible event states, nothing was ignored, and only the "Exit" option of the main menu was listed as a terminal component.
`-cp`	Classpath of the Java application. For this particular case, this was a large input as JMeter had 60 different JARs included in its *lib* folder which needed to be included in the classpath. All of these JARs may not have been required, but it was assumed that they should be included to support a complete execution of the application. This argument includes the full file path to each of these JARs, and that listing is the delimited by a colon character when executed in a Unix system (or a semi-colon when executed on a Windows system).
`-d`	Delay (in milliseconds) after each GUI component is triggered before activating another. This configuration was desired to be applied to ensure adequate time was applied for other events that could be loaded which may have just been delayed.
`-i`	Initial warm-up time (in milliseconds) of the application prior to allowing the ripping process to begin.

- **Location** Location of a GUI component which is specified by a horizontal (x) and vertical (y) coordinate.

- **Class** Type of GUI component based on the languages type of component. For

an example in Java Swing, a window panel would be a `javax.swing.JPanel`.

- **Visible** Boolean indicator to signify if that GUI component is visible in the current view.
- **Enabled** Boolean indicator to signify if that GUI component is enabled. For example, a button may be disabled for certain states of a given form.
- **Focusable** Boolean indicator to signify if the GUI component is in focus in the current view.

Other attributes are available about components; however, most of them are not that critical when determining event information of the application. Within the XML contents, one can recursively search into a container to find other GUI components. This is the approach that will be further explained in the implementation notes of how to handling the GUI information which the GUITAR ripping process produces.

For these experiments, the machine which is being utilized has a 2.4 GHz Intel Core i7 processor, 4 GB 1333 MHz DDR3 memory, running on the Mac OS 10.6 operating system. The time cost on ripping the JMeter application, on average, took 116.218 seconds (when using a 500 ms delay between GUI component activation). The results of the ripping process can be viewed in Table 4.2. With this listing of GUI components, the events were later assessed with the GUI to EFG tool (`gui2efg`) in the GUITAR suite, where it was recognized to have 260 events. When represented in the EFG, there were 260 nodes which had 6969 edges. In Table 4.2, it lists the summary results by GUI component type which was extracted from JMeter.

The results of the ripping process brought forth light to the complexity of event relationships which exist in the application. By having 6969 edges that are established between the 260 events in the graph, it showed that many events were reach-

able from other events. Therefore, a single change to one event, had the potential of impacting many events. The number of relationships seemed high; however, it is still only 10% of the potential maximum edges (V^2) in the graph and therefore could be seen as sparse. Since the number of event relationships was below the potential, our test selection process can restrict the number of tests that what the retest all method would select.

One of the important factors described earlier in assessing a dynamic analysis approach, was to have a timely response rate in the automated execution. The process to performing the GUI ripping was executed in a loop of 20 times to determine an average time cost in performing the analysis. In Table 4.3, a listing of the times are listed. In Table 4.4, a listing of the minimum, maximum, and average are listed.

When performing iterations of the same analysis of the application, differences in the output were assessed. One noticeable change in the output was investigated to be a different window placements during its execution. If a GUI component were to show a window in a different placement from the previous execution, different coordinate attributes would exist, which would impact how that component is identified in the application. This could be accomplished by intercepting the ripping process and moving a window while it is capturing information. By discovering this finding, it became critically known that the process which is performing the dynamic analysis must be executed in complete isolation. If anything invaded this process during analysis, it would disrupt the opportunity of being able to correlate events from this extraction to a previous extraction. This is due the alteration of the identifiable attributes (placement based on x and y coordinates) of the GUI component, and would then affect the identification of the event (appear as a different event).

Another factor of isolation is the host environment (i.e., OS) which the ripping

process is being executed. When ripping the application, if GUI components are to view state of the OS (such as the filesystem), different flows will occur based on differences of state. For example, if you run one rip of the application, and it analyzes the “Save As” window, it may invoke the “Save” button, which creates some file with a default name during the ripping process. If you were to run another rip of the application again and it were to analyze the “Save As” window and invoke the “Save” button, it may invoke a new error window instructing that this file already exists, or something informational that you are about to overwrite an existing file. Therefore, there are points of application where state can be inherited from the hosting environment. This can expand to settings or configuration files which support some type of change to the software, and therefore a change in that file between executions of the ripping process would alter the state of the application and further change what the ripping process could determine during its analysis. A generic approach may be applied, where the application under analysis would be executed in a virtual machine host. Once the ripping process is completed, the state of the virtual machine would rollback to its state prior to execution of the ripping process. This can be accomplished through the concept of snapshots which are generally offered with many of the software virtualization solutions. This greatly reduces the complexity of work for the individual trying to perform analysis of the application. Without this generic approach, they would need to be concerned about the specifics of what could have changed, which would require a deep knowledge and understanding of the implementation details of the application.

When assessing the accuracy of all the events which were extracted, it was noticed that some associations may not exist due to the reconstruction of menu-items during run-time when state changed of the GUI application. An example of this in JMeter is the ability to add different components to a workbench (which is used to

exercise tests). When adding a component to the workbench, the state changes for having a component available for adding. The added component is built into a tree-view pane and the added component is then selected in that view. By having that component selected, the menus change on what is available. In the base JMeter ripping of the application, it would add a component, which was a `Property Display` from the `Non-Test Elements` menu, and then subsequently would try to add another component. However, this state change of adding the component would not allow adding other types as the `Edit` menu would change to no longer have the `Add` menu-item (illustrated in Figure 4.2). During ripping, these menu-items are considered part of the frontier of GUI components to invoke; however, they no longer capable of being invoked for that current state.

Based on these findings, it was then assessed of scoping the analysis for given flows by using ripping configurations. In the GUITAR ripping framework, it allows configuring the ripper by specifying what elements (by name) to ignore. This allows for more control over the ripping process by dictating what elements should should be avoided if specific workflows cause early termination or block the ripping process from hitting all desired areas of the application.

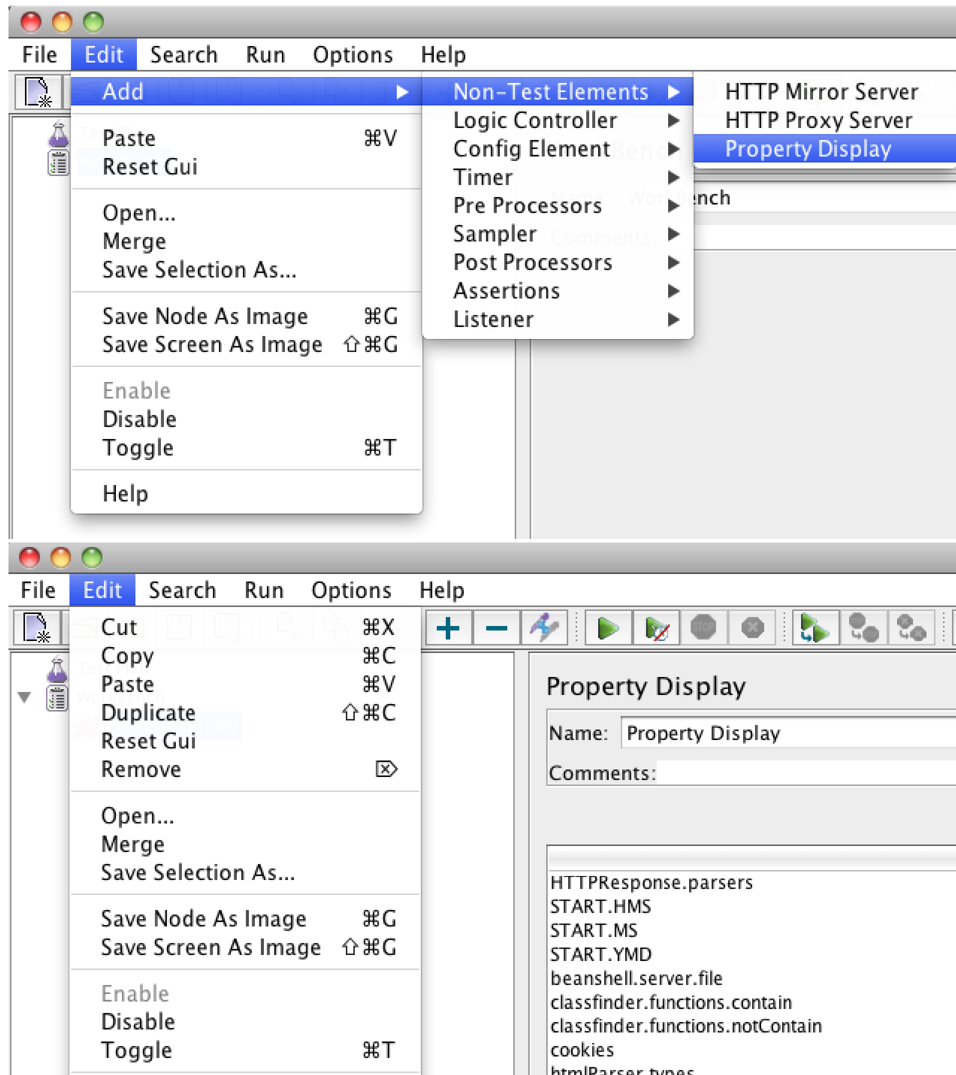

Figure 4.2: Initial Edit menu options and Edit menu options after state change

Table 4.2: JMeter GUI Ripping Results

GUI Component Type	Occurrences
com.apple.laf.AquaFileChooserUI ScrollPaneCornerPanel	2
com.apple.laf.AquaFileChooserUI JSortingTableHeader	2
javax.swing.JCheckBoxMenuItem	1
org.apache.jmeter.gui.MainFrame	1
javax.swing.JDialog	4
javax.swing.JLabel	13
javax.swing.JButton	36
javax.swing.JScrollPane ScrollBar	8
javax.swing.plaf.basic.BasicOptionPaneUI MultiplexingTextField	1
org.apache.jmeter.control.gui.WorkBenchGui	1
org.apache.jmeter.gui.util.VerticalPanel	2
javax.swing.JOptionPane	2
com.apple.laf.AquaFileChooserUI$JTableExtension	2
javax.swing.Box Filler	28
javax.swing.JScrollPane	4
org.apache.jmeter.gui.util.JMeterToolBar	1
org.apache.jmeter.gui.MainFrame$3	1
javax.swing.Box	3
javax.swing.JLayeredPane	5
javax.swing.JMenuItem	184
javax.swing.JComboBox	4
com.apple.laf.AquaSplitPaneDividerUI	1
javax.swing.JSplitPane	1
javax.swing.JTextField	3
org.apache.jmeter.gui.util.JMeterMenuBar	1
javax.swing.JFileChooser	2
javax.swing.JRootPane	5
javax.swing.CellRendererPane	9
com.apple.laf.AquaFileChooserUI 6	2
org.apache.jmeter.gui.CommentPanel	1
javax.swing.JToolBar Separator	7
javax.swing.JMenu	22
javax.swing.JSeparator	1
javax.swing.JPanel	45
org.apache.jmeter.gui.NamePanel	1
javax.swing.JTextArea	1
javax.swing.JPopupMenu Separator	10
javax.swing.JViewport	6
com.apple.laf.AquaComboBoxButton	4

Table 4.3: JMeter GUI Ripping Timings (in seconds)

Iteration	Elapsed Real Time	User CPU time	System CPU time
1	113.619	97.116	0.687
2	113.235	96.994	0.712
3	113.098	96.651	0.677
4	142.996	96.571	0.713
5	113.665	97.096	0.716
6	113.127	96.720	0.707
7	113.566	97.112	0.735
8	113.692	97.415	0.733
9	113.155	96.803	0.714
10	111.913	96.065	0.568
11	112.312	96.286	0.565
12	113.703	96.680	0.578
13	114.093	96.712	0.695
14	113.082	96.693	0.718
15	113.082	96.827	0.690
16	114.156	96.809	0.697
17	113.116	96.624	0.691
18	143.500	97.041	0.714
19	113.122	96.023	0.556
20	112.132	96.193	0.572

Table 4.4: JMeter GUI Ripping Timing Summary (in seconds)

Calculation	Elapsed Real Time	User CPU time	System CPU time
Minimum	111.913	96.023	0.556
Maximum	143.500	97.415	0.735
Average	116.218	96.722	0.672

4.5 Event Dependence Graph Construction

The construction of the EDG can be viewed in two stages of processing: parsing the output from the GUI ripping and analyzing this data to identify event relationships to construct the graph. The parsing phase will first be described

4.5.1 Parsing the EDG output from GUI Ripper

When constructing the EDG, the input from the ripping process must be handled. The format of this data is XML, and there are various types of XML parsers which can be utilized in Java. However, when investigating the uses of the GUITAR framework, it was discovered that this functionality is available in one of their library artifacts. The gui-model-core JAR, which is utilized by GUITAR, has modeled entities which support parsing the XML that is produced from the ripping process. The implementation for the experiment then used this library for its parsing of the XML into its model objects, and processed the model object for its graph representations. An initial test of costs on parsing this data was applied based on the results of the initial ripping process. To gain an accurate cost of time to parse the data, two different approaches where applied in gathering data on timings. One approach was to simply parse a file multiple times in a loop in the Java Virtual Machine (JVM) and measure the time taken (in milliseconds) of each iteration in the loop. The second approach parsed the file and captured the time taken in the JVM, but executed each iteration in a fresh JVM. The second approach was applied when the data was initially assessed from the first approach. It became clear that forms of caching were occurring in the JVM; as the initial times were high, but immediately dropped on secondary executions of parsing phase when executed in the same JVM host. This progression in performance times is illustrated in the line chart of Figure 4.3. When

executing the parse in a fresh JVM instance each time, the times were more consistent with the time of the first iteration in the earlier approach. From this data (listed in Table 4.5), it was apparent that the parsing phase was relatively expensive, as its average time to process the initial ripping output of JMeter was 538.25 milliseconds.

Table 4.5: JMeter GUI Parsing Times of Ripping Output (in milliseconds)

Iteration	**Execution time (single JVM)**	**Execution time (individual JVMs)**
1	512	538
2	105	527
3	89	536
4	66	540
5	83	533
6	68	542
7	69	533
8	67	539
9	63	538
10	59	531
11	70	555
12	61	535
13	61	540
14	63	540
15	62	526
16	62	530
17	60	538
18	56	585
19	57	526
20	54	533

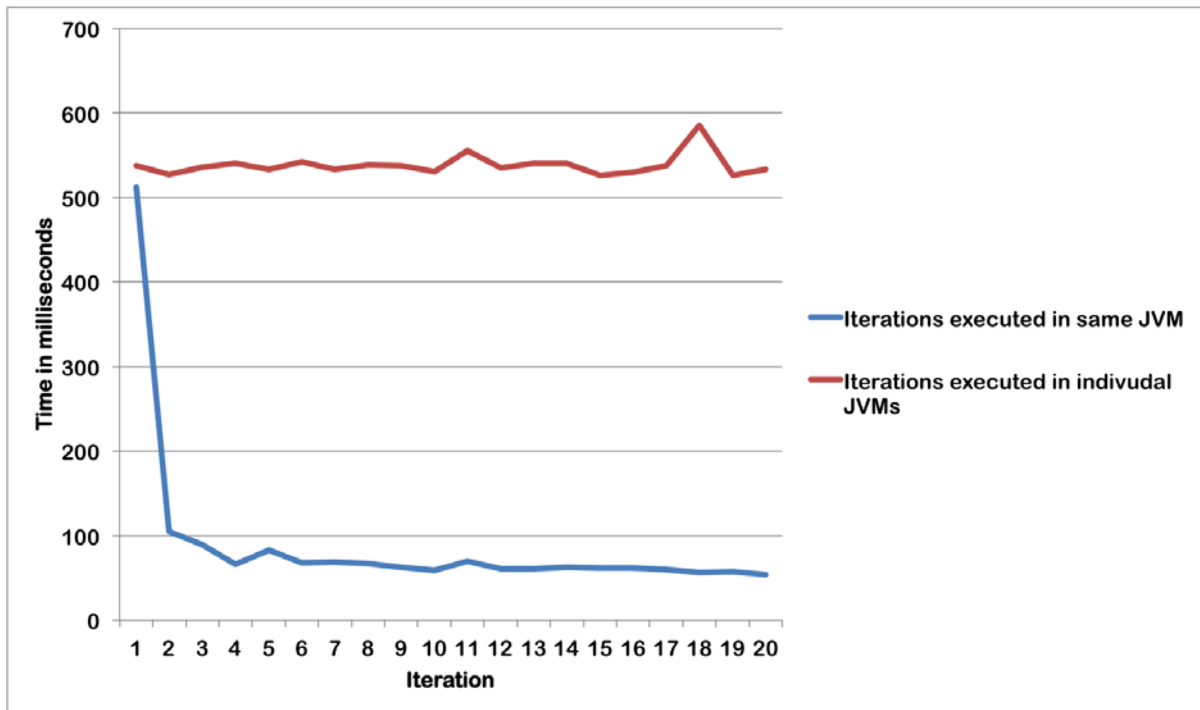

Figure 4.3: Graph of time taken for GUI event parsing

4.5.2 Analyzing GUI model to identify events in Event Dependence Graph

Once the GUITAR GUI model was constructed from the XML, it was matter of then analyzing and traversing through its model to construct the event dependence graph. In the GUITAR model, the core elements to begin inspection, are the windows which have components that initiate other components through events. Therefore, the main window of JMeter is the first element of analysis, and then other windows are listed, which are identified by GUITAR as being invoked from another event.

In Table 4.2, a listing of GUI component types were identified from the application. These are simply the Java Swing types which are used to represent the concepts. During analysis, it was desired to build a mapping of these types so the output would be more understandable by listing the logical GUI component

type (i.e., button) versus the Java type (i.e., `javax.swing.JButton`). This mapping of types was based on the Java package summary documentation of all the Swing data types (`javax.swing` package). With this mapping, it then allowed for specific GUI component types to be modeled and constructed differently. The GUITAR GUI model objects are generic in nature and have a listing of attributes that can be queried, but this type of model causes complexity in the analysis. The complexity is caused from exposing generic types throughout the stack of processing, and each tier in that stack must treat it generically, and check for a larger array of attributes to fully determine its type. By constructing the GUI component objects differently by type (through a factory pattern), it greatly simplified the downstream processing of listing the logical representation of the GUI component associated to an event. A simple example of this, is a Java Swing button. From the GUITAR ripping output, this would have a type of `javax.swing.JButton` and would may have a title, and possibly an icon. When constructing a GUI component from the parsed GUITAR ripping output, this object type would have a enumeration of `BUTTON` and its description would first utilize the title attribute (if available), else it would use the icon attribute (which is the name of the image used for the button). From the evaluation of JMeter, the names of the icons where typically descriptive. As a result, it was then possible to query the EDG for events which are tied to a GUI component based on type and description. This became a common activity when validating the construction of the graph and a useful tool as it was more natural way of seeking events (in this case action listeners) within the functioning program based on the GUI component that was initiating that action.

In constructing the EDG, timings were taken (similar to the steps of parsing the GUITAR ripping output) to validate the general performance cost relative to other steps of the process. These timings were taken on constructing the EDG on the

same GUITAR ripping output of the JMeter application. The process of performing these tests first parsed the contents of ripping output into the desired model object, started the timer, constructed the EDG from that model object, and then capture the timing. Therefore, timing was only around the construction of the EDG; however, object references were leveraged from the model object of the parsing for the EDG construction, so the cost was dramatically lower. Twenty executions of the EDG construction were performed, and each iteration of that test was performed in a new JVM. The average time to construct the EDG was 44.9 milliseconds. The times and variance between execution can be viewed in Figure 4.4.

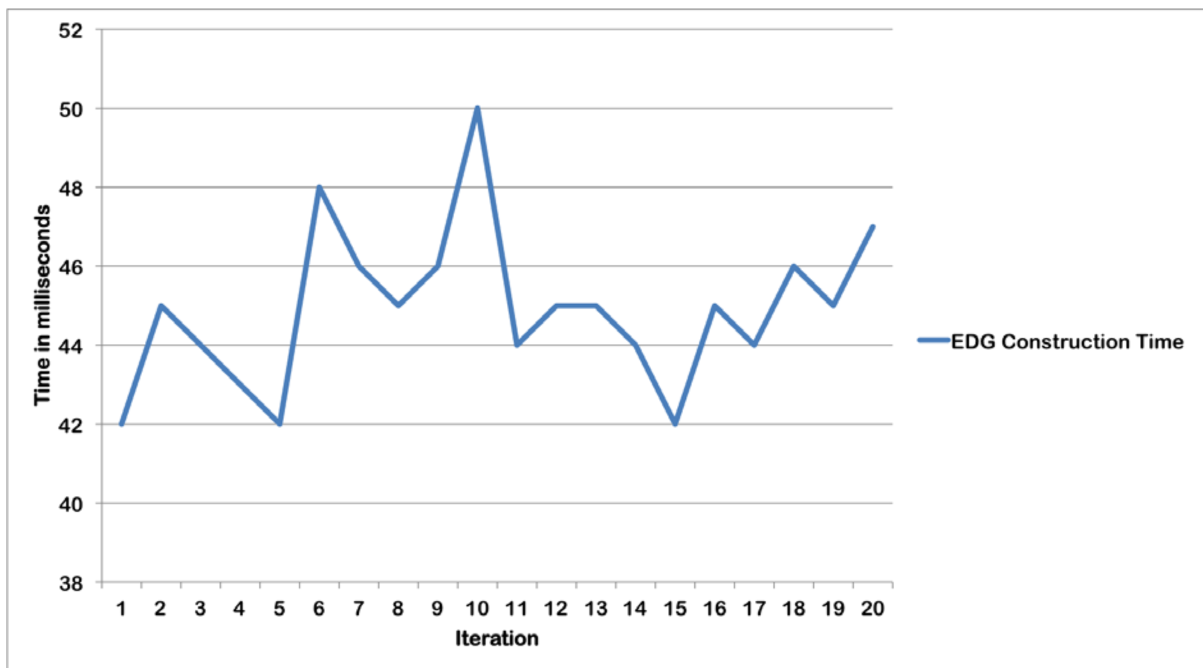

Figure 4.4: Graph of time taken for EDG construction

4.6 Test Case Selection Data Model and Process

The test case selection portion of the evaluation requires designing a data store which will support the needs of selecting tests associated to events in an application. Since mapping of events to test cases (and events to applications) was highly relational, a relational data model was utilized. A basic model was devised to have six entities:

- `GUI_APPLICATION`: Represents the GUI application which is under test. One entity would exist for a single application as it evolves over time. By having this record tied to a given application, it allows for reports which are application specific and across versions.

- `GUI_APPLICATION_VERSION`: Represents a version of a `GUI_APPLICATION`. There can be many versions for one application. This entity is a core entity which supports tracking event changes over time of the application.

- `GUI_APPLICATION_VERSION_EVENT`: Represents a GUI event which is associated to a GUI application version. This entity serves as an associative entity to resolve the many-to-many relationship between `GUI_EVENT` and `GUI_APPLICATION_VERSION`. A single GUI event may relate to many or all versions of an application and a version of the GUI application may relate to many GUI events.

- `GUI_EVENT`: Represents a GUI event of the application. This entity is a unique instance of an event which relates to one or many versions of the application.

- `GUI_EVENT_TEST_CASE`: Represents a relationship of a GUI event to a test case. This associative entity exists to resolve a many-to-many relationship between

GUI_EVENT and TEST_CASE entities. This fact of the data modeling is that a single GUI event may be associated to many test cases and a single test case may be associated to many GUI events.

- TEST_CASE: Represents the test cases which have been already created for applications. A test case will relate to one or many GUI events.

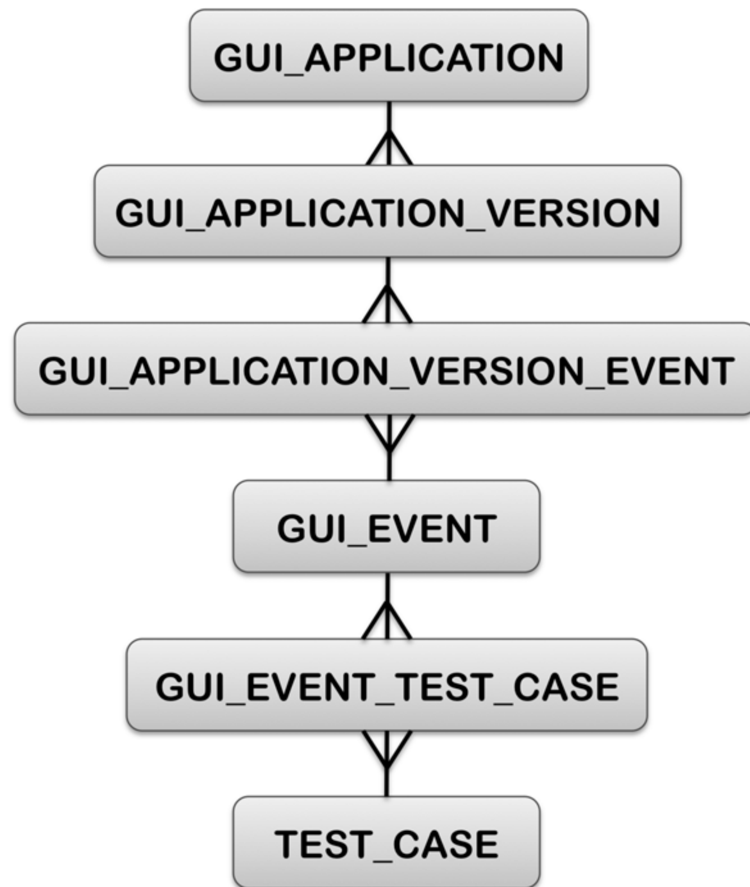

Figure 4.5: GUI Test Case Entity Relationship Diagram

In this data model (illustrated in Figure 4.5), it was desired to have a focus on establishing a listing of GUI events which are extracted during the ripping process and associate them to a specific application version. These events would then be associated to specific test cases. With this data model, one may then be able to

determine several key facts: what events are associated to a GUI application version, what events have been added/removed between application versions, what test cases relate to a specific set of events, how many test cases are supported a given application version (or all versions). In Table 4.6, an example listing of data is provided to illustrate the relationships of data.

Table 4.6: Conceptual Example of Data

Entity	**Data**
GUI_APPLICATION	{JMeter}
GUI_APPLICATION_VERSION	{2.5, 2.5.1}
GUI_APPLICATION_VERSION_EVENT	{(2.5, E1), (2.5, E2), (2.5.1, E1), (2.5.1, E3)}
GUI_EVENT	{E1, E2, E3}
GUI_EVENT_TEST_CASE	{(E1, T1), (E2, T1), (E3, T2), (E3, T3)}
TEST_CASE	{T1, T2, T3}

For the purposes of this experiment, the focus was hosting all of the event information for the targeted application (JMeter) in the data model, and establish a set of test cases related to the application. Of the test cases, it was desired to build test cases to support our experimental needs. The experiment is to focus on one area of the JMeter application (Search window), and have an adequate listing of test cases which are associated to events of that window and to events which are related in the event dependence graph. In addition, test cases which are not related to these events will also be included. Throughout the experiments, the implementation will be tested by specifying changed events which are related to the Search window events and ensure the expected set of test cases which have been documented for all related events are chosen. During the assertion of these results, it will also check that other test cases which are known not to be related to these events are not selected.

The implementation of the relational database used in the experiments is the Apache's Derby database [9]. This solution was used for its ability to represent a relational database which could support a larger scale use, but offers the ease of hosting an embedded database in the program. The embedded database simplified the work of deployment for utilizing the application. The supporting files for the database were just hosted in a subdirectory of the application's run-time directory. When assessing the test case selection based on changes to a specific set of GUI components (within JMeter's search window), data was desired to be generated for all events in the graph. A test case generator was built which generates three tests for each event in the graph. The name of the test is a generic pattern which indicates to test the related GUI component (i.e., Search text field) to assist in clarifying the target of the tests (based on the associated event in the data model). With this listing of tests from the generator, the test selection was then tested by supplying an EDG based on the JMeter GUI rip which specifically included the context of the Search window (which is reachable in JMeter through the `Search` menu and then selecting the `Search` menu-item), as illustrated in Figure 4.6.

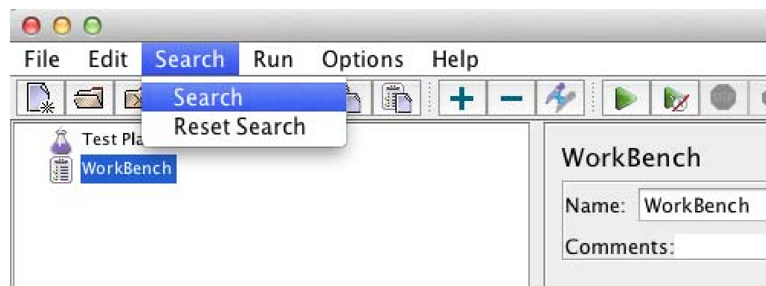

Figure 4.6: JMeter Search menu

The `Search` window, illustrated in Figure 4.7, contains several GUI components which have events tied to them: text box, checkboxes, and buttons. The events which were discovered in this window are all accessible from the `Search` menu-

item selection. One test will be, indicating that the `Search` menu-item has changed, and ensure tests are selected which exist in this window. When performing this test, the test selection process selected all the tests associated to the events associated which were hosted in the `Search` window. In addition, it discovered other tests since this is a menu-item that has larger visibility to have other events invoked downstream that were available in the appellation toolbar. When narrowing the tests down further, and instructing the `Case sensitive` checkbox event was changed, the test selection process correctly selected all the valid tests associated to the events which were tied to the GUI events available in that window (and adjacent to the `Case sensitive` checkbox event in the graph. In addition, no other test cases were selected for GUI components that were not capable of being invoked outside of this modal window.

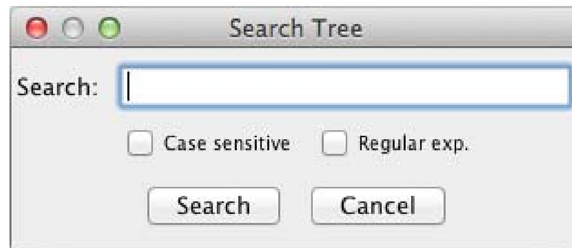

Figure 4.7: JMeter Search window

4.7 Feasibility of the Proposed Solution Implementation

During the implementation of the proposed solution, the area of feasibility was an important area of focus. The goal of this research is to determine an affective solution to a common problem of regression testing for GUI applications. The feasibility perspective of the solution helps gauge the practicality of the solution and how much effort it would require to put into practice. This section will cover some

of the areas of interest which were discovered during the experiments applied to the implementation.

The first area to address will be the GUI event extraction process which was implemented through the GUITAR framework. The GUI ripping process was an effective way of getting an initial measure of the events that exist in an application. As discussed in the earlier section on GUI ripping, one of the key challenges that was encountered was obtaining accurate event information of the application. Due to the natural complexities of GUI applications, it was difficult to get a complete listing of events due to the dynamic nature of events changing state in the application. The state change in the application during runtime would cause altered paths in the ripping process, which may not result in all events getting invoked prior to program termination. By not being capable of invoking all possible events, the event dependence graph would only be partially capable of selecting all possible events affected by a change. This would result in the implementation in not being completely safe on the test case selection process, as it may miss selecting tests without the full context of event relationships for the graph to make its decisions.

Another interesting finding when utilizing the GUITAR framework, was its interpretation of different event types during the ripping process. All events which were identified in the JMeter application were considered as system interaction events. This type of designation in events caused the event dependence graph to be somewhat limited in the possible style of construction for this application. Regardless, the graph traversal to find dependent events still progressed as this type of event was would not hinder in how the `FollowDependentSet` would seek dependent events. In terms of feasibility, it would be desirable to further investigate the functionality of the GUITAR ripping process to determine why this designation of events was occurring. This investigation may lead into further changes to the

ripping functionality to ensure the source of input for the graph construction is as expected. If this investigation is not pursued, the implementation may not be fully complete as the input for event analysis is limited (by being biased to the system interaction event type tied to all the events).

Automated detection of event changes was not included in the implementation used in these experiments. The current implementation would be supplied an event identifier which has changed and it would use this when deciding which test cases to select. It is desired that this would also be an automated process; however, it was discovered through these experiments that detection of an event change is a difficult challenge based on the source of identification of events. When leveraging the GUITAR framework for GUI ripping, other core attributes of the associated GUI component are extracted and are available to make further decisions. These attributes directly relate to the characteristics of the GUI itself; such as, location, size, active or in-focus. Therefore, if the change is applied to one of these attributes, it would be capable of detecting that change by comparing the the ripping results against previous versions of the application. The challenge introduced would be determining the natural key (set of attributes) for identifying the events which would not cause a new identity based on the GUI component's change. The identification that is used currently, is a hash computation based on some of the GUI component's attributes, and therefore if these attributes change, the identity of the event will also change. By having a new identity, you could not simply compare by ID for events, as a new identity would be created each time. This would result in the analysis viewing a modification of an event, as an event which was removed and a new event was added (one identity lost, one identity created).

4.8 Summary

In this chapter, an implementation of the proposed solution was explained and assessed. The targeted application (Apache JMeter) that was under test was described and reasoning for utilizing this program in the experiment was presented. The illustrated implementation was comprised of several key components: a GUI ripping utility (GUITAR framework) for assessing dynamic state of the application, a Java analysis tool was built which consumed the output of the ripping process, a relational data model was constructed for associating test cases to events, and lastly functionality implemented in Java was used to select test cases which were associated to affected events based on the event dependence graph. Tests were applied on the GUI ripping process to evaluate the accuracy of events selected. Challenges and limitations of the ripping process became known and were explained. During the experiments on the implementation, general performance metrics were captured to help illustrate some of the relative time costs in the execution of each phase. One GUI component from Apache JMeter was then focused on, the Search window, to examine what test plans would be filtered when assessing the entire set of test plans associated to the application. Through the test of this example, all the desired tests associated to the GUI events which could be triggered from the Search window were then selected, and no other tests which were undesirable for this change set were selected. Finally, a feasibility section was provided which listed the challenges and risk points when assessing to utilizing this approach on GUI applications. In this section, several areas were discovered for future work to support the area of GUI event analysis.

Chapter 5

Contributions and Areas for Further Research

5.1 Summary

This thesis defines the event control dependence predecessor (ecd-predecessor) and the event control dependence successor (ecd-successor); identifies the event relationship and represents them as an event dependence graph. It also provides an algorithm to construct event dependence graphs for GUI components. Then it expands and applies the regression test selection technique to event based GUI components. At last, it presents a regression test selection algorithm `EventTestSelections` for the GUI component by using the event dependence graph. The algorithm takes a GUI component G, its modified version G', and the test suite T for G, and returns T', a set that contains test cases that are modification traversing from G and G'.

Experiments are applied which provide an implementation of the proposed solution. The implementation utilizes the GUITAR framework to extract event information. A Java application was built which constructs a graph from the GUITAR

event extraction information that is capable of determining dependent events. A data model was designed and utilized which allows for selection of test cases for events which are considered affected from a specified event change. Tests are applied against the Apache JMeter application to prove it is capable of identify relevant tests for a specified changed on a non-trivial application.

5.2 Research Contributions

Throughout this research, several key contributions to the area of regression testing and GUI event analysis were made.

The first major contribution was the presentation of a new model which illustrates GUI event relationships and how those can be considered when assessing which test plans should be selected during regression testing. Currently, there are significant costs which exist with regression testing and this model promotes assessing event relationships to assist in accurately selecting tests plans for a GUI component change. This approach is a paradigm shift from traditional approaches which would assess more of workflows and static relationships of the underlying code, rather than the rich and extensive nature of events within the GUI application. By further examining the event relationships of a GUI application, many new opportunities of assessing relationships of software can be made which can dramatically improve how software is tested and focusing on tests which exercise the affected areas of the application.

With the proposed solution, the experiments of implementing this approach provided rich context to the true challenges of assessing events and the fluid nature of GUI applications. From doing prior research in these areas, it was not apparent how difficult it was to extract and accurately discern event relationships of a GUI appli-

cation until it was attempted in these experiments. Other related research in these areas did not address the challenges of controlling the environment of the GUI application and the importance of naturally determining an identity of an event simply off of its current state. Feasibility notes were provided which further helps provide transparency of true challenges in these areas which have not been expressed in other related areas of GUI event analysis.

Finally, the implementation notes supporting the proposed solution, provides a working solution which can select test cases based on a specified changed event by evaluating all the related events in its model. These contribute to future efforts in the area of assessing what implementations exist to support GUI event analysis and how that data can be further leveraged (in this case relating to test cases). When utilizing the GUITAR framework, documentation exists for select uses and for extending their framework, but it was not clear on the candidate data which it can capture and how the framework needs to be used to accurately capture that information. The experiments provide further transparency in utilizing this tooling and how it was orchestrated to support the goal of the implementation (which may related to other areas of GUI event analysis functionality).

5.3 Future Work

The scope of this work was focused on providing a selective regression test selection technique for modified GUI applications. Several important related topics such as the coverage criteria and test coverage requirements for the modified GUIs are not addressed. When working through the experiments applied in this research, several areas in regards to event analysis are candidates for future work. The following areas are listed below which related to event identification through its lifetime of

modification, automatic event change identification, and improved process of event information extraction through GUI ripping.

5.3.1 Accurately Maintaining an Identity of a Modified Event

During the experiments to support the proposed solution, it was discovered that there is a significant challenge in identifying an event based on its natural identifiable characteristics. This is because those characteristics can be changed with the software, which can ultimately change the identity of that event. This change to the event's identity can cause disruption in the event to test case relationships which have been established. Therefore, it would be desirable to seek out other means of determining the natural key of an event based on its characteristics in the GUI application which can maintain its original identity as changes are applied to those characteristics. This is to strengthen the relationships to existing test cases for an event, since a change to the event's identity would cause it to possibly lose its relational identity to those previous test cases. By maintaining this history of identity, the history of test cases associated to that event can be leveraged for test case selection (rather than a hashing of new event identities on each change which orphans its previously established relationships to existing test cases).

5.3.2 Automated Identification of Changed Events

As mentioned in the feasibility related section in Chapter 4, the implementation did not support the automated means of determining the difference between two versions of a GUI application. This limitation hinges on the future work listed earlier on being capable of accurately relating event changes when that change alters an attribute which alters its natural identity. Meaning, if an event is changed and that

change makes it appear as a new event, the automated process may not be able to determine that the event was modified, rather being an event which has been removed and a new event added. Therefore, it would desirable to expand the means of comparing event dependence graphs so that changes between application versions can be accurately captured through an automated means.

5.3.3 Improving the GUI Ripping Process

Lastly, it would be highly desirable to improve the GUI ripping process to become a more robust process which is capable of exploring additional events in a given application. As mentioned in Chapter 4 when explaining the feasibility of the implementation approach, it was discovered that not all possible events were being discovered due to the complexity of previous event invocations changing the applications state to prevent other event paths from being potentially explored. It would be beneficial to make the GUI ripping process capable of taking snapshots of the flows which it has explored and then replaying subsets of those known workflows again in a fresh application state but take a different path to explore (i.e., selecting the second menu item rather than the first one again at a given state). Currently, the ripping process will just explore all possible events until termination (by only avoiding specific components which have been specified to be ignored). This approach builds a large rich application state as the chain of events which are being invoked in sequence is large, but it doesn't necessarily invoke all the possible events or in the flows which would expose other relationships.

Bibliography

[1] Sergio Antoy and Dick Hamlet. Automatically checking an implementation against its formal specification. *IEEE Trans. Softw. Eng.*, 26(1):55–69, Jan. 2000.

[2] F. Belli, M. Beyazit, and N. G§andler. Event-based GUI testing and reliability assessment techniques – an experimental insight and preliminary results. In *Software Testing, Verification and Validation Workshops (ICSTW), 2011 IEEE Fourth International Conference on*, pages 212 –221, Mar. 2011.

[3] Fevzi Belli, Christof J. Budnik, and Lee White. Event-based modelling, analysis and testing of user interactions: approach and case study: Research articles. *Softw. Test. Verif. Reliab.*, 16(1):3–32, Mar. 2006.

[4] Penelope Brooks and Atif M. Memon. Automated GUI testing guided by usage profiles. In *ASE '07: Proceedings of the 22nd IEEE international conference on Automated software engineering*, Washington, DC, USA, 2007. IEEE Computer Society.

[5] Renee Bryce, Sreedevi Sampath, and Atif M. Memon. Developing a single model and test prioritization strategies for event-driven software. *IEEE Transactions on Software Engineering*, NN(N), 2011.

[6] Robert Cartwright and Mattias Felleisen. The semantics of program dependence. In *Proceedings of the ACM SIGPLAN 1989 Conference on Programming language design and implementation*, PLDI '89, pages 13–27, New York, NY, USA, 1989. ACM.

[7] Bruce A. Cota, Douglas G. Fritz, and Robert G. Sargent. Control flow graphs as a representation language. In *Proceedings of the 26th conference on Winter simulation*, WSC '94, pages 555–559, San Diego, CA, USA, 1994. Society for Computer Simulation International.

[8] Brett Daniel, Qingzhou Luo, Mehdi Mirzaaghaei, Danny Dig, Darko Marinov, and Mauro Pezzè. Automated GUI refactoring and test script repair. In *Proceedings of the First International Workshop on End-to-End Test Script Engineering*, ETSE '11, pages 38–41, New York, NY, USA, 2011. ACM.

[9] Apache Derby. `http://db.apache.org/derby`.

[10] Emine Dumlu, Cemal Yilmaz, Myra B. Cohen, and Adam Porter. Feedback driven adaptive combinatorial testing. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ISSTA '11, pages 243–253, New York, NY, USA, 2011. ACM.

[11] O. El Ariss, Dianxiang Xu, S. Dandey, B. Vender, P. McClean, and B. Slator. A systematic capture and replay strategy for testing complex GUI based java applications. In *Information Technology: New Generations (ITNG), 2010 Seventh International Conference on*, pages 1038 –1043, Apr. 2010.

[12] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9:319–349, 1987.

[13] Todd L. Graves, Mary Jean Harrold, Jung-Min Kim, Adam Porter, and Gregg Rothermel. An empirical study of regression test selection techniques. *ACM Trans. Softw. Eng. Methodol.*, 10(2):184–208, Apr. 2001.

[14] Mark Grechanik, Qing Xie, and Chen Fu. Maintaining and evolving gui-directed test scripts. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 408–418, Washington, DC, USA, 2009. IEEE Computer Society.

[15] GUITAR. `http://sourceforge.net/apps/mediawiki/guitar`.

[16] Mary Jean Harrold, James A. Jones, Tongyu Li, Donglin Liang, Alessandro Orso, Maikel Pennings, Saurabh Sinha, S. Alexander Spoon, and Ashish Gujarathi. Regression test selection for java software. *SIGPLAN Not.*, 36(11):312–326, Oct. 2001.

[17] D. Hoffman, L. Sobotkiewicz, Hong-Yi Wang, P. Strooper, G. Bazdell, and B. Stevens. Test generation with context free grammars and covering arrays. In *Testing: Academic and Industrial Conference - Practice and Research Techniques, 2009. TAIC PART '09.*, pages 43 –47, Sept. 2009.

[18] Apache JMeter. `http://jmeter.apache.org/`.

[19] A. Memon, I. Banerjee, N. Hashmi, and A. Nagarajan. Dart: a framework for regression testing "nightly/daily builds" of GUI applications. In *Software Maintenance, 2003. ICSM 2003. Proceedings. International Conference on*, pages 410 – 419, Sept. 2003.

[20] A.M. Memon, M.E. Pollack, and M.L. Soffa. Hierarchical GUI test case generation using automated planning. *Software Engineering, IEEE Transactions on*, 27(2):144 –155, Feb. 2001.

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

[22] Atif M. Memon. GUI testing: Pitfalls and process. *Computer*, 35(8):87–88, 2002.

[23] Atif M. Memon. Automatically repairing event sequence-based GUI test suites for regression testing. *ACM Trans. Softw. Eng. Methodol.*, 18(2):4:1–4:36, Nov. 2008.

[24] Atif M. Memon, Ishan Banerjee, and Adithya Nagarajan. GUI ripping: Reverse engineering of graphical user interfaces for testing. In *Proceedings of The 10th Working Conference on Reverse Engineering*, Nov. 2003.

[25] Atif M. Memon, Martha E. Pollack, and Mary Lou Soffa. Using a goal-driven approach to generate test cases for GUIs. In *ICSE '99: Proceedings of the 21st international conference on Software engineering*, pages 257–266, Los Alamitos, CA, USA, 1999. IEEE Computer Society Press.

[26] Atif M. Memon, Martha E. Pollack, and Mary Lou Soffa. Automated test oracles for guis. In *Proceedings of the 8th ACM SIGSOFT international symposium on Foundations of software engineering: twenty-first century applications*, SIGSOFT '00/FSE-8, pages 30–39, New York, NY, USA, 2000. ACM.

[27] Atif M. Memon and Mary Lou Soffa. Regression testing of GUIs. In *ESEC/FSE-11: Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 118–127, New York, NY, USA, 2003. ACM Press.

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

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

[30] I. A. Natour. On the control dependence in the program dependence graph. In *Proceedings of the 1988 ACM sixteenth annual conference on Computer science*, CSC '88, pages 510–519, New York, NY, USA, 1988. ACM.

[31] A. Orso, M.J. Harrold, D. Rosenblum, G. Rothermel, M.L. Soffa, and H. Do. Using component metacontent to support the regression testing of component-based

software. In *Software Maintenance, 2001. Proceedings. IEEE International Conference on*, pages 716 –725, 2001.

[32] Thomas Ostrand, Aaron Anodide, Herbert Foster, and Tarak Goradia. A visual test development environment for GUI systems. In *Proceedings of the 1998 ACM SIGSOFT international symposium on Software testing and analysis*, ISSTA '98, pages 82–92, New York, NY, USA, 1998. ACM.

[33] G. Rothermel and M.J. Harrold. Selecting regression tests for object-oriented software. In *Software Maintenance, 1994. Proceedings., International Conference on*, pages 14 –25, Sept. 1994.

[34] G. Rothermel and M.J. Harrold. Analyzing regression test selection techniques. *Software Engineering, IEEE Transactions on*, 22(8):529 –551, Aug. 1996.

[35] G. Rothermel, L. Li, and M. Burnett. Testing strategies for form-based visual programs. In *Proceedings The Eighth International Symposium On Software Reliability Engineering*, pages 96 –107, 1997.

[36] Gregg Rothermel, Sebastian Elbaum, Alexey G. Malishevsky, Praveen Kallakuri, and Xuemei Qiu. On test suite composition and cost-effective regression testing. *ACM Trans. Softw. Eng. Methodol.*, 13(3):277–331, July 2004.

[37] Gregg Rothermel and Mary Jean Harrold. A safe, efficient regression test selection technique. *ACM Trans. Softw. Eng. Methodol.*, 6(2):173–210, Apr. 1997.

[38] Gregg Rothermel, Mary Jean Harrold, and Jeinay Dedhia. Regression test selection for c++ software. *Software Testing, Verification & Reliability*, 10:2000, 1999.

[39] J. Strecker and A.M. Memon. Relationships between test suites, faults, and fault detection in GUI testing. In *Software Testing, Verification, and Validation, 2008 1st International Conference on*, pages 12 –21, Apr. 2008.

[40] Yanhong Sun and Edward L. Jones. Specification-driven automated testing of gui-based java programs. In *Proceedings of the 42nd annual Southeast regional conference*, ACM-SE 42, pages 140–145, New York, NY, USA, 2004. ACM.

[41] L. White and H. Almezen. Generating test cases for GUI responsibilities using complete interaction sequences. In *Software Reliability Engineering, 2000. ISSRE 2000. Proceedings. 11th International Symposium on*, pages 110 –121, 2000.

[42] L.J. White. Regression testing of GUI event interactions. In *Software Maintenance 1996, Proceedings., International Conference on*, pages 350 –358, Nov. 1996.

[43] Q. Xie and A.M. Memon. Rapid “crash testing” for continuously evolving gui-based software applications. In *Software Maintenance, 2005. ICSM'05. Proceedings of the 21st IEEE International Conference on*, pages 473 – 482, Sept. 2005.

[44] Qing Xie and A.M. Memon. Model-based testing of community-driven open-source GUI applications. In *Software Maintenance, 2006. ICSM '06. 22nd IEEE International Conference on*, pages 145 –154, Sept. 2006.

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

[46] Qing Xie and Atif M Memon. Using a pilot study to derive a GUI model for automated testing. *ACM Trans. Softw. Eng. Methodol.*, 18(2):7:1–7:35, Nov. 2008.

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

[48] Xun Yuan and A.M. Memon. Alternating GUI test generation and execution. In *Practice and Research Techniques, 2008. TAIC PART '08. Testing: Academic Industrial Conference*, pages 23 –32, Aug. 2008.

[49] Xun Yuan and A.M. Memon. Generating event sequence-based test cases using gui runtime state feedback. *Software Engineering, IEEE Transactions on*, 36(1):81 –95, Jan.-Feb. 2010.

[50] Xun Yuan and Atif M. Memon. Using GUI run-time state as feedback to generate test cases. In *ICSE '07: Proceedings of the 29th International Conference on Software Engineering*, pages 396–405, Washington, DC, USA, May 2007. IEEE Computer Society.

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