GUI Test Path Coverage and Optimization Using Ant Colony Optimization

MS-SE Thesis
Submitted By:
Mashal Ibrar
291-FBAS/MSSE/F09
Supervisor: Dr. AamerNadeem
Associate Professor DCS, MAJU
Co-Supervisor: M. Imran Saeed

Department of Computer Science
Faculty of Basic and Applied Sciences
International Islamic University Islamabad
http://iiu.edu.pk
In the name of Almighty Allah,

The most Beneficent, the most Merciful
A dissertation submitted to the
Department of Computer Science and Software Engineering

International Islamic University, Islamabad as a partial
fulfillment of the requirements for the award of the MS (SE)
degree
Dedicated To

My Family
DECLARATION

I, Ms. Mashal Ibrar declare that this thesis, neither as a whole nor as a part thereof has been copied out from any source. It is further declared that this dissertation is made entirely on the basis of my personal efforts under the sincere guidance and help of my supervisor. No portion of the work presented in this report has been submitted in support of any application for any other degree or qualification of this or any other university or institute of learning.

Mashal Ibrar
Reg.#291-FBAS/MSSE/F09
ACKNOWLEDGEMENT

In the name of Allah who is the most gracious, merciful and creator of this universe. I am thankful to Him who blessed me with abilities to do this thesis work.

I am extremely thankful to my supervisor, Dr. AamerNadeem, who was the real source of encouragement and motivation during the whole work of thesis. His useful suggestions, advice and ideas to bottleneck problems encountered during this thesis work were just immensurable.
My appreciation’s also goes to M. Imran Saeed for his precious guidance, fruitful discussions and encouragement throughout this thesis work. I am thankful for his kindness, patience and feedbacks. His expertise, devotion and constant encouragement was very helpful and made this effort an enjoyable one.

At last, I am thankful to my family especially my Mother and Sister for her kind support and encouragement. Indeed, without their prayers, emotional and financial support it would not be possible for me to accomplish this work.
ABSTRACT

Graphical user interface (GUI) has become imperative and commonly used to interact with software system. GUI is a type of user interface that allows users to interact with system. GUI contains approx. 50-60% (Memon, 2001) (Khamis et al. 2012) of the total software code. However GUI make ease to the user to interact with the software but the development process is becoming complex day by day due to large number of GUI interactions. A software organization aims to produce a high quality software product which can be accomplished through testing from various perspectives.

GUI testing is a critical activity that is designed to find defects in the GUI of overall application, and aims to produce reliable, accurate and cost effective system. GUI testing is a system level testing in which event sequences are tested to validate that the desired functionality is full filled or not. Sequences of events are tested against the functionality. Model based software testing (MBST) are used to generate test paths automatically by traversing the model. However test paths generated from these models may be feasible or infeasible.

In MBST, Event Flow Graph (EFG) is used to generate test paths efficiently. In EFG model GUI objects are denoted by events and the edges between events shows the dependencies between events. The testing needs to be done in such a way that it provides effectiveness, efficiency, improved fault detection rate and maximum coverage. To cover all test paths and events of EFG, a technique is proposed that is used to test and generate all paths of event flow graph using ant colony optimization optimally. Another challenging question in the software testing is that how much testing is enough to achieve maximum quality ware software product. A better coverage criterion is use to answer that how much testing is required. Simple graph traversal algorithm generates both feasible and infeasible test paths. Infeasible test paths leads to unknown states which cannot execute. Ant Colony Optimization (ACO) algorithm is also used along with the Event Flow Graph to generate feasible test paths optimally. Due to large and complex nature of software system, testing is done in minimum time to achieve full coverage through event flow graph using ant colony optimization algorithm. The coverage criterion is used to measure software quality by testing the whole system. This ensures that maximum coverage is achieved from event-interaction coverage criterion as compared to simple event coverage criterion. The proposed approach generate feasible test paths of all events and all event-pair interaction.
# Table of Contents

1. **Introduction** .................................................................................................................. 9
   1.1 Software Testing: ........................................................................................................... 9
       1.1.1 Test Paths ............................................................................................................. 10
       1.1.2 Test Data Generation ......................................................................................... 10
   1.2 Graphical User Interface Testing: .................................................................................. 10
   1.3 Event Flow Graph: ........................................................................................................ 11
   1.4 Problem statement ......................................................................................................... 11
   1.5 Research objective ....................................................................................................... 12
   1.6 Organization of Thesis ............................................................................................... 13

2. **Background** .................................................................................................................. 14
   2.1 GUI TESTING: .............................................................................................................. 14
   2.2 Code coverage .............................................................................................................. 15
   2.3 Graphical user interface (gui) coverage ...................................................................... 15
   2.4 Event Flow Graph: ...................................................................................................... 16
   2.5 Types of Gui events .................................................................................................... 17
   2.6 Types of Event Coverage: .......................................................................................... 18
   2.7 Optimized Test data generation: ................................................................................ 20
       2.7.1 ANT COLONY OPTIMIZATION: .................................................................... 21

3. **Literature Survey** ......................................................................................................... 25
   3.1 State-Based Techniques: ............................................................................................ 26
       3.1.1 Belli et al. ............................................................................................................. 26
       3.1.2 Memon and Sofa et al ...................................................................................... 27
       3.1.3 White et al. ....................................................................................................... 27
       3.1.4 Jin and Wang et al. ......................................................................................... 28
   3.2 Event Flow graph ......................................................................................................... 28
       3.2.1 Memon ............................................................................................................... 28
       3.2.2 Lu and Wang et al. .......................................................................................... 28
       3.2.3 Memon and Yuan ......................................................................................... 29
   3.3 Genetic algorithm ....................................................................................................... 29
8.1 Analysis of Experimental Results ................................................................. 71
9 Conclusion and future work ........................................................................... 76
  9.1 Conclusion ................................................................................................. 76
  9.2 Future Work ............................................................................................... 76
10 References ................................................................................................... 77

List of Figures
Fig 2.1 Example of Graphical User Interface (GUI) ........................................ 14
Fig 2.2 An Event Flow Graph (EFG) for a part of MS WORD ......................... 16
Fig 2.3 Example of Restricted Focus Event .................................................... 17
Fig 2.4 Example of Unrestricted Focus Event ................................................ 18
Fig 2.5 EFG before event coverage ................................................................. 19
Fig 2.6 EFG after event coverage ................................................................. 19
Fig 2.7 EFG before event-interaction coverage .............................................. 19
Fig 2.8 EFG after event-interaction coverage ............................................... 19
Fig 2.9 Pheromone trail of real Ants approach .............................................. 21
Fig 2.10 Ant Colony Optimization Algorithm Flowchart .................................. 23
Fig 3.1 Example of simple finite state machine ............................................ 26
Fig 5.1 Abstract level Research Approach ..................................................... 39
Fig 5.2 Diagram of Proposed Approach ....................................................... 44
Fig 6.1 Tool Architecture ............................................................................... 47
Fig 7.1 Event Flow Graph (EFG) Example of Notepad .................................. 56
Fig 7.2 Comparison of event coverage in existing and proposed approach .... 59
Fig 7.3 Comparison of edge coverage in existing
List of Table

Table 3 Comparison of different approaches ..........................................................33
Table 5 ACO Parameter Setting ..............................................................................41
Table 7.1 Comparison of event coverage .................................................................59
Table 7.2 Comparison of Event-interaction Coverage .............................................63
Table 7.3 Event Coverage of Internet Explorer (IE) .................................................66
Table 7.4 Event-interaction Coverage of Internet Explorer (IE) .............................68
Table 8.1 Comparison of Experimental Results .....................................................71
Table 8.2 Coverage criterion for internet explorer ..................................................72
1 INTRODUCTION

Software Testing is the major and important part of software development. Testing takes almost 50-60% (Memon, 2001) (Khamis et al., 2012) of effort and cost of the software development. A test case normally encompasses of an input, output, anticipated result and the actual result. A collection of test cases are called test suite. A test suite contains goals and objectives of each test case. More than one test case is required to test the whole functionality of the GUI application. Due to the importance of the testing phase in a software developmental lifecycle, testing has been divided into graphical user interface (GUI) based testing, logical testing, integration testing, unit testing etc. GUI Testing has become very important as it provides ease of use to user to interact with the software. As the time passes the complexity of GUI testing is increased. The basic aim for the software testing is to provide effectiveness, correctness, better fault detection rate and maximized coverage. Simple testing techniques are used to generate test cases but the generated test cases are might be feasible and infeasible. Due to complex and real nature of software, numbers of generated test cases are infinite. Ant colony optimization is used to generate the optimized and feasible test cases and for finding shortest path which has been purposed to overcome the limitations of finite test cases.

1.1 SOFTWARE TESTING:

Software testing is done through various methods like manual testing and automated testing techniques. In Manual testing, tools such as capture and replay, scripts-driven, and data-driven approach test cases are generated but there are numerous defects exists due to difficulty of massive manual work, low adaptability to software variation, and lack of management for test cases and their coverage. Manual GUI testing is done by tester manually and having a more chance of error in it. Manual testing is tremendously slow, time consuming and expensive. After that automated technique was introduced that enables the process of generating test cases automatically. Automated GUI Testing consist of automated testing tasks that have been done
manually before, using automated techniques and tools. It is more effective, reliable, 
accurate and cost effective (Myers, 2011)(Memon 2001) (Whittaker and James, 2000)

1.1.1 Test Paths

Test path is the execution of sequence of statements from start point to exit or end 
point. Program can have a large or even infinite number of paths and there is a pair of 
input, its expected outcome and generated outcome for each path in a program. 
Feasible and infeasible are two types of Generated Paths (Naik and Tripathy, 2008)

- **Feasible Paths:** A path for which a set of input values and preconditions exists 
  which causes it to be executed
- **Infeasible Paths:** A path that cannot be exercised by any set of possible input 
  values

1.1.2 Test Data Generation

In automated software testing sufficient test data generation is the process of identifying 
a set of program input data, which satisfies a given testing criterion (Tracey et al, 
1998). Automated testing is used to generate quality wise product in a low cost. For 
achieving this functionality test data generation techniques such as random, symbolic, 
and dynamic test data generation techniques exist. Swarm optimization techniques 
have been applied recently to generate test data successfully.

1.2 **Graphical User Interface Testing:**

Graphical user interface testing is the process of testing software's graphical user 
interface to preserve and achieve fault free software. In GUI testing, some tasks are 
performed and then the actual result is compared with the predictable output. GUI 
testing is a critical activity that is designed to find defects in the GUI or in the overall 
application, and increasing the self-assurance in its correctness which gives reliable, 
accurate and cost effective system. GUI testing is a system level testing in which event 
sequences are tested to validate that the desired functionality is full filled or not. 
Sequences of events are tested against the functionality. Model based GUI testing
(MBST) are used to generate test paths automatically by traversing the model. However test paths generated from these models may be feasible or infeasible.

Li et al. discussed in their research work that there are some important characteristics of GUI systems and their limitation are described: (Li and pinget al, 2007)

- In GUI there are extremely large number of input and events which leads to large number of states to be tested. Large number of states leads testing complex and challenging.
- The synchronizations and dependencies between objects cannot be controlled in the same window (e.g. object in one window may be linked to an object in another window).
- Object oriented software programming
- In GUI applications, the user may use a keyboard shortcuts, a button click, a menu option, a click on another window etc. How many of these should be tested.
- If the windows are closed before completing a transaction may leave the application or the database in an inconsistent state.

1.3 **Event Flow Graph:**
To model GUI component and objects, Event-Flow Graph (EFG) is commonly used. Event-Flow Graph represents all possible interactions amongst the events in a GUI component. In Event-Flow Graph, events are represented as vertices (nodes) and the relationships or interaction between events are represented as edges (arrows) connecting pairs of event vertices. The Event Flow Graph used for Automated GUI testing.

1.4 **Problem Statement**
As discussed earlier that traditional software testing requires much time, cost and effort in manual testing. To overcome this problem Automated software testing techniques are used to generate test data automatically. The generated test suite doesn’t guarantee the fault free software product. Many errors are encounter from user perspectives after
deployment.
Automated test data generation for single path coverage has disadvantage of inefficient and time acquiring in condition of large and complex software systems. But multiple paths test data generation is more efficient due to multiple paths coverage from many perspectives. The generated test cases might be feasible and unfeasible. The infeasible path problems are caused by the execution order among GUI events or states. Huang at el. solves the problem of infeasible path generation by using combinatorial interaction testing (CIT) that is used to explore GUI event sequences. Due to large and complex nature of software system there might be infinite number of test data generation. To deal with the large event statespace problem ant colony optimization is used. Ant colony algorithm (ACO) is applied to dynamically generate feasible test cases, which are capable of finding faults. The ant colony algorithm is an algorithm for finding optimal paths. Huang et al. proved that test cases generated by ACO are better than the ones generated by simple graph traversing algorithm. Huang et al. present ACO algorithm to generate test cases that are more fault-prone without infeasible problems but problem is that they didn’t cover all the events from different perspectives. Coverage of all events needs improvements because it will affect the accuracy and fault-free software.

1.5 Research objective
Our main objective is to generate test path which cover all the events and all event-interaction coverage in event flow graph. We will improve efficiency of test data generation. We will generate the test paths in less number of test execution and less time by calculating the probability. Existing problem is that the generated test paths didn’t cover all events and their child relations (follow-relation). So, we enhance the coverage of all events. In this research work a fully automated test case generation framework for GUI testing is proposed. The main objective is to generate all possible events and event-interaction relationship dynamically by using Ant colony algorithm. Also we eliminated the infeasible test path generation whereas the test path generation is automatic. After the experimental results it has been proved that proposed solution...
covers all events and all event-interaction (follow-relation events). The generated test path from the proposed solution also remove the infeasible test path.

### 1.6 Organization of Thesis

This thesis is organized as follows. Chapter 2 gives the detail background knowledge of automated software testing and Ant Colony Optimization algorithm. Chapter 3 includes the related works of automated test data generation techniques and test data generation from event flow graph using ACO. This chapter also includes the comparison of existing approaches. Chapter 4 include the problem statement which was deduced from literature survey. Chapter 5 provides the proposed solution of the problem which is discussed in literature survey. Chapter 6 include the Implementation details and tool architecture of the proposed solution. Chapter 7 contain the case studies of different GUI examples on which proposed approach is applied. Chapter 8is comprised of result and discussion which explains and evaluates the experimental results of the proposed approach. Chapter 9 concludes the thesis work and gives the future work.
2 BACKGROUND

This chapter includes background of automated GUI testing techniques in detail, coverage criteria and details of Ant colony optimization that are used to achieve accurate and optimize test data generation.

2.1 GUI TESTING:

GUI testing is vital to make the entire system safer and more reliable. GUI testing is a critical activity aimed at finding defects in the GUI application to increase the confidence on its correctness and accuracy. Software testing techniques can be categorized into two main types i.e., static testing and dynamic testing. In Static testing, program is not executed which is used to find errors by reading the code and examination the design as non-execution or verification technique performed by tester or automated tool. This technique cannot perform detailed testing. Some static analysis methods are code inspections, code walkthroughs, and code reviews. Static testing is more error prone, extremely slow and unacceptably expensive(Whittaker et al, 2000).

Fig 2.1 Example of Graphical User Interface (GUI)
Dynamic testing is known as validation technique. This technique must execute the code and verify the output with expected outcome. Dynamic testing has further classified into two parts, i.e., black box testing and white box testing.

2.2 Code coverage:

Code coverage is a technique to ensure that code must be tested through test cases. For accurate software, coverage criteria are used to achieve fault-free software. Coverage criteria are sets of rules used to define whether a test suite has adequately tested program. The most renowned code coverage criteria are statement coverage, branch coverage, decision coverage, decision-condition coverage and path coverage.

In Statement Coverage criterion every statement in the program must be executed at least once. Statement coverage is considered as a weaker criterion than others because it is not sure that it executes the same statement in different sequence (Yuan and Memon, 2011).

2.3 Graphical user interface (GUI) coverage

For GUI testing various models are used to represent the GUI components as events or nodes and relation between them is shown in dependency. For GUI testing some models are used for testing know as Model based testing (MBST). In MBST models are traversed to generate test paths. For Model Based GUI testing different coverage criterion are proposed. One of them is path coverage criterion. Path Coverage Criterion must execute all the paths from the begin node to the end node in the flow graph.

Due to complex nature of software there are different paths in a program having loops resulting infinite test paths which didn’t guarantee the accuracy of a tested program through path coverage. This means that the testing cannot finish in a finite period of
time. But due to time pressure and scarcity of other resources, software testing must be fulfilled within a limited fixed period of time (Hamlet, 1994).

A path coverage criterion is stronger than statement and branch coverage because it considers all possible sub sequences of a program.

2.4 EVENT FLOW GRAPH:

To model GUI component, Event-Flow Graph (EFG) is commonly used. Event-Flow Graph represents all possible interactions amongst the events in a GUI component. Memon et al. worked a lot on the GUI testing along with event flow graph. In Event-Flow Graph, events are represented as vertices (nodes) and the relationships between events are embodied as edges (arrows) connecting pairs of event vertices. The Event Flow Graph used for Automated GUI testing.

An Event-Flow Graph contains all events that may be executed at a given time. When GUI components are dynamic, it may be accessed at the same time when parent of GUI component are accessed. Event-Flow Graph always contains events for a GUI component along with all of its child (adjacent) GUI components. A link from one event to another means that the second event can be executed after the first event. In a typical GUI component, there is a high connectivity between GUI events.

![Event Flow Graph](image)

**Fig. 2.2 An Event Flow Graph (EFG) for a part of MS WORD (Memon and sofa, 2001)**
However above coverage criteria do not ensure the adequacy of GUI test cases for following reasons. First, the source code of pre-complied elements may not always be available to be used for coverage evaluation. Secondly, GUI input consists of sequence of events and the number of possible variations of the events may lead to a large number of GUI states. For adequate testing, a GUI event may need to be tested in a large number of these states. Due to high level of abstraction GUI event sequence can’t be obtained from code. Similarly the code can’t be used to guarantee the adequacy of the sequence of events that have been tested.

(Memon, 2001) presented some contribution related to the characteristics of the coverage criteria to overcome the above challenges of the coverage criteria based on events in GUI. GUI events are divided in further sub groups.

2.5 Types of GUI Events

- **Restricted-focus events** open modal windows, the windows that once invoked, they monopolize the GUI interaction, restricting the focus of the user to a specific range of events within the window until the window is explicitly terminated by a termination event. Open menu of file menu in MS Word is an example of restricted-focus events in many GUI systems; the user clicks on open, a window opens and the user spends time in selecting and customizing the options, and finally explicitly terminates the interaction by either clicking Open or Cancel.
Fig 2.3 Example of Restricted Focus Event

- **Unrestricted-focus events** open modeless windows that do not restrict the user's focus; they simply expand the set of GUI events available to the user. Note that the only difference between menu-open events and unrestricted-focus events is that the later open windows that have to be explicitly ended. For example, in the MS Word, open Edit menu then select Find are displayed in an unrestricted-focus window.

![Unrestricted-focus event example](image1)

Fig 2.4 Example of Unrestricted Focus Event

- **System-interaction events** interact with the underlying software to perform some action. Examples are cutting and pasting text, and opening object windows.
- **Termination events** close modal windows. Examples include Ok and Cancel.

### 2.6 Types of Event Coverage:

1. **Intra-component Coverage:** Intra coverage criteria contain Event Coverage, Event-interaction Coverage (event pair coverage) and Length-n Event-sequence Coverage. Detail of these coverage criteria is described below.
2. **Inter-component Criteria**: Inter coverage criteria divides into following category: Invocation Coverage, Invocation-termination Coverage and Inter-component Length-n Event-sequence Coverage. In this paper we only consider Intra-component coverage criteria.

- **Event Coverage**

In Event coverage, each event in the component must be performed at least once. It is necessary to check whether each event executes as expected. When all the events are executed at least once event coverage criteria is achieved.

For example In Fig 2.5 Empty circle shows events sequences which is not executed and filled circle shows that events are executed or traversed at least once represented in Fig 2.6

![Fig 2.5 EFG before event coverage](image1)
![Fig 2.6 EFG after event coverage](image2)

- **Event-interaction Coverage**

Event-interaction criteria is also known as event-pair coverage criteria. In GUI testing it is important to check the interactions among all possible pairs of events in the component. For achieving event-interaction coverage there is constraint to checks the pairs of events that may be performed in a sequence.
In this criterion all event-interaction of event A should be executed at least once after an event A has been performed, like F are executed at least once than event-interaction criteria for event A is fulfilled. Event is set as completely explored when all its incident events are executed at least once. Lines in events shows that the events are executed at least once. When all its incident node are executed its color changed into solid blue like in event A.

- **Length-n Event-sequence Coverage**

Sometime the behavior of events may change when performed in different contexts. In that situation event coverage and event-pair coverage criteria is not appropriate for sufficient testing. For this purpose a criterion is defined that captures the contextual impact formally. As the length of the event-sequence increases, the number of possible contexts also increases.

### 2.7 Optimized Test Data Generation:

State-based testing is frequently used in automated GUI testing. Test data generation is very crucial in software testing. Accurate generated test suite is not only to detect the errors in software but also ensure cost reduction associated with software testing. State-based testing is a frequently used approach in software testing. There are two main problems related to state-based software testing:

1. Some infeasible test case are generated.
(2) Some redundant test data are generated to achieve the necessary testing coverage.

2.7.1 ANT COLONY OPTIMIZATION:

Software testing is one of the major parts in the software developmental life cycle. Due to cost, time and other environment, exhaustive testing is not feasible and selecting the right test path is one of the problems in software testing. To overcome these problems we need to automate testing process and generation of effective test paths can decrease the overall cost of testing as well as chance of finding the defects in the software system. For this situation we need to apply ant colony in our real software system to generate feasible and optimize test paths in less time.

Ant Colony Optimization (ACO) is a meta-heuristic approach inspired from the behavior of real ant colonies. The approach seeks ants to find the shortest path from their nest to the food source with the help of a chemical substance called pheromone. Due to time and other resource scarcity it is the need of the software to be adequately tested in an optimized way. For this purpose ACO technique is used to obtain optimized test data generation which covers all events and event-pair interaction.

**Working of Ant Colony:**

According to (Suriet al, 2012) the working of ACO is as follows:

- The ants walk from the nest to the food source while leaving a substance called “pheromone” on their path.
- Other ants are able to smell this pheromone which serve as a guidance to choose their paths depending upon the stronger pheromone concentrations.
- The pheromone deposited on the ground makes the pheromone trail. This trail allows other ants to find the sources of food that have previously identified by their colony ants showing in Figure 2.9.
Fig 2.9 Pheromone trail of real Ants approach

- With the passage of time pheromone keeps on evaporating at some precise rate of evaporation. The ant which cover shorter path would be the first one to return to the nest. This is due to high probability of the ant choosing the same shorter path on its return journey. At the end all the colony ants converge to follow the shortest path after some time which is shown in figure 2.9 last path where all the ants are moving through the shortest path.

To construct probabilistic solutions, the pheromone trails reflects developed search experience of ants and heuristic information related to the problem.
In Ant colony optimization, Local search algorithms is means to find the best and optimal solution and find it till not found. It start from a complete initial solution and try to find a better solution in suitably defined neighborhood of the current solution. For improving solution the algorithm searches the neighborhood. If improved and optimal solution is found, it replaces the current solution and the local search continues until no
improving neighbor solution is left behind in the neighborhood of the current solution and the algorithm ends in a local optimum.
3 Literature Survey

This chapter includes the literature survey of automated test data generation using finite state machine (FSM) and event flow graph (EFG). As mentioned the research problem in above chapter 1, the generated paths are grouped into feasible and infeasible test paths. Some of automated test data generation techniques produces infeasible data.

In the early age of test data generation author used manual testing techniques and strategies. These techniques requires more time and effort and the produced system are more error-prone. To overcome the limitation of manual testing some authors used the specified models to represent the GUI interface. Then test data is generated from these models. This acquire less effort and cost related to testing.

Many researchers have been worked on automated test data generation as in (Yuan and Xuebing, 2010) proposed the approach of test data generation based on finite state machine. Major research have done on State diagram for automated test data generation. Several types of state machine models have been used for software testing, such as Finite State Machine Models (FSM), UML Diagram-based Models and Markov Chains. All of them use a model of the software, algorithms generate test cases from the model.

Due to large and complex software system problem of large input-space have faced. To resolve this situation Event flow graph (EFG) are used. GUI software have different interaction with user and other events of GUI. To represent these interactions between events, EFG are used for GUI testing.

Many researcher have worked on the test data generation optimally. To gain this functionality many swarm optimization techniques are used. Genetic algorithm (GA) and Ant colony optimization (ACO) are related to swarm optimization to test software system optimally.

Here are some techniques that is used to generate test data automatically proposed by many researchers.
3.1 State-Based Techniques:
Many researchers have been worked on automated test data generation. The most commonly proposed approach is test case generation through finite state machine and its variation. To construct a state machine model, assume GUI behavior as a state machine. State transition in the state machine shows the input events.

3.1.1 Belli et al.

Finite state machines have been used to model GUIs. Inside GUI there are different user interactions and interactions between states. GUI states are represented as windows and interaction is consider as a transition in the FSM. Test case or test path in GUI testing corresponds to the sequence of user events. In this paper the author converted FSM into simple formal expressions. The formal expression were used to generate event sequences.

Fig 3.1 Example of simple finite state machine

Limitation:
Due to the large number of possible states as shown in figure and Complex GUI events, FSM have faced scaling problems.

3.1.2 Memon and Sofaet al.

Memon et al. used Artificial intelligence (AI) to accomplish the state-space problem by eliminating unambiguous states. Software tester constructs description of GUI states manually. This description is in the form of planning operators, which defines the preconditions and post-conditions of each GUI event. Test cases are generated automatically by invoking a planner which traverse the path from initial to the target state. This approach works well for systems with a small command language.

Limitation:

Complex GUI have a large number of operations such as menus, buttons, and windows, so this approach needs to be extended to manage large number of operators.

3.1.3 White et al.

White et al. presented state-machine model for GUI test case generation that divides the state space problem into different complete interaction sequences (CIS). The test designer categorizes each user task into machine model called CIS. The CIS is used to generate relative test cases. This approach is effective for partitioning the GUI into manageable functional units and unit testing is done efficiently.

Limitation:

Large manual effort is required in designing the finite-state-machine-model for testing, especially when the code is not available. The creation of state model require executive resources. The state machine models depend upon the understanding and realizing the models according to test designer. When GUI applications are larger and complex it is difficult to manage and analyze. Integration testing is not done through this methodology.
3.1.4 Jin and Wang et al.

In this paper Finite State Machine works with operational profile (op). Probability of random selection of input creates the OP. Then generate the test paths on the basis of these operational profiles. At last the process of validating the effectiveness of this method is measured through design experiment.

Limitation:

The combinations of GUI objects state spaces are enormous, and it is impossible to test all of them. If the object is inmediate complicated, it is almost impossible to take the advantage of finite state model. There might be chance of choosing incorrect operational profile. Only unit testing is done in this case.

3.2 Event Flow Graph

To overcome the state-space problem in finite state machine and generation of large number of states in large and complex software system, many authors generated the test data using Event-Flow graph.

3.2.1 Memon

Many authors used different types of graph models (e.g. event-flow graphs) to generate specific test cases. Memon combine all of the models into one scalable event-flow model and outlines algorithms to semi-automatically reverse-engineer the model from an implementation (Memon, 2007). GUI ripper is used to automatically generate EFG that represents all possible event sequences of GUI. The model defines event-space exploration strategies (ESESs) and creates an end-to-end GUI testing process.

Limitation:
Size of the space of all possible event interactions grows exponentially with length.

3.2.2 Lu and Wang et al.

Y. Lu et al. presented a new GUI automation test model based on the event-flow graph modeling. In this model, authors have presented a technique to generate test cases in the daily smoke test based on an improved ACO.
On the other hand, spanning tree obtained by deep breath-first search (BFS) approach is used to generate test cases from initial point to target point.

**Limitation:**

GUI automation test software needs to verify the validation of the model automatically. Lack of adaptability of the various GUI Operating system. Event-flow graph needs improvement to explain the complex logic problem and reduce the involvement of manual verification.

### 3.2.3 Memon and Yuan

This paper presents a new automated model-driven technique to generate test cases by using feedback from the execution of a seed test suite on an application under test. The test cases in the seed suite are aimed to generate automatically and efficiently. Feedback is achieved from GUI runtime information to generate test case batches iteratively. This new technique can repair infeasible test cases based on the feedback information.

**Limitation:**

The restrictions of this approach is that the methods are too complicated and expensive to generate models from EFG to EIG, from EIG to event semantic interaction graph.

### 3.3 GENETIC ALGORITHM

Genetic algorithm are also used for GUI test data generation. Some of the contribution of the researcher are following.

#### 3.3.1 Khamis et al.

This paper presents a new general technique for the automatic test data generation for spanning sets coverage. The proposed methodology automatically generate the spanning sets of program units that satisfy test coverage criteria. Genetic algorithm is used to automatically generate test data to cover these spanning sets. The GA starts by creating an initial population of individuals randomly. Crossover and mutation is done on
these population set to obtain the required test data. The fitness function is calculated in some problem-dependent way.

**Limitation:**

The problems of infeasible paths identification and train the system to avoid the infeasible paths during finding the optimal solution.

### 3.3.2 Rauf et al.

Genetic algorithm searches for the best possible combinations that are according to some test criterion. This criterion measure how much coverage is achieved by coverage function. This methodology include following steps.

Start: It generate a random population of n chromosomes.

Fitness: Evaluate the fitness of each chromosome x in the population.

New population: Create a new population by selection, crossover and mutation then acceptance of the new populated off spring.

Fitness Function: In fitness function input is given then the fitness function returns a result signifying the acceptability of the program.

Rauf et al. used Genetic Algorithm to find out optimized test data for GUI testing. Genetic Algorithm has been used for the optimization of test coverage. A genetic algorithm is especially appropriate to the solution of indefinite problems or nonlinear complex problems

**Limitation:**

Manual test data generation through clicking on various GUI elements.

Increase in the number of generations for improving coverage which is time consuming and slow.
3.3.3 Preeti et al.

UML state chart diagram using Genetic algorithm are used to generate optimize test cases. To generate new test sequence, crossover has been applied and productivity of the test sequences is calculated by Mutation Analysis. Generate the State flow diagram and collect all the possible paths between the starting to the ending State. Select two possible paths and then perform crossover on these selected paths. It will generate a new path after that mutation is applied on this new path to eliminate the dead paths or infeasible paths.

**Limitation:**

If the number of test sequences are less the result should not be ideal. Only suitable for complex and real time application

Expert’s user’s assumption take longer paths through the input event interaction space when performing activities as compared to the novice user

3.4 **ANT COLONY OPTIMIZATION**

3.4.1 Li and P. Lam

This paper proposes an Ant Colony Optimization approach to test data generation for the state-based software testing. The advantages of the proposed approach are feasible, non-redundant data generation. ACO is a probabilistic technique that can be useful to generate solutions for combinatorial optimizations problems. To represent the State chart model a directed graph is created. Using ACO algorithm, a group of ants can effectively discover the graph and generate optimal test data to achieve test coverage condition.

**Limitation:**

Don’t cope up the redundant states of the state chart model.

Number of states grows exponentially with the size of states in the state chart.
3.4.2 Li and Zhang et al.
This paper presents a model of generating test data based on an improved ant colony optimization and path coverage criteria. In this paper, an approach combining the ant colony algorithm with the branch function technique to automatically generate test data based on path coverage criteria is proposed.

Limitation:
They didn’t focus on the best proportion strategy to choose among poorest.
Character type problem are not handled in this techniques.

3.4.3 Huang et al.
The purposed approach in this paper automatically generate GUI test cases. In ACO the generated test cases are feasible and optimized.

The framework comprises reverse engineering of GUI to create GUI structure and event flow graph (EFG). Ant colony algorithm is applied to these information and model to generate testcases. The benefits of GUI test case generation using the event-flow graph are automatic modelling of GUI objects to reduce expenditure of complex modelling, test cases are executable. For gaining all the above goals automated test case generation through ACO is required.

Author used an approach which uses event-flow-based model to achieving GUI element information through a new accessible technique and implementing a new search algorithm (ACO) to find a feasible event sequence.

In this Paper an Ant Colony Algorithm is used for finding the all optimal path in CFG of Software under test. This Algorithm is helpful for finding all Paths in between the nodes. Selection of path is depends on probability. The higher the probability means higher chance of choosing that path. The probability value of path depends on Pheromone value and heuristic information of path. This is an effective approach which can easily generate optimal paths.

Limitation:
It only provide event coverage and don’t provide full coverage of GUI like follow relation events (event interaction coverage).

A single test path cannot be used to detect all the possible defects in the software.

### 3.5 Coverage Criteria

There is a close relationship between test-case generation techniques and the underlying coverage criteria used. Much of the literature on GUI test case generation focuses on describing the coverage criterion to achieve fault-free software.

#### 3.5.1 Memon et al.

Memon et al. explains various Coverage Criteria using event sequences to identify the adequacy for GUI software. Along with the adequacy of the software system the event sequences can be inaccurate due to the large amount of infeasible event. Memon’s model in this paper also suffers inaccuracy. For example the three main events in this GUI component are YES, NO and CANCEL. Events NO and CANCEL are termination events because they terminate the modal window. However, the event YES can be a restricted-focus event or a termination event based on preconditions before the invocation of this modal dialog.

**Limitation:**

We cannot create an event flow graph for this component using Memon’s definition because we cannot classify the YES event.

### 3.6 Comparison of existing techniques

<table>
<thead>
<tr>
<th>Author Name</th>
<th>Technique Used</th>
<th>Coverage Criteria</th>
<th>Single/Multiple path generation</th>
<th>Infeasible path possibility</th>
<th>Automated</th>
</tr>
</thead>
<tbody>
<tr>
<td>Belli et al.</td>
<td>Finite state machines</td>
<td>State Interaction Sequences</td>
<td>Multiple</td>
<td>Yes</td>
<td>Partial</td>
</tr>
<tr>
<td>Memon</td>
<td>event-flow graph, Artificial intelligence</td>
<td>All event sequences</td>
<td>Not defined</td>
<td>Yes</td>
<td>Partial</td>
</tr>
<tr>
<td>--------------</td>
<td>------------------------------------------</td>
<td>---------------------</td>
<td>-------------</td>
<td>-----</td>
<td>---------</td>
</tr>
<tr>
<td>White et al.</td>
<td>State machine, Complete interaction sequences</td>
<td>All paths</td>
<td>Multiple</td>
<td>Yes</td>
<td>Partial</td>
</tr>
<tr>
<td>Jin et al.</td>
<td>Finite State Machine, OP</td>
<td>State Transition</td>
<td>Not defined</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Memon et al.</td>
<td>Feedback</td>
<td>Event interaction sequences</td>
<td>Multiple</td>
<td>NO</td>
<td>Yes</td>
</tr>
<tr>
<td>Lu et al.</td>
<td>event-flow graph, ACO</td>
<td>All event sequences</td>
<td>Not defined</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Memon et al.</td>
<td>event-interaction graph, Feedback</td>
<td>Event interaction sequences</td>
<td>Multiple</td>
<td>NO</td>
<td>Partial</td>
</tr>
<tr>
<td>Khamis et al.</td>
<td>Genetic algorithm</td>
<td>Data coverage</td>
<td>Multiple</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Rauf et al.</td>
<td>Genetic Algorithm</td>
<td>Event coverage</td>
<td>Multiple</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Preeti et al.</td>
<td>state chart diagram, Genetic algorithm</td>
<td>All states sequence</td>
<td>Multiple</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Li et al. 2007</td>
<td>State chart , Ant Colony Optimization</td>
<td>All state coverage</td>
<td>Multiple</td>
<td>No</td>
<td>Partial</td>
</tr>
<tr>
<td>Li et al. 2009</td>
<td>Ant Colony Optimization</td>
<td>All path coverage</td>
<td>Multiple</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Huang et al.</td>
<td>Event flow graph &amp; Ant Colony Optimization</td>
<td>All event coverage</td>
<td>Single</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Table 3: Comparison of different approaches
4 PROBLEM STATEMENT

Graphical user interfaces (GUI) have been widely used for interacting with software systems. End user interacts with software system through GUI events such as menu selection, pressing buttons and mouse movements etc.

Due to innovation in technology day by day GUI testing is very challenging for real time and safety critical system. GUI testing needs huge improvement to enhance the entire system’s security, safety and reliability. GUI testing can be performed either manually by software tester or automatically by automated methods.

GUI testing involves several tasks like to test all object events, mouse events, menus, fonts, images, content, control lists, etc. GUI testing is performed to check the user interface and test the functionality working properly or not. In GUI testing set of tasks are carried out to test the event sequences against the expected result. If the results differs with each other than it means there must be some faults in the software system.

For Model based GUI testing, GUI objects are modeled as graph for easy understanding and testing in earlier stage when code is not available. Event flow graph (EFG) is created from various ways which are based on requirement, Code and system interface then it is verified through different coverage criterion. To test the adequacy of the software system coverage criterion is used. A coverage criterion is like set of rules that test engineers use to define the nature and extent of essential code to test. In event flow graph (EFG) the events shows the functionality of the system and edges which shows the dependency of these events. There are number of graph traversal algorithm which can generate both feasible and infeasible paths (see Introduction pg. 6). Infeasible test data leads to dead state or state not existing at that time. Infeasible test paths are those paths which can’t execute due to inaccurate sequence of events.

The problem of infeasibility was removed by Event-interactiongraph (EIG) which is refined form of Event flow Graph (EFG) that represents all possible sequences of events that may be executed on the GUI during test execution. The runtime state of
GUI objects are used to automatically identify an Event Semantic Interaction (ESI) relationship between pairs of events. This relationship captures how a GUI event is related to another event. The ESI relationships are used to automatically construct a new model of the GUI, called the Event Semantic Interaction Graph (ESIG) (Yuan and Memon, 2010).

Event semantic interaction graph (ESIG) shows structural and dynamic relationship between pairs of events during execution. Event semantic interaction graph shows all sequence of events defined in the Event flow graph. The problem of infeasible test data generation can be handled with the technique of creating event semantic interaction graph. In this graph only executable event sequences (feasible events) are made.

Another challenging task is optimal test paths generation. To handle this problem ant colony optimization (ACO) algorithm is used. In ACO the generated test paths are optimized because path selection is depending upon the heuristic information. Heuristic value depend upon problem-oriented solution. Huang et al. uses ACO to generate optimized test paths which are not achieved through other simple traversing algorithm.

However the generated test paths only provide event coverage leaving some follow relation (child events) uncovered.

### 4.1 Limitation on existing methodologies:

- Simple graph traversal algorithm can be applied for coverage but it can generate both feasible and infeasible paths. Infeasible paths cannot be implemented by any set of possible input values because the events are disable or execution order among events are not appropriate.
There are various coverage criterion used for adequate test data generation but event coverage criterion is widely used for this purpose. Event coverage criterion only cover events which are not enough for adequate testing.

### 4.2 Proposed solution:

The overall work in this thesis includes the following steps:

- In proposed approach test path generation covers both event coverage and event interaction coverage which is stronger coverage criterion than simple event coverage criterion. Due to proposed approach fault detection capability are maximized because it covers all events and all edges.

- Feasible test path generation by identifying valid interaction sequences.

- Generated test paths include only feasible paths and all events and their follow-relation (child events) along with edges.
Chapter 3 includes the literature survey of automated test data generation from state machine based model and event flow based model. Nevertheless these models have some limitation also, because of complex and real nature of software. From literature we have identified the problems of infeasibility and single generated path which affect the overall performance of software system and sometimes produced flop software product. This chapter includes the proposed approach of event-flow graph (EFG) using Ant Colony Optimization algorithm for optimal test data generation.

Huang et al. focused on the feasible test path generation from event flow graph along with the ant colony optimization algorithm for optimal data generation. In this paper author develop a new user interface accessibility framework (UIA) to model GUI based events on event-flow graph (EFG) automatically, and then ant colony algorithm is applied to generate feasible and optimal test data, which are useful for finding faults.

Ant colony optimization algorithm generates test paths according to the probability of the events. Those event which have higher probability their chances of selection will be higher. The probability depends upon the pheromone value and the heuristic value, heuristic value tells the visibility of the event.

In proposed approach Ant Colony Optimization (ACO) which seeks for optimal test path generation. Optimal in term of efficiency and coverage which provides the coverage of all events and its edges based on probability and generating multiple paths from event-flow graph. The overall work includes the following steps:-

- To generate the test data, an approach is proposed that would satisfy all event coverage and event-pair interaction coverage criteria which provide all edges and all follow relation (child or adjacent) events in event flow graph.
- Construct an optimal solution that contains sequence of events to finish the
traversal of Event-Flow graph in the form of test cases.

- The Proposed approach ensures that each event and its edges in their traversal are executable at that time. This will ensure that only feasible paths are generated.
- Developed the strategy for feasible test data generation and how to cover all follow relationship events to eliminate the maximum faults from all perspectives.
- To improve the test data generation technique the test paths are generated automatically and randomly that increase the efficiency of test data generation.
- Comparing our results with existing approach.
- Focus of the proposed approach is on feasible test paths generation and provide full coverage.

5.1 Research Approach

The proposed research approach contains coverage of Event-Flow graph and optimal test path generation which will explain in detail in this section.

![Fig5.1 Abstract level Research Approach](image)

First of all input is given in the form of Event-flow graph (EFG) which tells events and their dependencies in the form of edges. In the proposed approach a program (ACO) is selected under test that randomly selects the initial event for test data generation. In
ACO, the probability of each event is calculated by using the formula in (Huang et al.) After calculating the probability we have to choose one node having the largest probability among all of them. Another input is given to ant colony optimization (ACO) algorithm is coverage criterion which guarantees that each event and each edges are visited. At the end the generated output is in the form of test path which shows the sequence of events and edges.

5.2 **ANT COLONY OPTIMIZATION TECHNIQUE**

Ant Colony Optimization (ACO) is a Meta heuristic method to solve combinatorial optimization problems by using behavior of real ant colonies. In ACO algorithms numerous generation of artificial ants search for upright solutions. Each ant builds a solution step by step going through a number of probabilistic decisions until a solution is established. Frequently, ants that found a good solution mark their paths by depositing the amount of pheromone on the edges. After that ants of the next generation are attracted by the pheromone which was deposited by previous ants so that they will search improved solutions. Real ants are capable of finding the shortest path from a food source to destination (Li, Zhang et al. 2009)

Probability calculation formula: \[ p_{xy} = \frac{(\tau_{xy}^\alpha)(\eta_{xy}^\beta)}{\Sigma(\tau_{xy}^\alpha)(\eta_{xy}^\beta)} \]

**Description about ACO parameters:**

Pheromone trail are represented by taw (T) showing the pheromone amount from one node to another node and is being continuously updated as the paths are navigated. Heuristic information is represented by eta (\(\eta\)) showing the perceptibility or the attractiveness of the path for an ant and is used to calculate the probability for the ant to choose an accurate path. Probability depends on the feasible set, heuristic and the pheromone level of the corresponding path. Alpha (\(\alpha\)) determines the relative importance of pheromone value and beta (\(\beta\)) defines the visibility of heuristic information.
Evaporation rate ($\rho$) must be greater than 0 and less than 1. These are tuning parameter which are used for selecting the optimal test paths and feasible solution.

### 5.2.1 Parameter setting

Parameter setting for ACO are shown in table below

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>INITIAL PHEROMONE VALUE ($\mathcal{T}$)</td>
<td>0.2</td>
</tr>
<tr>
<td>ALPHA ($\alpha$)</td>
<td>0.2</td>
</tr>
<tr>
<td>BETA ($\beta$)</td>
<td>0.9</td>
</tr>
<tr>
<td>PHEROMONE EVAPORATION RATE ($\rho$)</td>
<td>0.3</td>
</tr>
</tbody>
</table>

*Table 5 ACO Parameter Setting*

Formerly all ants have finished their tour; the amount of pheromone on edges is restructured (by applying the global updating rule).

Pheromone Updating rule: $\mathcal{T}_{xy}(t) \leftarrow (1 - \rho) \cdot \mathcal{T}_{xy}(t) + \rho \cdot \tau_{0}$ (where $\rho$ is the evaporation rate) (Li, Zhang et al. 2009) (Huang et al. 2011)

The pheromone updating rules are designed so that they have a tendency to give more pheromone to edges which should be visited by ants. In general, the higher the two values, the higher the probability of choosing the linked edge. Updating the pheromone trail values in two phases. First, pheromone evaporation is applied to decrease pheromone values. The aim of pheromone evaporation is to avoid an infinite increase of pheromone values and to allow the ant colony to manage poor choices done previously. Pheromone deposit is applied to increase the pheromone values that fit to good solutions the ants have engendered.

### 5.3 Proposed Approach
1. **Build Event Flow Graph (EFG) model**
   Firstly analyze the entire requirement carefully then identify all the events regarding functionality and their dependency with each other. When all events are identified model them in Event Flow Graph (EFG) which represents the events and their relationship with other events.

2. **Ant Colony Optimization Algorithm**
   a) **Put Ant on EFG**
      For GUI events traversing place the ants on the Event Flow Graph. These artificial ants would respond as natural ant practice for food search. Ants seeks for the optimal solution.
   b) **Ant records the number of nodes**
      Input of Event flow graph is given in the form of Xml. Then ant traverses the graph and records the total number of nodes and its adjacent events (child event).
   c) **Traverse the initial event**
      After calculating total number of nodes and its adjacent nodes the ants traverse the initial node randomly.
   d) **Calculate probability**
      Calculate the probability of adjacent events of the current event. Choose the highest probability node among all adjacent node of current event.
   e) **Update local pheromone value**
      After the event of high probability is selected. Update the pheromone value using formula which is given above pp. This is step by step updating of pheromone value.
   f) **Traverse until end node**
      Traverse or visit all events and event-pair interaction (follow-relation events) through calculating the probability until the initial event achieved.
   g) **Record the path**
      Record the path of all node which have highest probability among other child or adjacent nodes. Record each event step by step having greatest probability.
   h) **Print generated path**
      Print the recorded path which shows the all the events and their path towards other events depending upon the probability and covers unvisited events on high priority.
   i) **Update global pheromone**
      When the path has been recorded. Update the global pheromone value using formula. The aim of pheromone evaporation is to avoid an infinite increase of pheromone values and to allow the ant colony to manage poor choices done previously.

3. **Coverage criterion fulfilled**
After all the paths are traversed check whether the event coverage and event interaction coverage criterion satisfied or not. If the paths covers all events and their edges its means that coverage criterion is complete otherwise repeat the procedure of event interaction coverage using ACO rule.

4. **Print all paths having events and event-pair relation**

At the end when coverage criterion is fully satisfied. Print all the paths having events and event interaction nodes. The generated test paths shows the path having sequence of events and edges which shows event-pair relation.
5.4 **Diagram of Proposed Approach**

- Build EFG model
- Put Ant on EFG
- Ant records the number of events
- Traverse the initial event
- Calculate probability
- Update local pheromone value
- Traverse until end node
- Record the path
- Print generated path
- Update global pheromone
- Print all paths having events and event interaction relation

**Fig 5.2: Diagram of Proposed Approach**
5.5 PROPOSED APPROACH ALGORITHM

1. Initialization:
   a. Set initial parameters: variable, events, function, input, output, input trail and output path.
   b. Set initial pheromone trails value, pheromone evaporation rate, alpha, beta and individual pheromone rate.
   c. Start traversing the node from the initial event.

   ♦ While termination conditions not meet do (visit all events)

2. Traverse the graph using TraverseGraph function

   ♦ While termination conditions not meet do

   a. Construct Optimal Solution:

      Each ant constructs a path by continuously calculating the getHighestProbability function which tells the probability of adjacent event depend upon the attractiveness of the move, and the trail level of the move.

      1. Select highest probability event
      2. Apply Local pheromone updating rule
      3. Record the path (trail)

   End While

3. If the path is traversed, update it.

   a) Update Trails

   b) Update global updating rule ( it contains pheromone deposit and pheromone evaporation )
- Evaporate a fixed proportion of the pheromone on each trail (which is less than 1 and greater than 0).

- For each trail apply global pheromone update.

- Reinforce the best tour by depositing the Individual pheromone on the trail.

4. \textbf{\textit{while}} coverage criterion full filled
   
   Print event and edges path

   Else

   Go to step 2(traverse the graph)

End While

End While
6 IMPLEMENTATION

In this chapter we will describe the implementation phases of proposed approach. It also includes the implementation of Ant Colony Optimization for optimized test path generation along with the feasible test paths generation and then presents the results of proposed approach.

6.1 TOOL ARCHITECTURE

System

Graph input in XML format

Ant Colony Optimization

Feasible test paths

Test path generation

Fig 6.1 Tool Architecture

6.2 DESCRIPTION ABOUT TOOL ARCHITECTURE

1. Graph input

Input of a program is event flow graph (EFG). Graph is saved in the XML file and then used by system under test (SUT). In XML file the event is given with their
child nodes for easily access.

2. **Ant Colony Optimization**

Ant colony Optimization algorithm is used for optimal generation of test path. ACO algorithm depend upon the pheromone value and the heuristic value. Heuristic value depend upon the problem-oriented solution. Here problem-oriented solution is to cover all events and their edges between events.

3. **Coverage Criterion**

The main purpose of my research is to answer that how much testing is required for fault free system. For testing we need coverage of full system so that maximum error would be detected. There are many types of coverage criterion. In previous technique event coverage criterion is used that is not enough to produce error free system. For accurate and adequate system we purposed event pair coverage criterion that ensure that each event and each edge between events is covered at least once.

4. **Feasible test paths**

In simple graph traversal algorithm there are chance of feasible and infeasible test paths generation which leads to the state which are not available at that time. This problem lead to number of useless test paths.

5. **Test path generation**

The resulted output is in the form of generated test paths which covers all the events and the edges between events. Also the generated test paths are feasible test paths.

6.3 **Graph representation**

The above graph example is saved in XML format which can be easily accessed by the program. The xml format shows the nodes of EFG and its child or adjacent nodes.

<Graph>
  <Node name="File_Menu_Item">
    <ChildNode name="Open_Menu_Item"/>
  </Node>
</Graph>
<ChildNode name="Edit_Menu_Item"/>
</Node>

<Node name="Open_Menu_Item">
  <ChildNode name="File_Menu_Item"/>
  <ChildNode name="Encode_Item"/>
  <ChildNode name="File_Name_Edit"/>
  <ChildNode name="File_Type_Item"/>
  <ChildNode name="Open_Button"/>
  <ChildNode name="Cancle_Open"/>
</Node>

<Node name="Encode_Item">
  <ChildNode name="File_Name_Edit"/>
  <ChildNode name="File_Type_Item"/>
  <ChildNode name="Open_Button"/>
  <ChildNode name="Cancle_Open"/>
</Node>

<Node name="File_Name_Edit">
  <ChildNode name="File_Type_Item"/>
  <ChildNode name="Open_Button"/>
  <ChildNode name="Encode_Item"/>
</Node>

<Node name="File_Type_Item">
  <ChildNode name="File_Name_Edit"/>
  <ChildNode name="Encode_Item"/>
  <ChildNode name="Open_Button"/>
  <ChildNode name="Cancle_Open"/>
</Node>

<Node name="File_Name_Edit">
  <ChildNode name="File_Type_Item"/>
  <ChildNode name="Open_Button"/>
  <ChildNode name="Encode_Item"/>
</Node>

<Node name="File_Type_Item">
  <ChildNode name="File_Name_Edit"/>
  <ChildNode name="Encode_Item"/>
  <ChildNode name="Open_Button"/>
  <ChildNode name="Cancle_Open"/>
</Node>
6.4 IMPLEMENTATION DETAILS

Step 1: Initialization

<table>
<thead>
<tr>
<th>Double ALPHA, Double BETA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Double PHEROMONE_EVAPORATION_RATE</td>
</tr>
<tr>
<td>Double INITIAL_PHEROMONE_VALUE</td>
</tr>
<tr>
<td>Double INDIVIDUAL_PHEROMONE_VALUE</td>
</tr>
</tbody>
</table>

Step 2: Start traversing the edge from the initial event
Step 3: Construct Optimal Solution:

```java
while (hasUnVisitedEdges()) {
    if (selectedEdgeLabel != null && !selectedEdgeLabel.isEmpty()) {
        edgeIndex.setVisited(true);
        path.append(selectedEdgeLabel);
        nodeLabel = tempEdge.getEndingNode();
    }
}
```
A. Select highest probability edge

```java
for (String nodeEdge : nodeEdgesList) {
    String dependsOn = tempEdge.getDependsOn();
    if (dependsOn != null && !dependsOn.isEmpty()) {
        Edge dependsOnEdge = this.edges.get(this.edgeIndexMap.get(dependsOn));
        if (dependsOnEdge.isVisited()) {
            double occurrence = calculateOccurrenceOfEdge(tempEdge);
            tempEdge.setOccurrence(occurrence);
            totalOccurrences += occurrence;
        } else {
            tempEdge.setOccurrence(0.0);
        }
    } else {
        double occurrence = calculateOccurrenceOfEdge(tempEdge);
        tempEdge.setOccurrence(occurrence);
        totalOccurrences += occurrence;
    }
}
for (String nodeEdge : nodeEdgesList) {
    double probability = tempEdge.getOccurrence() / totalOccurrences;
    probabilities.add(probability);
    probabilityEdgeMap.put(probability, nodeEdge);
}
Collections.sort(probabilities);
Collections.reverse(probabilities);
highestProbabilityEdge = probabilityEdgeMap.get(probabilities.get(0));
return highestProbabilityEdge;
```
for (String childNode : childNodes) {
    int nodeIndex = this.nodeIndexMap.get(childNode);
    Node tempNode = this.nodes.get(nodeIndex);
    double updatedPheromoneValue = ((1 - Constants.PHEROMONE_EVAPORATION_RATE) * (Constants.INDIVIDUAL_PHEROMONE_VALUE));
}

B. Apply Local pheromone updating rule

C. Record the path (trail)

for (String tempEdgeLabel : pathEdges) {
    double evaporatedPheromoneValue = ((1 - Constants.PHEROMONE_EVAPORATION_RATE) * edge.getPheromoneValue()) + (Constants.PHEROMONE_EVAPORATION_RATE * Constants.INITIAL_PHEROMONE_VALUE);
    edge.setPheromoneValue(evaporatedPheromoneValue);
}

Step 4: Update global updating rule (it contains pheromone deposit and pheromone evaporation)

Step 5: End
7 CASE STUDIES

To justify our proposed approach, two case studies are used to perform experiment. One on them is the example of Notepad Graphical User Interface (GUI) which is already used in existing paper (Huang et al.). Example shown below in Fig 7.1 demonstrate the events and edges. Events define the functionality of the system represented in circle and the arrow between them shows the dependency between each other.

Fig 7.1 Event Flow Graph (EFG) Example of Notepad

7.1 DESCRIPTION ABOUT EXAMPLE

We are using the same example of Notepad for proposed approach as used (Huanget al.). This example expresses how the ants dynamically construct the feasible test cases. At first choose the initial event randomly which is not depending on other events (vertex of the EFG). Then, at each step it moves along the follow relationship events (adjacent events) of the EFG. The program chooses an event by calculating the probability of each adjacent event and drop pheromone on that selected adjacent event. The
probabilistic rule is based on pheromone rate and heuristic information, the higher the pheromone and the heuristic value of an event, the higher the probability an ant will choose that particular event.

Probability calculation formula:  
\[ p_{xy} = \frac{(r_{xy})^\alpha (\eta_{xy})^\beta}{\sum (r_{xy})^\alpha (\eta_{xy})^\beta} \]  

\( T \) is the current pheromone rate placed on every node and \( \eta \) is the heuristic information which is calculated by count of events followed by current event(y) +1 / count of event y have been visit +1 as shown in equation 1. Choose the highest probability node after that update pheromone rate on that chosen event.

Local pheromone updating rule:  
\[ T_{xy}(t) \leftarrow (1 - \rho) \cdot T_{xy} \]  

Each of the pheromone values will be decreased by a certain percentage after each step by applying local pheromone updating rule. An ant has created a solution once it has visited all the events and edges of the graph or it cannot move on anymore (no follow events left). \( \rho \in (0, 1] \) is the evaporation rate shown in equation 2.

Take the EFG of notepad as shown in Fig. 7.1 at the beginning, an initial event (e.g. Filemenuitem event) is selected randomly from EFG, filemenuitem is in enabled mode means it is not dependent upon other event. EditMenuitem and openmenuitem are the adjacent event of filemenuitem which shows follow relation. To choose the next event, equation (1) is used to calculate the probability of current event (filemenuitem) follow relations e.g. editmenuitem, openmenuitem. \( T \) (taw) = 0.2, \( \eta \) (eta) of openmenuitem = 6+1 /0+1 =7, \( \eta \) of editmenuitem = 4+1 /0+1 = 5

\[ P_{\text{filemenuitem, openmenuitem}} = \frac{0.2^{-2} \cdot 7^{-0.9}}{0.2^{-2} \cdot 7^{-0.9} + 0.2^{-2} \cdot 5^{-0.9}} = 0.5751 \]

\[ P_{\text{filemenuitem, editmenuitem}} = \frac{0.2^{-2} \cdot 5^{-0.9}}{0.2^{-2} \cdot 5^{-0.9} + 0.2^{-2} \cdot 7^{-0.9}} = 0.4248 \]

The probability of openmenuitem is greater than probability of editmenuitem, so we choose openmenuitem. After that update pheromone value as shown in equation (2).

\[ T_{\text{openmenuitem}} \leftarrow (1 - 0.3) \cdot 0.2 = 0.14 \] so current pheromone value is decreased by 0.14.
Now we check the follow relation of openmenuitem for choosing the next event through the probability calculation formula and local pheromone updating rule is applied on the selected event. Similarly do until all events are traversed.

When a path is traversed deposit some individual pheromone to strengthen the optimized path and pheromone evaporation to remove the infinity of the increase in pheromone value as shown below in equation (3)

Global pheromone updating rule: $T_{(y)}(t) = (1 - \rho) \cdot T_{xy}(t) + \rho \cdot T \cdot 0$  \hspace{1cm} (3)

$\rho$ is the evaporation rate. $T0$ is the initial pheromone value and $T_{xy}$ is the current value of pheromone at time t.

When one path is generated or traversed update the global pheromone value on the whole path. On every iteration the generated test path generates the following sequences. The output of 1st iteration is: file_menu_item -> open_menu_item -> file_type_item -> encode_item -> file_name_edit -> encode_item -> file_type_item -> file_name_edit -> cancle_open

When one path is generated the control goes to the initial node then start traversing the path again similarly in each iteration the program generate the sequence of events in the test paths and edges.

7.2 GENERATED TEST PATHS

All traversed test path events are following which shows that every event is covered or visited at least once:

Path 1: file_menu_item -> open_menu_item -> file_type_item -> encode_item -> file_name_edit -> encode_item -> file_type_item -> file_name_edit -> cancle_open

Path 2: file_menu_item -> edit_menu_item -> find_menu_item -> case_check -> find_edit -> up_item -> down_item -> find_edit -> case_check -> up_item -> down_item -> edit_menu_item -> find_menu_item -> cancle_find -> edit_menu_item -> find_menu_item -> paste_menu_item -> copy_menu_item -> edit_menu_item -> find_menu_item -> case_check -> find_edit -> up_item -> down_item -> find_edit -> case_check -> up_item -> down_item -> find_edit -> case_check -> up_item -> down_item -> find_edit -> case_check -> up_item -> down_item -> cancle_find -> edit_menu_item -> paste_menu_item -> copy_menu_item -> edit_menu_item -> paste_menu_item -> copy_menu_item
Path 3: file_menu_item -> open_menu_item -> open_button

<table>
<thead>
<tr>
<th>No of path generation</th>
<th>Event coverage in Existing approach</th>
<th>Event coverage in Proposed approach</th>
</tr>
</thead>
<tbody>
<tr>
<td>Path 1</td>
<td>17</td>
<td>6</td>
</tr>
<tr>
<td>Path 2</td>
<td>0</td>
<td>10</td>
</tr>
<tr>
<td>Path 3</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Total number of events</td>
<td>17</td>
<td>17</td>
</tr>
</tbody>
</table>

Table 7.1 Comparison of event coverage

![Comparison of event coverage](image)

Fig. 7.2 Comparison of event coverage in existing and proposed approach

All traversed edges of event flow graph which covers all event-pair relation are as follows:

**Edge Path:**

**Edge Path 1:** (file_menu_item-open_menu_item) -> (open_menu_item-file_type_item) -> (file_type_item-file_name_edit) -> (file_name_edit-file_type_item) -> (file_type_item-encode_item) -> (encode_item-file_type_item) -> (file_type_item-file_name_edit) -> (file_name_edit-encode_item) -> (encode_item-file_name_edit) -> (file_name_edit-
Edge Path 11: (file_menu_item-edit_menu_item) -> (edit_menu_item-find_menu_item) -> (find_menu_item-up_item) -> (up_item-down_item) -> (down_item-up_item) -> (up_item-edit_menu_item) -> (edit_menu_item-copy_menu_item) -> (copy_menu_item-edit_menu_item) -> (edit_menu_item-find_menu_item) -> (find_menu_item-down_item) -> (down_item-edit_menu_item) -> (edit_menu_item-paste_menu_item) -> (paste_menu_item-edit_menu_item) -> (edit_menu_item-file_menu_item)

Edge Path 12: (file_menu_item-open_menu_item) -> (open_menu_item-encode_item) -> (encode_item-cancle_open) -> (cancle_open-file_menu_item)

Edge Path 13: (file_menu_item-edit_menu_item) -> (edit_menu_item-copy_menu_item) -> (find_menu_item-edit_menu_item) -> (edit_menu_item-find_menu_item) -> (paste_menu_item-copy_menu_item) -> (copy_menu_item-edit_menu_item) -> (edit_menu_item-file_menu_item)

Edge Path 14: (file_menu_item-open_menu_item) -> (open_menu_item-open_button) -> (open_button-file_menu_item)

Edge Path 15: (file_menu_item-edit_menu_item) -> (edit_menu_item-find_menu_item) -> (find_menu_item-cancel_find) -> (cancel_find-edit_menu_item) -> (edit_menu_item-find_menu_item) -> (find_menu_item-find_button) -> (find_button-edit_menu_item) -> (edit_menu_item-paste_menu_item) -> (paste_menu_item-edit_menu_item) -> (edit_menu_item-file_menu_item)

Edge Path 16: (file_menu_item-open_menu_item) -> (open_menu_item-cancelOpen) -> (cancel_open-file_menu_item)

Edge Path 17: (file_menu_item-edit_menu_item) -> (edit_menu_item-find_menu_item) -> (find_menu_item-find_edit) -> (find_edit-cancel_find) -> (cancel_find-edit_menu_item) -> (edit_menu_item-copy_menu_item) -> (copy_menu_item-edit_menu_item) -> (edit_menu_item-paste_menu_item) -> (paste_menu_item-edit_menu_item) -> (edit_menu_item-file_menu_item)

Edge Path 18: (file_menu_item-open_menu_item) -> (open_menu_item-file_type_item) -> (file_type_item-file_name_edit) -> (file_name_edit-file_type_item) -> (file_type_item-encode_item) -> (encode_item-file_type_item) -> (file_type_item-file_name_edit) -> (file_name_edit-encode_item) -> (encode_item-file_type_item) -> (file_type_item-open_button) -> (open_button-file_menu_item)

Edge Path 19: (file_menu_item-edit_menu_item) -> (edit_menu_item-find_menu_item) -> (find_menu_item-case_check) -> (case_check-cancel_find) -> (cancel_find-edit_menu_item) -> (edit_menu_item-copy_menu_item) -> (copy_menu_item-edit_menu_item) -> (edit_menu_item-paste_menu_item) -> (paste_menu_item-edit_menu_item) -> (edit_menu_item-file_menu_item)
Edge Path 20: (file_menu_item-open_menu_item) -> (open_menu_item-file_name_edit) -> (file_name_edit-encode_item) -> (encode_item-file_name_edit) -> (file_name_edit-open_button) -> (open_button-file_menu_item)

Edge Path 21: (file_menu_item-edit_menu_item) -> (edit_menu_item-find_menu_item) -> (find_menu_item-up_item) -> (up_item-cancle_find) -> (cancle_find-edit_menu_item) -> (edit_menu_item-find_menu_item) -> (find_menu_item-down_item) -> (down_item-cancle_find) -> (cancle_find-edit_menu_item) -> (edit_menu_item-paste_menu_item) -> (paste_menu_item-edit_menu_item) -> (edit_menu_item-file_menu_item)

Edge Path 22: (file_menu_item-open_menu_item) -> (open_menu_item-encode_item) -> (encode_item-open_button) -> (open_button-file_menu_item)

Edge Path 23: (file_menu_item-open_menu_item) -> (open_menu_item-file_menu_item)

Edge Path 24: (file_menu_item-edit_menu_item) -> (edit_menu_item-copy_menu_item) -> (copy_menu_item-edit_menu_item) -> (edit_menu_item-find_menu_item) -> (find_menu_item-edit_menu_item) -> (edit_menu_item-paste_menu_item) -> (paste_menu_item-copy_menu_item) -> (copy_menu_item-edit_menu_item) -> (edit_menu_item-file_menu_item)

Edge Path 25: (file_menu_item-open_menu_item) -> (open_menu_item-file_type_item) -> (file_type_item-cancle_open) -> (cancle_open-file_menu_item)

Edge Path 26: (file_menu_item-edit_menu_item) -> (edit_menu_item-copy_menu_item) -> (copy_menu_item-edit_menu_item) -> (edit_menu_item-find_menu_item) -> (find_menu_item-find_edit) -> (find_edit-find_button) -> (find_button-edit_menu_item) -> (edit_menu_item-paste_menu_item) -> (paste_menu_item-edit_menu_item) -> (edit_menu_item-file_menu_item)

Edge Path 27: (file_menu_item-open_menu_item) -> (open_menu_item-file_name_edit) -> (file_name_edit-cancle_open) -> (cancle_open-file_menu_item)

Edge Path 28: (file_menu_item-edit_menu_item) -> (edit_menu_item-find_menu_item) -> (find_menu_item-case_check) -> (case_check-find_button) -> (find_button-edit_menu_item) -> (edit_menu_item-copy_menu_item) -> (copy_menu_item-edit_menu_item) -> (edit_menu_item-paste_menu_item) -> (paste_menu_item-edit_menu_item) -> (edit_menu_item-file_menu_item)

Edge Path 29: (file_menu_item-open_menu_item) -> (open_menu_item-encode_item) -> (encode_item-cancle_open) -> (cancle_open-file_menu_item)

Edge Path 30: (file_menu_item-edit_menu_item) -> (edit_menu_item-find_menu_item) -> (find_menu_item-up_item) -> (up_item-find_button) -> (find_button-edit_menu_item) -> (edit_menu_item-copy_menu_item) -> (copy_menu_item-edit_menu_item) -> (edit_menu_item-find_menu_item) -> (find_menu_item-down_item) -> (down_item-find_button) -> (find_button-edit_menu_item) -> (edit_menu_item-paste_menu_item) -> (paste_menu_item-edit_menu_item) -> (edit_menu_item-file_menu_item)
No of path generation | Unvisited edge coverage in Existing approach | Unvisited edge coverage in Proposed approach
--- | --- | ---
Path 1 | 35 | 11
Path 2 | 0 | 19
Path 3 | 0 | 2
Path 4 | 0 | 8
Path 5 | 0 | 2
Path 6 | 0 | 1
Path 7 | 0 | 1
Path 8 | 0 | 2
Path 9 | 0 | 1
Path 10 | 0 | 1
Path 11 | 0 | 1
Path 12 | 0 | 4
Path 13 | 0 | 1
Path 14 | 0 | 1
Path 15 | 0 | 2
Path 16 | 0 | 1
Path 17 | 0 | 1
Path 18 | 0 | 2
Total number of edges covered | 35 | 61

Table 7.2 Comparison of Event-interaction Coverage
Fig. 7.3 Comparison of edge coverage in existing and proposed approach

From the table and the generated paths it is proved that the proposed approach provide better coverage than the existing approach. The proposed approach generates the all events and all edges which enhance the testing of GUI system.

### 7.3 Case Study 2

There is another example which strengthens the proposed approach. Internet Explorer (formerly Microsoft Internet Explorer) abbreviated as IE is a series of graphical web browsers. Internet Explorer is a web based Graphical user interface which provides ease to the browsing all over the world. Internet Explorer (IE) is one of the most widely used web browsers.
7.4 EXPERIMENTAL RESULTS OF EVENT COVERAGE AND EVENT-INTERACTION COVERAGE

Experimental Results of Event Path Coverage:

Path 1: file_menu -> open_file -> file_type -> open_button -> cancel_open -> file_name -> cancel_add -> add_bookmark -> bookmark

Path 2: file_menu -> edit_menu -> find_menu -> match_whole -> find_menu -> match_case -> find_menu -> next -> find_menu -> previous -> find_menu

Path 3: file_menu -> open_file -> file_type -> open_button -> cancel_open -> file_name -> cancel_add -> add_bookmark -> add_button -> add_bookmark -> bookmark_name -> add_bookmark -> bookmark -> show_bookmark -> bookmark

Path 4: file_menu -> edit_menu -> copy -> edit_menu -> paste -> edit_menu -> cut -> edit_menu -> paste -> edit_menu -> copy -> edit_menu -> cut -> edit_menu

Path 5: file_menu -> new_window
Path 6: file_menu -> new_tab

<table>
<thead>
<tr>
<th>No of pathsgeneration</th>
<th>Event coverage in Proposed approach</th>
</tr>
</thead>
<tbody>
<tr>
<td>Path 1</td>
<td>9</td>
</tr>
<tr>
<td>Path 2</td>
<td>6</td>
</tr>
<tr>
<td>Path 3</td>
<td>3</td>
</tr>
<tr>
<td>Path 4</td>
<td>3</td>
</tr>
<tr>
<td>Path 5</td>
<td>1</td>
</tr>
<tr>
<td>Path 6</td>
<td>1</td>
</tr>
<tr>
<td><strong>Total number of events</strong></td>
<td><strong>23</strong></td>
</tr>
</tbody>
</table>

**Table 7.3  Event Coverage of Internet Explorer (IE)**

**Experimental Results of Edge Path coverage:**

**Edge Path 1:** (file_menu-open_file) -> (open_file-file_menu)

**Edge Path 2:** (file_menu-edit_menu) -> (edit_menu-find_menu) -> (find_menu-file_menu)

**Edge Path 3:** (file_menu-bookmark) -> (bookmark-file_menu)

**Edge Path 4:** (file_menu-open_file) -> (open_file-file_type) -> (file_type-file_menu)

**Edge Path 5:** (file_menu-edit_menu) -> (edit_menu-file_menu)

**Edge Path 6:** (file_menu-new_window) -> (new_window-file_menu)

**Edge Path 7:** (file_menu-new_tab) -> (new_tab-file_menu)

**Edge Path 8:** (file_menu-bookmark) -> (bookmark-add_bookmark) -> (add_bookmark-bookmark) -> (bookmark-file_menu)

**Edge Path 9:** (file_menu-open_file) -> (open_file-cancel_open) -> (cancel_open-file_type) -> (file_type-open_button) -> (open_button-file_type) -> (file_type-file_name) -> (file_name-file_type) -> (file_type-file_menu)
Edge Path 10: (file_menu-edit_menu) -> (edit_menu-find_menu) -> (find_menu-file_menu)

Edge Path 11: (file_menu-open_file) -> (open_file-open_button) -> (open_button-cancel_open) -> (cancel_open-open_button) -> (open_button-file_name) -> (file_name-open_button) -> (open_button-file_type) -> (file_type-open_button) -> (open_button-cancel_open) -> (cancel_open-file_name) -> (file_name-cancel_add) -> (cancel_add-file_menu)

Edge Path 12: (file_menu-edit_menu) -> (edit_menu-file_menu)

Edge Path 13: (file_menu-bookmark) -> (bookmark-add_bookmark) -> (add_bookmark-cancel_add) -> (cancel_add-add_bookmark) -> (add_bookmark-add_button) -> (add_button-file_menu)

Edge Path 14: (file_menu-new_window) -> (new_window-file_menu)

Edge Path 15: (file_menu-new_tab) -> (new_tab-file_menu)

Edge Path 16: (file_menu-open_file) -> (open_file-file_name) -> (file_name-file_type) -> (file_type-file_name) -> (file_name-open_button) -> (open_button-file_name) -> (file_name-cancel_add) -> (cancel_add-file_menu)

Edge Path 17: (file_menu-edit_menu) -> (edit_menu-paste) -> (paste-edit_menu) -> (edit_menu-copy) -> (copy-edit_menu) -> (edit_menu-cut) -> (cut-edit_menu) -> (edit_menu-find_menu) -> (find_menu-match_whole) -> (match_whole-find_menu) -> (find_menu-match_case) -> (match_case-find_menu) -> (find_menu-next) -> (next-find_menu) -> (find_menu-previous) -> (previous-find_menu) -> (find_menu-file_menu)

Edge Path 18: (file_menu-bookmark) -> (bookmark-show_bookmark) -> (show_bookmark-bookmark) -> (bookmark-file_menu)

Edge Path 19: (file_menu-open_file) -> (open_file-file_menu)

Edge Path 20: (file_menu-edit_menu) -> (edit_menu-file_menu)

Edge Path 21: (file_menu-bookmark) -> (bookmark-add_bookmark) -> (add_bookmark-bookmark_name) -> (bookmark_name-add_bookmark) -> (add_bookmark-bookmark) -> (bookmark-file_menu)

Edge Path 22: (file_menu-new_window) -> (new_window-file_menu)

Edge Path 23: (file_menu-new_tab) -> (new_tab-file_menu)

Edge Path 24: (file_menu-open_file) -> (open_file-file_type) -> (file_type-file_menu)
Edge Path 25: (file_menu-edit_menu) -> (edit_menu-find_menu) -> (find_menu-file_menu)

Edge Path 26: (file_menu-bookmark) -> (bookmark-add_bookmark) -> (add_bookmark-cancel_add) -> (cancel_add-add_bookmark) -> (add_bookmark-add_button) -> (add_button-add_bookmark) -> (add_bookmark-bookmark) -> (bookmark-show_bookmark) -> (show_bookmark-bookmark) -> (bookmark-file_menu)

<table>
<thead>
<tr>
<th>No of path generation</th>
<th>Unvisited edge coverage in Proposed approach</th>
</tr>
</thead>
<tbody>
<tr>
<td>Path 1</td>
<td>2</td>
</tr>
<tr>
<td>Path 2</td>
<td>3</td>
</tr>
<tr>
<td>Path 3</td>
<td>2</td>
</tr>
<tr>
<td>Path 4</td>
<td>2</td>
</tr>
<tr>
<td>Path 5</td>
<td>1</td>
</tr>
<tr>
<td>Path 6</td>
<td>2</td>
</tr>
<tr>
<td>Path 7</td>
<td>2</td>
</tr>
<tr>
<td>Path 8</td>
<td>2</td>
</tr>
<tr>
<td>Path 9</td>
<td>6</td>
</tr>
<tr>
<td>Path 10</td>
<td>8</td>
</tr>
<tr>
<td>Path 11</td>
<td>2</td>
</tr>
<tr>
<td>Path 12</td>
<td>2</td>
</tr>
<tr>
<td>Path 13</td>
<td>2</td>
</tr>
<tr>
<td>Path 14</td>
<td>14</td>
</tr>
<tr>
<td>Path 15</td>
<td>2</td>
</tr>
<tr>
<td>Path 16</td>
<td>2</td>
</tr>
<tr>
<td>Path 17</td>
<td>1</td>
</tr>
<tr>
<td><strong>Total number of edges covered</strong></td>
<td><strong>55</strong></td>
</tr>
</tbody>
</table>

Table 7.4 Event-interaction Coverage of Internet Explorer (IE)
From the resulted test paths of events it is concluded that all events in the event flow graph are visited at least once. As shown in the generated paths there is no event unvisited left behind so from these test paths it is concluded that event coverage criterion is satisfied.

As discussed earlier that event coverage criterion is not enough to detect the faults in software system so we have to choose stronger coverage criterion than the simple event coverage criterion. For this purpose event-interaction (event-pair) coverage criterion is used. This criterion ensures that all the events and the edges of the event flow graph are traversed. If all edges or event pair relation are visited this confirms that every event and every edge is covered so these are less chance of error left. Since all the events and edges between events are covered it means that functionality of the software system are tested fully and the system will become fault tolerant.
8 RESULTS AND DISCUSSION

In this chapter we will describe and calculate the appropriate ability of the proposed approach. This chapter includes the comparison of the existing and the proposed approach then presents the results of proposed approach.

In this research we have covered the event coverage and event-interaction coverage criteria which calculate all paths having events and their edge coverage. This coverage ensures that all events and their follow-relation event are covered from all edge views.

In the previous chapter we discussed the parameters and how we performed the experiment in detail. We also showed the results of selected examples after performing experimentation. Now in this chapter we will compare our results of proposed approaches with Huang et al. results.

8.1 ANALYSIS OF EXPERIMENTAL RESULTS

When ACO is performed on the Event flow Graph (EFG) all events that are selected are constructed based on calculating the probability of every event. Then we find out the results that are given below.

We also implemented the Huang et al. work and found out the results, and then we compared our work with the work of Huang et al. work. Results show that event coverage gives the same result as Huang et al. (2011) proposed, but the event-interaction coverage criteria gives better results than Huang et al. results. It means that our proposed method are better than Huang et al. method.

For easy illustration, EFG example in Fig. 7.1 is used for our experiment result. This EFG contains 17 events and 61 follows as shown in the Fig. 7.1
We can see the results of Fig. 7.1 that all 17 events and 61 follow relations are visited. Huang et al. (2011) only covered 35 out of 61 follow relations. Some research used simple graph traversing algorithm to generate longer test cases. However, some paths on the EFG are not available because of the event context. For example, if use Depth-First-Search algorithm, path selected to PasteMenuItem is: FileMenuItem -> EditMenuItem -> PasteMenuItem -> CopyMenuItem. This path is infeasible because PasteMenuItem is not enabled. PasteMenuItem can be enabled only when CopyMenuItem is performed before it. To overcome the infeasible problem the depending paths are identified earlier and describe their depending relation.

Although the length of test cases has no significant impact on the number of faults detected longer test cases, of course, detects the same faults as those shorter test cases which contained in them. In our approach we constructed multiple test paths instead on one single path.

<table>
<thead>
<tr>
<th></th>
<th>Total Events</th>
<th>Events Covered</th>
<th>Total Edges</th>
<th>Edges Covered</th>
</tr>
</thead>
<tbody>
<tr>
<td>Huang et al.</td>
<td>17</td>
<td>17</td>
<td>61</td>
<td>35</td>
</tr>
<tr>
<td>Proposed Approach</td>
<td>17</td>
<td>17</td>
<td>61</td>
<td>61</td>
</tr>
</tbody>
</table>

Table 8.1 Comparison of Experimental Results
As shown in the table 8.1 that the event coverage fulfill the criteria of covering all the events at least once. In the Fig. 7.1 there are total 17 events in the EFG and the resulted output also covers the 17 events this means the event coverage is satisfied. Existing approach generated one long path to cover all events but in our approach we have generated multiple paths which has also covered all the events. In the existing methodology there are total 17 events and 61 event-pair relation. To achieve the full coverage that make the software fault free we have to cover all these 17 events and 61 event-pair relation (follow relation events). In our proposed approach these criteria are enhanced and instead of covering 35 out of 61 follow relation, 61 follow relation events are covered. From the above table it is concluded that our proposed approach is better than the existing approach.

<table>
<thead>
<tr>
<th>Case study 2</th>
<th>Total Events</th>
<th>Events Covered</th>
<th>Total Edges</th>
<th>Edges Covered</th>
</tr>
</thead>
<tbody>
<tr>
<td>Proposed Approach</td>
<td>23</td>
<td>23</td>
<td>55</td>
<td>55</td>
</tr>
</tbody>
</table>

Table 8.2 Coverage criterion for internet explorer
From the above table 8.2 it is concluded that the proposed approach covers all events along with the event-interaction coverage. This coverage criterion is stronger than the simple event coverage criterion because it covers the edges and events both.

Simple event coverage might be miss some edges or follow-relation of the events. That’s why for adequate testing we need stronger coverage criterion which provide full events and full edge coverage. From the experimental result it is shown that the existing methodology only cover events of the Event Flow Graph (EFG) leaving some follow relation unvisited. But the proposed approach cover all events of the EFG and also provide full coverage to traverse the edges of the EFG. Hence proved from the experimental result that the coverage criterion in the proposed approach is better than the existing approach.

There are many GUI path coverage criterion that are used for GUI testing. Among all of them Event coverage criterion covers all the events which shows the GUI objects and components. For stronger GUI testing Coverage criterion event-interaction coverage criterion is used that covers all the edges which shows the follow-relation
events (Child events). In event coverage all events are covered but might be some edges are left uncovered. These uncovered edges might produce faults in software system. To cover these edges which represent the dependencies between different edges, event-interaction coverage criterion is used. It is proved from the case studies that existing approach used only event coverage so there is a change of error. To overcome this missing or uncovered edges, event-interaction coverage is used. The test path covers all feasible events and all their feasible edges.
9 CONCLUSION AND FUTURE WORK

In this chapter we summarize our main contribution for this thesis work and also give some future work direction.

9.1 CONCLUSION

GUI testing is an important phase in software development which ensures the software quality, performance and reliability. Automated software testing makes software testing more efficient and less error prone. In our work we are implementing feasible test paths generation. Most of work has done on automated test data generation for coverage criterion and generation of feasible test paths. One of the authors worked on coverage criterion for effective testing but does not perform the full coverage for event flow graph (EFG).

So, our main contribution is to generate the test path having full coverage and eliminating the infeasible paths using Ant Colony Optimization (ACO) algorithm. In proposed approach event-interaction coverage criterion is used to provide full coverage which covers events and their adjacent edges (child events). We perform the experimentation to validate our methodology. We compare our results with other work Huang et al. Our methodology are effective and efficient than huang et al approach. Existing approach provide only events coverage while missing some edges uncovered. Our hypothesis accepted that accuracy is improved due to event-interaction coverage criterion. Feasible path generation and event-interaction coverage using ACO both are automated.

9.2 FUTURE WORK

We will apply other GUI coverage criterion for efficient test path generation on large and complex case study. Our main focus is generation of feasible test paths. We will also consider the infeasible paths repairing in future work. We have selected initial event but
we are not considering the initial event randomly. So, it will be also our future work that we will select initial node randomly on the basis of event availability. This means event is not depending upon the other event when selection of event is random.

Generation of all coverage criterion and all feasible paths from program is automated but event flow graph representation in XML form is not automated which will consider as future work for automation.

10 References


Memon, Atif M. "GUI testing: Pitfalls and process."Computer vol. 35.8 pp. 87-88, 2001


Yuan, Xun, and Atif M. Memon. "Generating event sequence-based test cases using GUI runtime state feedback." Software Engineering, IEEE Transactions on Vol.36 No.1 pp. 81-95, 2010


