FULLY AUTOMATED GUI TESTING AND COVERAGE ANALYSIS USING GENETIC ALGORITHMS

ABDUL RAUF, ARFAN JAFFAR AND ARSHAD ALI SHAHID

Department of Computer Science
National University of Computer and Emerging Sciences
Islamabad, Pakistan
{a.rauf; arfan.jaffer; arshad.ali}@nu.edu.pk

Received November 2009; revised March 2010

Abstract. Graphical user interface (GUI), is a graphical front-end to a software system, contains graphical objects with certain distinct values whose purpose is to ascertain the state of the GUI at any time. In order to ensure that the quality of the software is par excellence, software developing organizations endeavor to test the software meticulously. Nevertheless, the process of testing a GUI application calls for a colossal effort, owing on account of the intricacy entailed in such applications. Subsequently, organizations were spurred to initiate the automation of GUI testing, thereby proposing various techniques to achieve this end. A GUI model event-flow graph, an innovative technique being utilized in the field of automated GUI testing, represents, likewise control flow graph, all promising progressions of events that can be executed on GUI. The search for utmost quality insurance for software, through the introduction of automated software testing, raises yet another challenging question, that of the “amount” of testing required so as procuring the best results. In the course of the development of the techniques for the automation of the software testing procedure, a few measures can be employed to provide guidance on the quality of an automatic test suite. Based on some predefined test criterion, genetic algorithm searches for the best possible test parameter combinations. Usually, this test criterion corresponds to a “coverage function” that measures how much of the automatically generated optimization parameters satisfies the given test criterion. In this paper, we have attempted to exploit the event driven nature of GUI. Based on this nature, we have presented a GUI testing and coverage analysis technique centered on genetic algorithms.

Keywords: GUI testing, Genetic algorithm, Coverage criterion, Coverage analysis, Event flow, Test data generation, Test path, Automation testing

1. Introduction. Quality of the delivered software depends heavily on the systematic activity of software testing. Testing related activities go on with the entire development life cycle and may consume a large fraction of the effort required for producing software [1]. Purpose of software testing is to improve software quality and increases confidence in software’s proper functioning. This purpose is achieved with support of software testing activities [2]; these activities include collecting test data, generation and execution of test cases, filtration and reduction of test cases, coverage analysis and reporting. Software testing is a labor intensive process and studies indicate that more than 50 Graphical user interfaces (GUIs) are one of the most important components of modern day software and are being considered as necessary part for most of today’s software. GUIs give user a relatively more ease and freedom to interact while accessing with the system [3]. Recognizing the importance of GUIs, software developers are dedicating more effort, up to 50 Common practice of GUI test designers is to generate and execute test cases to traverse parts of GUI application. These test cases need to focus on a subspace in order
to maximize fault detection in an efficient manner. Graphical user interfaces (GUIs) can be considered as a collection of widgets associated with event handlers where event handlers are assigned the task of responding to individual events. This response can vary depending on the current state of the GUI, which is established by preceding events and their execution order. The degree of freedom offered by GUIs to end users can be visualized as allowable number of permutations of GUI input events which in most nontrivial applications is extremely very large. We also have to be mindful of the fact that events in GUIs have complex interactions. A situation to elaborate on this fact is that a user interacting with a GUI may execute an event sequence X that puts the GUI in such a state that a subsequent event sequence Y causes erroneous execution”. The thing to understand is that unless a context was established by the event sequence X, the event sequence Y might not have led to the error. The research has shown that many GUI events may or may not exhibit similar behavior. These events cause an error in the GUI in one context but not in another context [3]. How much to test? Or determining the coverage criteria for software testing and especially for GUI testing has always been a challenging question. Any test designer must be assured that its test suit is sufficient to test a software or GUI component. Unlike a CLI (command line interface) system, a GUI has many operations that need to be tested. A very small program such as Microsoft WordPad has 325 possible GUI operations [1]. The number of operations can easily be an order of magnitude larger in case of larger programs. Automated GUI testing has been facing this problem. To overcome this problem, D. J. Kasik and H. G. George introduced an interesting method of generating GUI test cases. This method uses the theory that good GUI test coverage can be attained by simulating a novice user [6]. According to their theory, one can hypothesize that a skilled user of a system will go after a very direct and conventional path all the way through a GUI and a beginner user would follow a relatively random path. Artificial Intelligence is going through period of extensive research and implementation in the past few years [40]. Recently, heuristics and meta-heuristics have been researched and used many times to develop different types of algorithms [39,42]. Genetic Algorithm can be used for finding out optimized test suite for GUI testing. Genetic algorithms work on the basis of ‘genes’, which are created randomly and then are subjected to some task. Genes with good performance are kept for next phases, while others are discarded. In case of testing, Genetic algorithm searches for optimal test parameter combinations that satisfy a predefined test criterion. This test criterion is represented through a “coverage function” that measures how much of the automatically generated optimization parameters satisfies the given test criterion. Genes optimizing the coverage function will survive and others will be discarded, the process is repeated again and again with optimized genes being replicated and more random genes will take place of discarded genes. Ultimately one gene (or a small group of genes) will be in the set and is logically the best fit for coverage function. We have proposed a technique based on event flow nature of GUI that analysis coverage based on these recorded events. This technique has shown very good results in analyzing coverage of GUI’s and works equally well on simulating a novice user idea as well. The major contributions of our proposed technique are following:

- Automatic event sequence recordings.
- Test coverage analysis is fully automated.
- GA has been used for optimization of test coverage.
- Two indigenous while three off the shelf applications were selected to experiment with.
- Results of the experiments are very encouraging and promising.
The remainder of the paper is organized as follows: In Section 2, we discuss related work in the field of software testing, GUI testing and optimization techniques. Section 3 describes proposed method in detail. Section 4 presents results of experiments related to test case prioritization and maximizing coverage, while in Section 5, some future directions have been presented and Section 6 concludes the paper.

2. Related Work. The functional accuracy of a graphical user Interface is necessary to assure the security, robustness and usability of an entire software system. However, at the same time, testing is the most expensive phase of Software Development Life Cycle (SDLC), as it nearly consumes two-third of the software development resources [14]. GUI testing techniques are very much resource intensive. Most of the techniques used to test GUIs are being extended from techniques that were used to test CLI programs earlier. However, most of these extensions are not as successful when they are applied to GUI's as they are in case of software. For example, Finite State Machine-based modeling becomes too complex and unmanageable for a GUI but can work well on a system that has a limited number of states. Although model based techniques have been used frequently for software testing. However, models are very expensive to create, and their applicability is limited as well. For these reasons, model based techniques are not being used for GUI testing frequently. However, in past few years, efforts have been made for developing different models for GUI testing. A. M. Memon and his team have worked a lot in automated GUI testing [8,9]. They have used several types of graph models (e.g., event-flow graphs) to generate specific types of test cases [8,9]. In [3], authors combine all of the models into one scalable event-flow model and outlines algorithms to semi-automatically reverse-engineer the model from an implementation. This model defines event-space exploration strategies (ESEEs) and creates an end-to-end GUI testing process. They also developed a model called the event-flow graph (EFG) that represents the space of all possible event sequences that may be executed on the GUI [3,4]. A. M. Memon and Xie also created an event-interaction graph (EIG) [3,4]. Kasik and George [11] had a novice idea to resemble novice GUI users. For this purpose, they have used genetic algorithms. In this approach, an expert manually generates a sequence of GUI events, and then uses the genetic algorithms to modify and lengthen the sequence. This approach relies on an assumption that novice users take longer “paths” through the input event interaction space when performing activities while in contrast expert users take a bit shorter paths [11]. L. White, H. Almezen and N. Alzeidi have developed a technique to address the User-based testing of GUI sequences and their interaction [12]. L. White and H. Almezen have also given techniques for Generating test cases for GUI responsibilities using complete interaction sequences [13]. In [9,10,14,15], A. M. Memon has proposed some models and developed some techniques to address the automation of specific aspects of the GUI testing process, test-oracle creation [16] and regression testing [17]). In [14,15], A. M. Memon has used goal-directed search for GUI test case generation while in [EF 14], author has used function composition for automated test-oracle creation. A. M. Memon also used metrics from graph theory to define test coverage criteria for GUIs [18], graph-traversal to obtain smoke test cases for GUIs that are used to stabilize daily software builds [9,19], and graph matching algorithms to repair previously unusable GUI test cases for regression testing [17]. There has been a growing interest in developing models to automate GUI testing. The most popular models for this purpose are state-machine models that have been proposed to generate test cases for GUIs [20-23]. The major inspiration for using these models is that a test designer simulates a GUI’s behavior as a state machine; each input event may trigger an abstract state transition in the machine. A path, i.e., sequence of edges followed during transitions, in the state machine represents a test case [20-23]. The state machine’s
abstract states may be used to verify the GUI’s concrete state during test case execution [20-23]. Shehady and Siewiorek [24] have developed variable Finite State Machines (FSMs) that decrease the number of abstract states by adding variables to the model. They argue that regularly used FSMs have extension problems for large GUIs [24]. White et al. present a different state-machine model for GUI test-case generation that divides the state space into different user tasks and relates these tasks to different complete interaction sequence (CIS) [12,13]. The test engineer classifies a user task that can be performed with the GUI. For each user task, the test engineer then identifies a machine model called the ‘complete interaction sequence’ (CIS). The CIS is then used to generate relative test cases. This approach is successful in partitioning the GUI into manageable functional units that can be tested separately [12,13]. The definition of responsibilities and the subsequent CIS is relatively straightforward; the large manual effort is in designing the FSM model for testing, especially when code is not available [12,13]. A genetic algorithm (GA’s) is especially appropriate to the solution of indefinite problems or nonlinear complex problems [40]. The critical impression of genetic algorithms (GA’s) is to replicate the progression law of nature’s unrefined struggle and natural selection. GA is competent enough to select the better species from the mother generation and randomly interchanging gene information in order to produce a better generation [38]. With steady fruition, the track would grant a generation that is best accustomed to the environment [38]. There have been a number of studies that use genetic algorithms (GA’s) for software testing. Jones et al. proposed a technique to generate test-data for branch coverage using GA [25,26]. This technique has shown good results with number of small programs. This technique uses the acyclic control-flow graph (CFG) to guide the search and the fitness value is based on the branch value and the branching condition. Michael et al. have developed a tool for generating test data on basis of four different algorithms [27]. Two of these algorithms were based on genetic algorithm. They named this tool as Gadget. This tool gives good condition/decision coverage coverage of C/C++ code [27]. Gadget requires that each branch in the code should be taken and that every condition (atomic part of a control-flow affecting expression) in the code should be true at least once and false at least once. Pargas et al. used a GA based on control dependence graph to search for test data giving good coverage [28]. They used the original test suite developed for the SUT as the seed for the GA. They compare their system to random testing on six small C programs. For the smallest programs there is no difference but for the three largest programs the GA-based method outperforms random testing. Tracey et al. presents a framework for test-data generation based on optimization algorithms for structural testing [29]. Also Tracey has used a similar technique for functional (black-box) testing. The research by Tracey et al. is unique in that they have evaluated their techniques on a real-world safety-critical system [29]. Y. Lu et al. presented a new GUI automation test model based on the event-flow graph modeling [30]. In this model, authors have presented a technique to generate test cases in the daily smoke test based on an improved ACO and a spanning tree is utilized to create test cases in the deep regression test [30]. Very little literature is available which discusses coverage criteria for GUI testing. In [32,33], authors have given an idea for coverage criteria based on events. Authors have described two different categories for coverage criteria, i.e., inter-component coverage and intra component coverage. Event coverage, Event-interaction coverage and Length-n event sequence coverage can be used for intra-component coverage, while Invocation coverage, Invocation-termination coverage and Inter-component length-n coverage can be used for inter-component coverage. In [32], authors have concluded that for GUI testing, coverage criterion based on events can be useful. In [32], authors have presented a correlation between event based coverage of a GUI and statement coverage of its software’s underlying code. Authors have shown
that more than 90. In [34], authors introduced the concept of systematically testing GUI applications using symbolic execution. Authors also made claim that communication between users and GUIs is event driven. Authors have shown that randomly generated test suite showed high coverage only if its size is twenty times larger. And results based on symbolic execution achieve 100. S. Ono et al. proposed a robust solution search method using Multi-Objective Genetic Algorithm (MOGA) to maximize the local search facility and minimize search cost [37]. This robust optimization helps to reduce the margins of errors, noises and other uncertainties on manufacture, design, observation [37]. Thus helps to achieve higher quality of results.

3. Proposed Method. A GUI is a hierarchical, graphical front-end to a software system that accepts as input user-generated and system-generated events from a fixed set of events and produces deterministic graphical output. A GUI contains graphical objects; each object has a fixed set of properties. At any time during the execution of the GUI, these properties have discrete values, the set of which constitutes the state of the GUI. To test GUI and analyze the coverage, we have proposed a method based upon Genetic Algorithms. We have divided our proposed system into three major blocks.

- Test data generation.
- Path Coverage Analysis.
- Optimization of Test Paths.

Working of genetic algorithm has been explained with the help of a block diagram in Figure 1.

3.1. Test data generation. Using events to generate date for GUIs testing is now becoming a common practice. For test data generation, we have also used event based techniques. A user developed calculator that receives inputs both from mouse as well as from the keyboard has been used as the first application to test our approach. Interface of calculator has been shown in Figure 2. For every event, there is a unique event ID as
shown in Figure 3. As an event occurs by the help of mouse or a key stroke, respective event ID gets added into event recorder.

After completion of user interaction with the calculator, a sequence of events is formulated, this is passed to next phase for further analysis. Sequence of generated events has been shown in Figure 4.

Another application that was used for experimentation was a user developed Notepad. This application also works on same principles as discussed above, i.e., events recording on basis of unique ID’s and-formulating sequences from these events. A user can interact with the application in same way as Microsoft’s notepad. Somehow the added functionality was that, each interaction of user is being recorded and a unique code is being generated.
for each mouse clicks or keyboard button being pressed. Interface of notepad is shown in Figure 5.

Having good results from our efforts with two user developed applications, we tried to generalize our approach. For this purpose, we have selected Microsoft’s Notepad as an off the shelf GUI product for testing. We have decomposed GUI into hierarchy modal that consists of nodes which represents objects like file is an object that have been represented as a node in our hierarchy modal. Connection between nodes represents the path between objects, e.g. To print a document we have to follow a sequence of events like first of all click file then it displays different objects, selecting print option from those objects. So to print a document we have to follow sequence of file to print. In this way, a hierarchy has been designed that represents the sequences of paths between different objects. Table 1 depicts possible path sequences from each tab in notepad and unique code defined against each of these tab options. Figure 6 shows path generation for Notepad on the basis of possible sequences of events.
3.2. Optimization of test paths using genetic algorithm. Genetic algorithms are inspired by Darwin’s theory about evolution. Solution to a problem solved by genetic algorithms is evolved. Algorithm is started with a set of solutions (represented by chromosomes) called population. Solutions from one population are taken and used to form a new population. This is motivated by a hope, that the new population will be better than the old one. Solutions which are selected to form new solutions (offspring) are selected according to their fitness – the more suitable they are the more chances they have to be reproduced. This is repeated until some terminating condition (for example, number of populations or no improvement of the best solution after certain iterations) is satisfied. Idea of evolutionary computing was introduced in the 1960s by I. Rechenberg in his work “Evolution strategies” (Evolutions strategies in original). His idea was then developed by other researchers. Genetic Algorithms (GAs) were invented by John Holland and developed by him and his students and colleagues. This lead to Holland’s book “Adaption in Natural and Artificial Systems” published in 1975 [31]. Although GA falls in the category of search methods but there are many differences between GA and traditional search methods [40]. These differences include GA does work with a coding modality; Search in GA is from a solution group to another group in solution space [40].

Following are steps of GA:

1. **[Start]** Generate random population of n chromosomes. Length of our chromosome is the longest path. We have initialized these chromosomes between 1 and maximum length.

   Example:
   
   \[
   \begin{bmatrix}
   2 & 4 & 3 & 1 & 2 
   \end{bmatrix}
   \]

2. **[Fitness]** Evaluate the fitness \( f(x) \) of each chromosome \( x \) in the population. We have calculated fitness of chromosome based upon the coverage analysis (How paths have been covered by a chromosome).

3. **[New population]** Create a new population by repeating following steps until the new population is complete.

   (a) **[Selection]** Select two parent chromosomes from a population according to their fitness (the better fitness, the bigger chance to be selected)
Table 2. Test paths with length

<table>
<thead>
<tr>
<th>No</th>
<th>Test Path</th>
<th>Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1, 2</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>1, 9, 2</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>1, 8, 2</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>1, 8, 4, 2</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>1, 8, 3, 4, 2</td>
<td>5</td>
</tr>
</tbody>
</table>

(b) [Crossover] With a crossover probability cross over the parents to form a new offspring (children). If no crossover was performed, offspring is an exact copy of parents.

(c) [Mutation] With a mutation probability mutate new offspring at each locus (position in chromosome).

(d) [Accepting] Place new offspring in a new population

4. [Replace] Use new generated population for a further run of algorithm.

5. [Test] If the end condition is satisfied, stop, and return the best solution in current population.


3.3. **Fitness function.** Given an input program, the fitness function returns a number indicating the acceptability of the program. The fitness function is used by the selection algorithm to determine which variants survive to the next iteration (generation), and it is used as a termination criterion for the search. Our fitness function is how much test cases have successfully validated?

Accuracy = Test Paths covered by chromosome/ Total number of chromosome

Example:

Let us explain the working of genetic algorithm with the help of one example. Table 2 represents available test paths and lengths of these test paths, we have to check that how many paths are being covered by our chosen chromosomes. This will tell us fitness function of each of the chosen chromosome.

Let us take a chromosome in which genes represent the sequence of path.

| 1 | 8 | 7 | 4 | 2 | 9 | 6 | 8 |

Fitness of above chromosome is evaluated using Equation (3).

Table 2 shows that total number of test paths are 5. Out of these 5 test paths, path 1, path 3 and path 4 are covered by the chromosome so the fitness or accuracy of this chromosome is 3/5 = 0.6.

3.4. **Reproduction operators.** There are two reproduction operators available in genetic algorithm: Cross over and Mutation. Cross over has two different types, one point cross over and two points cross over. However, we will apply these reproduction operators to increase the coverage efficiency.

Now, we take the second chromosome;

| 6 | 3 | 2 | 7 | 2 | 1 | 9 | 2 |

Its fitness function = 1/5 = 0.2.

Now we will generate a random number to find the cross over point.

Rand = 5

As random number is 5, so we will cut chromosome 1 after 5 genes and will combine it 2nd chromosome to generate a child chromosome.

| 1 | 8 | 7 | 4 | 2 | 1 | 9 | 2 |

Now, fitness function of the child chromosome = 4/5 = 0.8.
Which is much better than fitness function of chromosome 1 (which was 0.4) and also of chromosome 2 (0.2).

3.5. **Mutation.**

\[
\begin{bmatrix}
1 & 8 & 7 & 4 & 2 & 1 & 9 & 2
\end{bmatrix}
\]

\[\text{Rand } = 3 \text{ for position}
\]

\[
\begin{bmatrix}
1 & 8 & 3 & 4 & 2 & 1 & 9 & 2
\end{bmatrix}
\]

Fitness function = 5/5 = 1.

4. **Experimental Results.** The proposed application for coverage analysis was designed and developed in MATLAB. The application has undergone extensive experimentation in order to determine its effectiveness. Five sample applications were selected to experiment with which included two user developed applications of calculator and notepad while three built-in products MS Notepad, MS Wordpad and MS WORD were chosen from Microsoft family of software products. These built-in applications were selected keeping in mind the following criteria:

- **Universal Applicability:** Applications have universal applicability. Our working on these applications demonstrates the capability of our approach to handle such applications which are complex in their nature and affect a larger population of end-users. This also means that interpretation of experiments and results is easier for larger research community.

- **Rich GUI:** These applications come with extensive GUI which provides us with ideal environment to execute and monitor effectiveness of our technique. The GUI of NotePad and WordPad is relatively simple yet effective. GUI of both these applications conforms to variety of usability engineering standards. Successful performance of our proposed approach can demonstrate the vitality of various usability engineering and HCI protocols from testing perspective.

- **Wider applicability:** These applications are part of a larger application domain. By testing our technique on these applications, we can also replicate the generated test cases on several other applications to broaden the scope of our exploration.

- **Long Term Perspective:** NotePad, WordPad and MS Word are part of the application domains which have a long term perspective, i.e., we can expect many future versions of both of these applications. Having such quality products as our test applications means that we have an opportunity to evolve our techniques as the applications evolve incorporating new concepts of GUI.

- **Highly structured applications:** The applications are highly structured which provides us with an opportunity to design chromosomes for various tests quite efficiently.

The test data was generated by clicking on various GUI elements and tailoring the course of click-events. This proved to be a laborious task as significant manual effort was required to generate an appropriate number of test cases. In all, 45 test cases were generated per application. These test cases manipulated various aspects of product interface, i.e., menus, toolbars, drop-down bars, etc. The composition of test suite was tried to represent a balance set of test suite such that it evenly covered all of these aspects of each product. Each test case was of variable length depending upon the sequence of events involved to perform that specific action.

Coverage analysis has shown that system was able to achieve more than 85% coverage. Fitness function was able to yield high coverage which shows its utility in the case of GUI testing. This coverage percentage shows that we still have significant room for improvement. Still, achieving such a high coverage makes our technique competitive
Table 3. Parameter used

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Population size</td>
<td>100</td>
</tr>
<tr>
<td>Number of generations</td>
<td>300-500</td>
</tr>
<tr>
<td>Mutation rate</td>
<td>0.2</td>
</tr>
<tr>
<td>Crossover rate</td>
<td>0.8</td>
</tr>
<tr>
<td>Termination criteria</td>
<td>Coverage &gt; 83% or Generation = 500</td>
</tr>
</tbody>
</table>

Table 4. Coverage according to number of generations

<table>
<thead>
<tr>
<th>Number</th>
<th>Coverage achieved</th>
<th>MS Notepad</th>
<th>MS Wordpad</th>
<th>MS Word</th>
<th>User Defined Notepad</th>
<th>Calculator</th>
<th>Average Coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>300</td>
<td></td>
<td>65%</td>
<td>68%</td>
<td>59%</td>
<td>71%</td>
<td>76%</td>
<td>68%</td>
</tr>
<tr>
<td>325</td>
<td></td>
<td>68%</td>
<td>68%</td>
<td>67%</td>
<td>75%</td>
<td>77%</td>
<td>71%</td>
</tr>
<tr>
<td>350</td>
<td></td>
<td>69%</td>
<td>69%</td>
<td>76%</td>
<td>77%</td>
<td>74%</td>
<td>74%</td>
</tr>
<tr>
<td>375</td>
<td></td>
<td>72%</td>
<td>69%</td>
<td>80%</td>
<td>77%</td>
<td>84%</td>
<td>76%</td>
</tr>
<tr>
<td>400</td>
<td></td>
<td>73%</td>
<td>71%</td>
<td>84%</td>
<td>84%</td>
<td>85%</td>
<td>79%</td>
</tr>
<tr>
<td>425</td>
<td></td>
<td>79%</td>
<td>76%</td>
<td>84%</td>
<td>84%</td>
<td>89%</td>
<td>82%</td>
</tr>
<tr>
<td>450</td>
<td></td>
<td>85%</td>
<td>77%</td>
<td>86%</td>
<td>88%</td>
<td>89%</td>
<td>85%</td>
</tr>
<tr>
<td>475</td>
<td></td>
<td>85%</td>
<td>84%</td>
<td>86%</td>
<td>88%</td>
<td>89%</td>
<td>86%</td>
</tr>
<tr>
<td>500</td>
<td></td>
<td>85%</td>
<td>84%</td>
<td>86%</td>
<td>88%</td>
<td>89%</td>
<td>86%</td>
</tr>
</tbody>
</table>

![Figure 7. Path coverage analysis](image)

The results have also shown that increase in number of generations resulted in enhanced coverage. To determine optimal number of generations, we experimented with our technique using generations between 300 and 500. Our experiments have shown that increase in number of generations above this range generates a flat bed scenario. It means that with increase in number of generations, the performance does not deteriorate, however, becomes stable at the highest coverage achieved. The effect of enhancing the number of generations is shown in Table 4.

The graphical representation of this improvement achieved in coverage is shown in Figure 7. This has shown gradual improvement until it reaches a saturation point. After reaching this saturation point, it becomes stable and adopts a flat bath instead of deteriorating.

w.r.t. other existing approaches. The details of parameters used in experimentation during testing each application are shown in Table 3.
The results have shown the overall effectiveness and improvement that our proposed technique has achieved in effective coverage analysis. We are in the process of generating further test cases for the same applications to further examine the performance of our approach for coverage analysis.

5. Future Work. In this paper, we have used manual test case generation, we are now in the process of developing an automated test generation tool for supporting our approach which will further increase its utility. We also plan to use other artificially intelligent optimization techniques to further enhance its effectiveness. We are currently in the process of developing such condition through applying of which, we can further extend our test coverage.

Another future area of work is to develop an efficient algorithm which ensures certain high degree of precision along with test coverage. Our aim is to extend this technique in such a way that it is automatically able to generate correct test data for the complete test coverage. One possible extension in this direction can be to use Design for Multi-Objective Six Sigma (DFMOSS) [37]. There might be some margins of errors like noise and other uncertainties on manufacture, design and in observation so DFMOSS can help in overcoming these issues. Some recent studies have discovered that two different species can co-evolve with each other. This mechanism is called coevolution and this has yielded encouraging outcomes in developments of GA and GP [41]. We also plan to use coevolution to optimize our results. One more area of interest can be complex human social behaviors inspired optimization algorithms [43].

In this paper, our focus was exclusively on maximizing coverage. In future, we also aim to consider other factors like cost and number of test cases. For this purpose, we can use multi-objective algorithms like NSGA-ii, Multi Objective PSO or a recently proposed new evolutionary algorithm for multi-objective constrained optimization (MCOP) [44].

6. Conclusion. Graphical User Interface testing has always been considered a critical element of overall testing paradigm for software applications. In this paper, we have proposed a genetic algorithm based technique for coverage analysis of GUI testing. The technique has been subjected to extensive testing. Five simple applications were selected to experiment with which included Notepad, WordPad and MS WORD. And the experiments have shown encouraging results. The results have also shown enhanced coverage increase in number of generations. The proposed technique offers an exciting new area of research which can be applied using different other artificial intelligence techniques.

Acknowledgement. The first author is sponsored by Higher Education Commission of Pakistan, Government of Pakistan, and wishes to acknowledge HEC for funding.

REFERENCES


[34] A. M. Memon, *Determining the Adequacy of GUI Test Case*, University of Maryland, 2002.