2010 Seventh International Conference on Information Technology

Automated GUI Test Coverage Analysis using GA

Abdul Rauf
Department of Computer Science
National university of Computer & Emerging Sciences
Islamabad, Pakistan
a.rauf@nu.edu.pk

Sajid Anwar
Department of Computer Science
National university of Computer & Emerging Sciences
Islamabad, Pakistan
sajid.anwar@nu.edu.pk

M. Arfan Jaffer
Department of Computer Science
National university of Computer & Emerging Sciences
Islamabad, Pakistan
arfan.jaffar@nu.edu.pk

Arshad Ali Shahid
Department of Computer Science
National university of Computer & Emerging Sciences
Islamabad, Pakistan
arshad.ali@nu.edu.pk

***Abstract*— A Graphical User Interface (GUI) is a graphical front-end to a software system. A GUI contains graphical objects with certain distinct values which can be used to determine the state of the GUI at any time. Software developing organizations always desire to test the software thoroughly to get maximum confidence about its quality. But this requires gigantic effort to test a GUI application due to the complexity involved in such applications. This problem has led to the automation of GUI testing and different techniques have been proposed for automated GUI Testing. Event-flow graph is a fresh technique being used in the field of automated GUI testing. Just as control-flow graph, another GUI model that represents all possible execution paths in a program, event-flow model, in the same way, represents all promising progressions of events that can be executed on the GUI. Another challenging question in software testing is, "How much testing is enough?" As development proceeds, there are fewer measures available that can be used to provide guidance on the quality of an automatic test suite. Genetic algorithm searches for the best possible test parameter combinations that are according to some predefined test criterion. 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

I. INTRODUCTION

Software testing is a systematic process of software development life cycle on which the quality of the delivered software depends. Testing related activities encompass the entire development process and may exhaust a large part of the effort required for producing the software [1]. The Purpose of software testing is to improve software quality and increase confidence in the software's proper functioning.

Graphical user interfaces (GUIs) are one of the most important components of modern day software. The need to test GUI stems from the fact that a typical GUI gives various degrees of freedom to an end-user. A test designer needs to develop test cases (sequences of events) that examine the enormous input interaction space of the GUI. However, in reality, it is not a common practice to test up to this level. Testers mostly take up capture/replay tools [4].

A Common practice among GUI test designers is to generate and execute test cases to traverse its parts. 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 software which is established by preceding events and their execution order. The degree of freedom offered by GUIs to end users can be visualized by the available number of permutations of GUI input events which, in most nontrivial applications, is extremely large. Moreover, one also has 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 a state such that a subsequent event sequence Y causes erroneous execution". The point to understand is that unless a context had been established by the event sequence X, the event sequence Y would not have led to the error. The research has shown that many GUI events may or may not exhibit similar behaviour. These events cause an error in the GUI in one context but not in another context [3]. The amount of testing required and the determination of the coverage criteria for software testing, and especially for GUI testing, has always been a challenging question. Any test designer must make sure 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 of an order of larger magnitude in the case of larger programs. Automated GUI testing has been facing this very problem. To overcome this problem, D.J. Kasik and H.G. George had introduced an interesting method of generating GUI test cases [6]. This method employs the theory that good GUI test coverage can be attained by simulating a novice user [6]. From their theory, it can be hypothesized that a skilled user of a system would go after a very direct and conventional path all the way through a GUI whereas a beginner user would follow a relatively random path [6].

Genetic Algorithm can be used to find out optimized test suite for GUI testing. Genetic algorithms work on the basis of 'genes', which are created randomly. They are then subjected to some task. Genes demonstrating good performance are kept for next phases, while others are discarded. As far as testing is concerned, 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 many of the automatically generated optimization parameters satisfy the given test criterion. Genes that optimize the coverage function will survive while others will be discarded; the process is repeated with optimized genes being replicated and further random genes replacing of discarded genes. Ultimately one gene (or a small group of genes) will be left in the set and this would logically be the best fit for coverage function.

The salient features of our proposed technique are as follows:

- A fully automated test coverage analysis;
- GA has been used for the optimization of test coverage;
- Three different simple applications were selected to experiment with;
- The results of the experiments are very encouraging.

The remainder of the paper is organized as follows: In the next Section, we discuss related work in the fields of software testing, GUI testing and optimization techniques. Section 3 describes the proposed method in detail. Section 4 presents the results of the experiments related to the prioritization of test cases and maximization of coverage, in section 5; some future directions have been presented. Section 6 concludes the paper.

II. 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. But 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. But in the past few years, efforts have been made for developing different models for GUI testing. Atif 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. Atif M. memon and Xie also created an event-interaction graph (EIG) [3, 4]. Kasik and George [11] have an innovative idea to resemble novice GUI users. White L, Almezen H, Alzeidi N. has developed a technique to address the User-based testing of GUI sequences and their interaction [12]. White L and Almezen H. have also given techniques for generating test cases for GUI responsibilities using complete interaction sequences [13]. In [9, 10, 14, 15] Memon A.M. 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 [17,18]). In [14, 15], Atif M. Memon has used goal-directed search for GUI test case generation while in [16], author have used function composition for automated test-oracle creation. Atif M. Memon also used metrics from graph theory to define test coverage criteria for GUIs [19], graph-traversal to obtain smoke test cases for GUIs that are used to stabilize daily software builds [9, 20], and graph matching algorithms to repair previously unusable GUI test cases for regression testing [18].

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 [21-24]. The major inspiration for using these models is that a test designer simulates a GUI's behaviour 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 [21-24]. The state machine's abstract states may be used to verify the GUI's concrete state during test case execution [21-24]. Shehady and Siewiorek [25] 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 [25]. 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 sequences(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 the responsibilities and the subsequent CIS is relatively straightforward; the large manual effort is in designing the FSM model for testing, especially when the code is not available [12, 13].

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 [26, 27]. This technique has shown good results with **a** number of small programs. Pargas et al used a GA based on the control dependence graph to search for test data that gives good coverage [29]. They used the original test suite developed for the SUT as the seed for the GA. They compared their system to random testing on six small C programs. For the smaller programs, there was no difference, but for the three larger programs, the GA-based method outperforms random testing. Tracey et al presented a framework for test-data generation based on optimization algorithms for structural testing [30]. Tracey also 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 [30]. Yongzhong Lu et al presented a new GUI automation test model based on the event-flow graph modeling [31]. In this model, the 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 [31].

III. PROPOSED METHOD

A GUI is a hierarchical, graphical front-end to a software system that accepts user-generated and system-generated events as input 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.

In order to test GUI and analyse 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

The Working of genetic algorithm has been explained with the help of a block diagram in figure 01.

A. Test Data Generation

For test data generation, we have used event based techniques. For this purpose we have selected Notepad as a GUI product for testing. We have decomposed GUI into hierarchy modal that consists of nodes which represent objects; for instance like file is an object that has 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 specific path; like firstly click file; from the list of options we then choose the PRINT option. Thus to print a document we must follow the sequence of file to print. In this way, a hierarchy has been designed that represents the sequences of paths between different objects.

Figure 2 shows the path generation for Notepad on the basis of possible sequences of events.

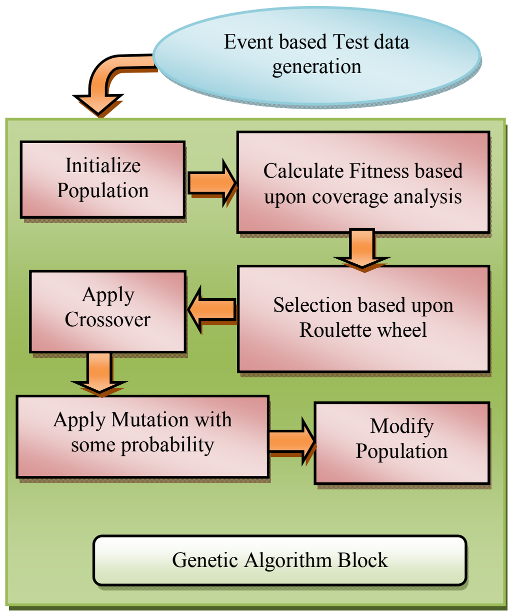
Event based Test data generation
Initialize Population
Calculate Fitness based upon coverage analysis
Selection based upon Roulette wheel
Apply Crossover
Apply Mutation with some probability
Modify Population
Genetic Algorithm Block

Figure 1. Block diagram of Genetic Algorithm

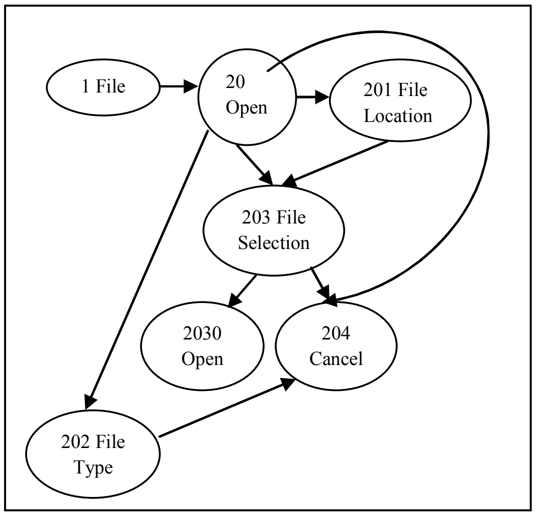

Figure 2. Path Generation for OPEN in Notepad

TABLE I. PATH GENERATION FOR NOTEPAD

10 New	20 Open	30 Save	40 Print
101 RT Document	201 Location	301 File location	401 Select Printer
104 OK	203 File Selections	302 File Name	402 Preferences
105 Cancel	204 Cancel	303 File Type	403 Find Printer
102 Text	202 File Type	304 Save	404 Page Range
104 OK	203 File Selection	305 Cancel	405 Number of Copies
105 Cancel	204 Cancel	302 File Name	406 Print
103 Unicode	203 File Selection	303 File Type	407 Cancel
104 OK	2030 Open	304 Save	408 Apply
105 Cancel	204 Cancel	305 Cancel	
104 Ok	204 Cancel		
105 Cancel			

B. *Optimization of Test Paths Using Genetic Algorithms*

Genetic algorithms are inspired by Darwin's theory about evolution. Solution of a problem being solved by genetic algorithms is evolving one. Algorithm is started with a set of solutions (represented by chromosomes) called a 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 older 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 would be reproduced. This is repeated until a pre-defined condition; for example, number of populations or improvement of the best solution is satisfied.

The Idea of evolutionary computing was introduced in the 1960s by I. Rechenberg in his work "Evolution strategies". His idea was then developed by other researchers. Genetic Algorithms (GA's) were introduced by John Holland. John Holland along with his students and (his) colleagues had developed GA. This lead to Holland's book "Adaption in Natural and Artificial Systems" published in 1975[32].

Following are steps of GA:

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

Example:

2	4	3	1	2

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 many paths have been covered by a chromosome?).
3. **[New population]** Create a new population by repeating the following steps until the new population is complete
   I. [Selection] Select two parent chromosomes from a population according to their fitness (the better fitness, greater the chance of being selected)
   II. [Crossover] With a certain crossover probability, crossover the parents to form a new offspring (children). If no crossover was performed, the offspring is an exact copy of parents.
   III. [Mutation] With a mutation probability, mutate the new offspring at each locus (position in chromosome).
   IV. [Accepting] Place the new offspring in a new population
4. **[Replace]** Use the newly 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
6. **[Loop]** Go to step 2

C. *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 as a termination criterion for the search. Our fitness function is:

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

Let us explain the working of genetic algorithm with the aid of an example.

Table 2 represents the available test paths and the lengths of these test paths, our aim is to check how many paths are being covered by our chosen chromosomes. This will let us know about the fitness function of each of the chosen chromosome.

TABLE II. TEST PATHS WITH LENGTH

No	Test Path	Length
1	1,2	2
2	1,9,2	3
3	1,8,2	3
4	1,8,4,2	4
5	1,8,3,4,2	5

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

1	8	7	4	2	9	6	8

Fitness function for the above chromosome can be calculated as follows = 2/5 = 0.4

D. *Reproduction Operators*

There are two reproduction operators available in the genetic algorithm: Crossover and Mutation. Crossover has two different types, one point cross over and two points cross over. We will apply these reproduction operators to increase the coverage efficiency.

To demonstrate the crossover operation, let's take the second chromosome;

6	3	2	7	2	1	9	2

Its fitness function = 1/5 = 0.2

Next, we generate a random number to find the cross over point.

Let's say, Rand = 5

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

1	8	7	4	2	1	9	2

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

E. *Mutation*

Now we take the child chromosome for mutation.

1	8	7	4	2	1	9	2

Again we will generate a random number for this purpose.
For position, Rand = 3

1	8	3	4	2	1	9	2

Fitness function = 5/5 = 1

IV. EXPERIMENTAL RESULTS

We had designed and developed the proposed application in MATLAB. The application has undergone extensive experimentation in order to determine its effectiveness. While three built-in sample applications were selected to experiment with**,** this included Notepad, WordPad and MS WORD. These applications were selected while, mindful of the following criteria:

- Ease of use: Applications have universal applicability with extensive GUI, which provides us with the ideal environment required to execute and monitor the effectiveness of our technique
- 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
- Highly structured applications: As the applications are highly structured, this provides us with an opportunity to design chromosomes for various tests quite efficiently.

The test data was generated manually by clicking on various GUI elements and tailoring the course of click-events. 36 to 50 test cases were generated per application. Each test case was of variable length, depending upon the sequence of events involved in performing that specific action.

Coverage analysis has shown that the 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.

The results have also shown that an increase in the number of generations resulted in enhanced coverage. In order to determine the optimal number of generations, we experimented with our technique using generations between 300 and 500(inclusive). The effect of enhancing the number of generations is shown in table 3.

TABLE III. COVERAGE ACCORDING TO NUMBER OF GENERATIONS

Number of Generations	**Coverage achieved**
300	65%
325	68%
350	69%
375	72%
400	73%
425	79%
450	85%
475	85%
500	85%

The results have shown the overall effectiveness and improvement that our proposed technique has achieved in effective coverage analysis.

V. 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. Not only that, we also plan to use other artificially intelligent optimization techniques so as to further enhance its effectiveness.

VI. CONCLUSION

Graphical User Interface testing has always been considered as a critical element of 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. Three simple applications were selected to experiment with, which included Notepad, WordPad, and MS WORD. The experiments have shown encouraging results. The results have also shown enhanced coverage with an increase in the number of generations.

ACKNOWLEDGMENT

The first author has been sponsored by Higher Education Commission of Pakistan, Government of Pakistan, and wishes to acknowledge HEC.

REFERENCES

[1] Bertolino "Software Testing Forever: Old and New processes and techniques for Validating Today's Applications", Keynote at 9th International Conference Product-Focused Software process Improvement (PROFES 2008), Monte Porzio Catone, June 2008, LNCS 5089 , 2008.

[2] Limerick, Ireland "Testing: a roadmap, International Conference on Software Engineering, Proceedings of the Conference on The Future of Software Engineering, Pages: 61 - 72, Year of Publication: 2000

[3] Memon, A. M. 2007. An event-flow model of GUI-based applications for testing: Research Articles. Softw. Test. Verif. Reliab. 17, 3 (Sep. 2007), 137-157. DOI= http://dx.doi.org/10.1002/stvr.v17:3

[4] Qing Xie and Atif M Memon. Using a pilot study to derive a GUI model for automated testing ACM Transactions on Software Engineering and Methodology Volume 18 , Issue 2 (November 2008) Article No. 7 Year of Publication: 2008 ISSN:1049-331X

[5] Korel, B.. Automated Software Test Data Generation. IEEE Transactions on Software Engineering (16)8: 870-879, August, 1990.

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

[7] Vaysburg, B., Tahat, L. H., and Korel. B., "Dependence Analysis in Reduction of Requirement Based Test Suites", In Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'02), pp, 107-111, 2002.

[8] MEMON, A. M. 2001. A comprehensive framework for testing graphical user interfaces. Ph.D. dissertation. Department of Computer Science, University of Pittsburgh, Pittsburgh, PA.

[9] Memon AM, Xie Q. Studying the fault-detection effectiveness of GUI test cases for rapidly evolving software. *IEEE Transactions on Software Engineering* 2005; 31(10):884–896.

[10] Atif M. Memon , Mary Lou Soffa , Martha E. Pollack, Coverage criteria for GUI testing, Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering, September 10-14, 2001, Vienna, Austria

[11] David J. Kasik , Harry G. George, Toward automatic generation of novice user test scripts, Proceedings of the SIGCHI conference on Human factors in computing systems: common ground, p.244-251, April 13-18, 1996, Vancouver, British Columbia, Canada [doi>10.1145/238386.238519]

[12] White L, Almezen H, Alzeidi N. User-based testing of GUI sequences and their interaction. Proceedings of the International Symposium on Software Reliability Engineering, 8–11 November 2001. IEEE Computer Society Press: Piscataway, NJ, 2001; 54–63.

[13] White L, Almezen H. Generating test cases for GUI responsibilities using complete interaction sequences. Proceedings of the International Symposium on Software Reliability Engineering, 8–11 October 2000. IEEE Computer Society Press: Piscataway, NJ, 2000; 110–121.

[14] Memon AM, Pollack ME, Soffa ML. Using a goal-driven approach to generate test cases for GUIs. Proceedings of the 21st International Conference on Software Engineering, May 1999. ACM Press: New York, 1999; 257–266.

[15] Memon AM, Pollack ME, Soffa ML. Hierarchical GUI test-case generation using automated planning. IEEE Transactions on Software Engineering 2001; 27(2):144–155.

[16] Memon AM, Pollack ME, SoffaML. Automated test oracles for GUIs. Proceedings of the ACMSIGSOFT 8th International mposium on the Foundations of Software Engineering (FSE-8), 8–10 November 2000. ACM Press: New York, 2000; 30–39.

[17] White L. Regression testing of GUI event interactions. Proceedings of the International Conference on Software Maintenance, 4–8 November 1996. IEEE Computer Society Press: Piscataway, NJ, 1996; 350–358.

[18] Memon AM, Soffa ML. Regression testing of GUIs. Proceedings of the 9th European Software Engineering Conference (ESEC) and 11th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE-11), September 2003. ACM Press: New York, 2003; 118–127.

[19] Memon AM, Soffa ML, Pollack ME. Coverage criteria for GUI testing. Proceedings of the 8th European Software Engineering Conference (ESEC) and 9th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE-9), September 2001. ACM Press: New York, 2001; 256–267.

[20] Memon A, Nagarajan A, Xie Q. Automating regression testing for evolving GUI software. Journal of Software Maintenance and Evolution: Research and Practice 2005; 17(1):27–64.

[21] Clarke JM. Automated test generation from a behavioural model. Proceedings of Pacific Northwest Software Quality Conference, May 1998. PNSQC: Portland, OR, 1998.

[22] Chow TS. Testing software design modelled by finite-state machines. IEEE Transactions on Software Engineering 1978; 4(3):178–187.

[23] Esmelioglu S, Apfelbaum L. Automated test generation, execution, and reporting. Proceedings of Pacific Northwest, Software Quality Conference, October 1997. IEEE Press: Piscataway, NJ, 1997.

[24] Bernhard PJ. A reduced test suite for protocol conformance testing. ACM Transactions on Software Engineering and Methodology 1994; 3(3):201–220.

[25] Shehady RK, Siewiorek DP. A method to automate user interface testing using variable finite state machines. Proceedings of the 27th Annual International Symposium on Fault-Tolerant Computing (FTCS'97), June 1997. IEEE Computer Society Press: Piscataway, NJ, 1997; 80–88.

[26] B.F. Jones, D.E. Eyres, H.H. Sthamer, A strategy for using Genetic Algorithms to automate branch and fault-based testing, The Computer Journal 41(2) (1998) 98-107.

[27] B.F. Jones, H.H. Sthamer, D.E. Eyers, Automatic structural testing using genetic algorithms, The Software Engineering Journal 11(5) (1996) 299-306.

[28] C. Michael, G. McGraw, M. Schatz, Generating software test data by evolution, IEEE Transactions on Software Engineering 27(12) (2001) 1085-1110

[29] R. Pargas, M. J. Harrold, and R. Peck. Test-data generation using genetic algorithms Journal of Software Testing, Verification and Reliability, 9(4):263–282, 1999.

[30] N. Tracey, J. Clark, K. Mander, J. McDermid, Automated test-data generation for exception conditions, Software Practice and Experience, 30(1) (2000) 61-79

[31] Yongzhong Lu Danping Yan Songlin Nie Chun Wang, Development of an Improved GUI Automation Test System Based on Event-Flow Graph. International Conference on Computer Science and Software Engineering, Date: 12-14 Dec. 2008

[32] John H. Holland, Adaptation in Natural and Artificial Systems, MIT Press April 1992.