EvoGUITest – A Graphical User Interface Testing Framework based on Evolutionary Algorithms

Gentiana Ioana Laţiu, Octavian Creţ and Lucia Văcariu
Technical University of Cluj-Napoca, Computer Science Department, 26-28 Bariţiu Street, Cluj-Napoca, Romania

Keywords: Graphical User Interface Testing, Evolutionary Algorithms, Testing Framework.

Abstract: Software testing has become an important phase in software applications' lifecycle. Graphical User Interface (GUI) components can be found in a large number of desktops and web applications and also in a wide variety of systems like mobile phones. In the last years GUIs have become more and more complex and interactive. The GUI testing process requires interaction with the GUI components, mainly by generating mouse and keyboard events. Given their increased importance, GUIs verification for correctness can contribute to the establishment of the correct functionality of the corresponding software application. Most of the current GUI testing methodologies are ad hoc and manual, therefore they are resource consuming. This paper presents EvoGUITest, a novel GUI testing framework based on evolutionary algorithms which tests the GUI independently from the application code itself. EvoGUITest framework is designed for testing GUIs of web applications.

1 INTRODUCTION

GUI is a specification for the look and feel of the software application (Bernard, 1998). GUI consists of graphical elements such as windows, icons, menus, buttons, testboxes. A well designed GUI is very intuitive and easy to be used by the users. The GUI components can be a crucial point in the users' decisions to either use or not use that specific software application (Pimenta, 2006).

While GUIs have become ubiquitous and increasingly complex, their testing remains largely ad-hoc. Due to his complexity, the testing process is problematic and time-consuming (Ganov et al., 2008).

During manual GUI testing procss, each test case needs a long time to execute (tens of seconds, for a medium complexity GUI). The manual checking process of the result needs another time spent by the human tester, which is also of a few tens of seconds. If for instance there is a suite of 10,000 test cases to be applied, then the total testing time becomes enormous (hundreds of hours) (Yang, 2011).

If the test cases are executed automatically, it takes around 3 seconds for each test case to be executed, and another 1 second for checking the output results. 10,000 test cases need around 10 hours to be executed, which shows an acceleration of one order of magnitude compared to the manual testing process (Yang, 2011) – that is why the research mainly focuses on automated GUI testing.

Some years ago, test cases were generated randomly during the automatic GUI testing process. Because the coverage of random input testing was very weak, the scientific community started studying the usage of the Evolutionary Algortihms (EA) for automating the GUI testing process.

In the last years the Evolutionary Art started to be used in a lot of applications, with interactive evolutionary algorithms in which user assigns scores to images based on their suitability (Bergen and Ross, 2011). EvoSpace framework is used for development of interactive algorithms for artistic design (Valdez et al., 2013).

The rest of this paper is organized as follows: Section 2 describes the automatic process for GUI testing, Section 3 describes in details the EA process, Section 4 describes our novel proposed GUI testing framework (EvoGUITest). Inside this Section the framework architecture and the experimental results are also presented. Section 5 concludes the paper, sumarizing the future work planned.

Ioana Latiu G., Creţ O. and Văcariu L..
EvoGUITest – A Graphical User Interface Testing Framework based on Evolutionary Algorithms.
DOI: 10.5220/0004518200750082
In *Proceedings of the 5th International Joint Conference on Computational Intelligence* (ECTA-2013), pages 75-82
ISBN: 978-989-8565-77-8

2 AUTOMATIC GUI TESTING

The GUI testing is a process which aims at testing the software application's user interface and detecting if the GUI is functionally correct. GUI testing includes checking how the software application handles mouse and keyboard events (Prabhu and Malmurugan, 2011).

The automatic GUI testing process includes automatic manual testing tasks performed by human testers. By the automatic testing process, a software program executes the testing tasks and analyzes if the GUI under test is functionally correct.

Automatic GUI testing can be executed using different techniques.

2.1 Capture/Replay Tools

These tools have two modes of functioning: capture and replay. In capture (record) mode, the tool is able to record testers' actions while they are interacting with the GUI. The set of actions are recorded inside test scripts. These tools provide a scripting language which can be used by engineers for maintaining the test scripts.

In *replay mode*, the recorded test scripts are executed. During execution of each test script, some mouse or keyboard events are executed on the GUI. The test scripts' execution process is automatic and can be repeated several times.

The most important disadvantage of these GUI testing tools is the lack of structure of the test scripts, which makes the maintenance process difficult. These tools don't provide any support to design and evaluate test cases based on coverage criteria.

Three examples for these tools are: Selenium (http://seleniumhq.org), WinRunner (http://mercury.com) and Rational Robot (http://www-01.ibm.com/software/awdtools/ tester/robot/)

2.2 Random Input Testing

This testing technique is also referred in the literature as stochastic testing or monkeys testing (Nyman, 2006). Random input testing refers to the idea that somebody seats in front of a software application and interacts randomly with it, by sending keyboard and mouse events.

The goal of monkeys testing is to crash the GUI of the software application under test. They generate tests cases randomly without knowing anything about the software application. The biggest problem of this testing technique is that monkeys cannot recognize software errors. There is a smarter category of monkeys called "smart monkeys" which have some knowledge about the software application under test. These monkeys can find more bugs, but they are more expensive to be developed (Pimenta, 2006).

Even if random input testing tools have a weak coverage, one of the biggest software companies has reported that 10-20% of the bugs in their software applications were found by using random input testing method (Nyman, 2006).

2.3 Unit Testing Frameworks

Unit testing technique for GUI testing requires programming the test cases. Unit testing frameworks like NUnit (http://nunit.org) can be used for executing GUI test cases.

These tools are helpful in case many bugs can only be discovered through a particular sequence of actions. With these tools the tester has to write code to simulate user interaction with the GUI under test. After executing the test cases the tester should check if the result obtained is the one expected.

In order to be effective, the GUI testing process using unit testing frameworks needs a lot of programming effort. There are some GUI libraries such as Abbot (http://abbot.sourceforge.net) which provide methods to simulate user interaction.

2.4 Model-based Testing

Model-based testing requires that GUI states and events are described with a certain type of model. Having these models in place, the test cases can be generated automatically, either randomly or according to some particular coverage criteria.

The model-based testing process is presented in Figure 1.

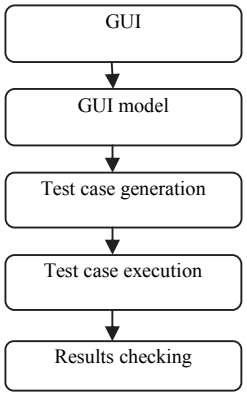

Figure 1: Model based testing.

The model based testing process starts with the construction of the GUI model. The model is used to generate test cases which are then executed over the GUI. In the last step, the obtained results are compared to the expected results described in the model.

The most important existing testing models used for model based testing are the following ones (Yang, 2011):

- *Event Sequence Graph* (ESG) – a directed graph which contains a finite set of nodes and a finite set of edges. Each node represents a GUI event and the sequence of nodes represents the sequence of GUI events. This model was proposed by Belli et al. (Belli, 2001).
- *Event Flow Graph* (EFG) and *Event interaction graph* (EIG) – inside EFG, each node represents a GUI event and all events which can be executed immediately after one event are directly linked with directed edges from this event. A path inside the EFG represents a sequence of GUI events and can be considered a test case. EIG is the later version of EFG. The structure of EIG is composed by all GUI events which represent the GUI nodes and all relations between events which represent the graph edges.

The model-based testing technique is usually used to test the structural representation of a GUI (Qureshi and Nadeem, 2013).

EvoGUITest framework uses in the beginning of testing process a random input testing method for generating the first set of test cases. Then the test cases evolve using the EA process. The aim of EvoGUITest framework is to find out the longest sequence of events which tests as much as possible GUI controls.

3 EVOLUTIONARY ALGORITHMS

EAs are software programs that attempt to solve complex problems by mimicking the processes of Darwinian evolution (Jones, 1990). They operate on a population of possible solutions by applying the principle of survival of the fittest to produce better approximations to a solution (Pohlheim, 2006).

During the EA process a big number of artificial individuals search the solution over the space of the problem.

The artificial individuals are usually represented by vectors of binary values. Each individual encodes a possible solution for the problem which needs to be solved.

The most widely known EA is the Genetic Algorithm (GA). In the following, the GA and the Simulated Annealing (SA) algorithms will be presented. These two algorithms were used for generating test cases inside the EvoGUITest application.

3.1 Genetic Algorithms

GA originated from the work of John Holland. They are the most obvious mapping of natural evolutionary process into a software application (Streichert, 2007).

The GA process begins with *a set of candidate solutions* which is called *population*. A population is composed of individuals who are constituted from one or more genes. A population's individuals are used to form a new population by using *crossover* and *mutation* operators. During the GA process there is an expectation that the newly generated individuals are better than their parents.

GAs are well known and widely used in scientific and technical research because of their parallel nature, of their design space exploration and also due to their ability to solve non-linear problems (Rauf, 2010).

A GA has four important phases:

- *Evaluation* – during this phase each individual is evaluated by the evaluation method. The *fitness function* is used for evaluation. It calculates how good the individual is to satisfy the test criteria.
- *Selection* – during this phase individuals are chosen randomly from the current population for creating new individuals in the next generation. The main idea of the selection methods is that fittest individual has the biggest probability of survival; therefore he has a greater probability to be picked for reproduction.
- *Crossover* – during this phase, recombination reproduces the chosen individuals and pair wise information will be exchanged and will result in a new population (Rauf, 2010). The crossover process joins two selected individuals at a crossover point, thus producing two new offsprings. During crossover, the first parent's right half genes are exchanged with the subsequent right half of the second parent. After crossover is performed, each parent pair will result in two offsprings. Crossover is the operator which is responsible for improving the individuals.
- *Mutation* – during this phase a randomly chosen bit is changed from '0' to '1' or from '1' to '0'. Each bit inside an individual has the same probability to mutate. Mutation is the operator

which is responsible for introducing variety inside the population.

3.2 Simulated Annealing

SA is a probabilistic method for finding the global minimum of a cost function that may possess several local minima (Bertsimas and Tsitsiklis, 1993). This algorithm emulates the physical process whereby a solid is slowly cooled until its structure becomes frozen. This happens at a minimum energy configuration.

The SA algorithm has four basic elements (Rutenbar, 1989):

- *Configurations* – these represent the possible problem solutions over which the process will search for the problem solution.
- *Move Set* – this set represents the computations performed to move from one configuration to another, as annealing proceeds.
- *Cost Function* – measures how "good" a particular configuration is?
- *Cooling Schedule* – anneal the problem from a randomly generated possible solution to a good solution. Usually the schedule needs a starting hot temperature and different rules for establishing when the current temperature should be decreased, by which amount temperature should be lowered and when the process should take end.

The most important feature of the SA algorithm is that it is a probabilistic method where during the search process the moves that increase the cost function are accepted in addition to moves which decrease the cost function (Nascimento et al., 1999). This feature is the central point of the algorithm which enables the search process to locate the global minimum among all the other local minima.

The most important challenge in improving the performance of the SA algorithm is to decrease the temperature and in the same time to ensure that the process does not stop in a local minimum.

The goal of the SA algorithm is to find the quickest annealing schedule that achieves a value for finding the global minimum equal to unity (Nascimento et al., 1999).

The SA algorithm is suitable for solving large scale optimization problems inside which the global minimum is located among many local minima values.

4 EvoGUITest

EvoGUITest is a GUI automated testing framework based on evolutionary algorithms. It automatically generates test cases which are used afterwards for testing the GUI. The test cases suite is generated automatically by an EA-based process. EvoGUITest's objective is to find the sequence of events which produces the biggest number of changes inside the GUI in a minimum amount of time.

4.1 The EvoGUITest Framework Architecture

The EvoGUITest application is a GUI testing framework which uses EAs for generating GUI test cases. It is developed in JavaScript and it runs on client side. Being developed in JavaScript it is very easy to be extended without any need of extra tools to write JavaScript. EvoGUITest is able to generate test cases for Web applications which have a GUI component already developed.

The testing process with this GUI testing framework consists of the following main steps:

- *Analysis* – the GUI state together with each GUI controls' states are analyzed. The result of this step is the list of HTML properties and events which correspond with each control located inside GUI.
- *Test cases generation* – generate test cases by using the specific EAs methods.
- *Test cases execution* – executes test cases.
- *Results verification* – verifies the results after the execution of the test cases.

Figure 2 presents the main components of the EvoGUITest framework.

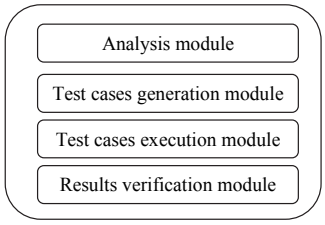

Figure 2: The EvoGUITest architecture.

The most important part of the framework is the module which generates test cases using EAs. Each test case is represented by an individual. The first population of individuals is randomly generated (Figure 3).

Each individual consists of an array of genes, each corresponding to a GUI control. In Figure 3 the array of genes for each individual corresponds to an array of ids which correspond to each GUI control. Each GUI control which appears inside an individual

id	genes
1	btn1,div11,btn2,div11,btn7,tf13
2	div1,btn10,btn2,i1,div3,btn10,btn7,tf31,p1
3	i2,header,p1,div8,tf13,btn5
4	div1,div11,btn5,tf21,div11,div10,div11,p3
5	tf21,tf31,btn6,span2,div7,tf13
6	tf36,tf31,span4,tf34,div2,span1,btn4,tf1,span3
7	span4,div9,div11,div9,tf31
8	div9,div3,tf38,div8,i1,tf18,i1,tf18
9	div8,btn8,div10,tf21,btn4,btn4,tf34,btn1
10	div6,div5,div7,tf13,btn9,tf19,i1,btn3,p2

Figure 3: Randomly generated individuals for testing GUI of Calculator application.

is linked with a user action on the GUI. After the first population of individuals is generated, the individuals are evolved by means of the EA process. After each generation, the new individuals are displayed together with their objective, age and fitness function. Figure 4 shows the individuals from the first generation. The population of individuals is generated for testing the GUI of a complex application.

id	genes	obj	age	fitness
1	btn4,div2,div4,tf38,btn2,btn10,btn2,tf31,tf19,div3	0.011364	1	0.0506
2	div6,tf18,btn10,btn9,span2,tf19,span2,btn7	0.012346	1	0.0494
3	btn6,tf34,tf31,btn1,btn8,btn7,span3	0.014706	1	0.0481
4	tf36,tf21,btn1,btn7,div10,btn6	0.014706	1	0.0469
5	div4,span3,btn6,div3,tf13	0.015385	1	0.0456
6	btn4,tf38,btn7,tf38,span3,i2,btn9,div11,tf18	0.015873	1	0.0444
7	div1,divhdn,div6,btn10,tf31,div11,i2,span1,i2	0.016667	1	0.0431
8	div7,div1,btn8,tf36,btn6,tf31	0.016667	1	0.0419
9	btn3,div9,btn4,span2,tf34,btn7,i1,span1,divhdn	0.016667	1	0.0406
10	div9,tf38,btn2,btn10,tf36	0.018868	1	0.0394
11	tf1,span3,btn7,tf18,tf21,btn10,header,span1	0.019608	1	0.0381
12	tf1,btn3,tf21,span5,btn9,div4	0.02	1	0.0369
13	div10,btn9,btn3,tf21,p1,tf34,btn7,btn3,btn9	0.020833	1	0.0356
14	btn10,p3,div3,div10,div6,div3	0.020833	1	0.0344
15	div8,btn3,span1,span3,p2,btn4,p3,tf19	0.022222	1	0.0331
16	span5,btn9,p1,div5,btn3	0.022222	1	0.0319
17	div4,btn9,div2,div7,btn2,span1,div3,div9,div1	0.025	1	0.0306
18	btn7,span4,tf36,div1,span2,btn2,btn4,div3,span3,tf34	0.026316	1	0.0294
19	btn1,tf13,div7,div5,div11,btn9,span4	0.028571	1	0.0281
20	tf31,div8,p1,tf31,tf18,btn4,p3,div4	0.028571	1	0.0269

Figure 4: First generation of individuals for testing complex GUI component.

The individuals are classified so that the first one is the best individual from the current generation. As it can be easily observed, the first individual is the one which contains more button controls; therefore it is the one which produces the biggest number of changes inside the GUI. Each individual has the age equal with 1, because they are individuals from the first generation. The age represents the current generation number. The objective column contains the objective value for each individual, and the fitness column contains the fitness value assigned to each individual. The objective attribute represents the performance of the individuals, while the fitness value represents rang of individuals inside the hierarchy.

For example, if we have the following objective values:

Individual 1: 2

Individual 2: 1000

Individual 3: 65536

if the roulette wheel selection will be applied on the above population of individuals the last individual won't have any chance to be selected for reproduction. If we assign a fitness function for each individual, who have the following values:

Individual 1: 2 Fitness: 0.5

Individual 2: 1000 Fitness: 0.3

Individual 3: 65536 Fitness: 0.2

than the last individual has a small chance to be selected for crossover.

The objective function which evaluates each individual is presented in formula (1.1):

$$Objective = (1 / no\_of\_changes) + 1/(100 \times no\_of\_similar\_states) + 1/(100 \times no\_of\_useless\_states) \quad (1.1)$$

Each individual should produce the greatest number of changes and the smallest number of similar states and useless actions. A useless action is an action which doesn't produce any change inside the GUI. A similar state is a state which has already appeared earlier inside the set of states produced by the same individual.

The EvoGUITest framework contains a separate section where the user can set values for the most important parameters used by the GA and SA algorithms. For each one of these two algorithms, the user can select the values for the parameters presented in Table 1. The variables that affect the outcome of the SA algorithm are: the initial temperature, the rate at which the temperature decreases (alpha) and the stopping condition of the algorithm (epsilon).

The number of individuals indicates how many individuals are there in each population and the number of generations represents the generations for which the GA algorithm will be performed. The number of genes represents the minimum and the maximum length of each individual from the first population. The number of selected pairs for crossover represents how many individuals will be selected for reproduction. The mutation probability represents the percentage value of applying the

Table 1: Parameters list for GA and SA algorithms.

GA	Values	SA	Values
Number of individuals	40	Initial temperature	100
Number of genes (min, max)	Min: 10 Max: 25	Epsilon	0.001
Number of selected pairs for crossover	20	Alpha	0.999
Mutation probability	0.2	-	
Mutation addition probability	0.5	-	
Mutation removal probability	0.5	-	
Number of generations	50	-	

mutation operator. Mutation can be applied in two ways: either by removing a gene from an individual or by adding a new gene.

Figure 5 displays the section which consists of the GA parameters list for the EvoGUITest application.

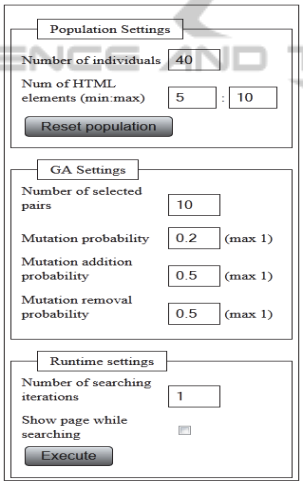

Figure 5: GA parameters settings area.

4.2 Experimental Results

All the experiments were performed on a computer having the following configuration: Intel I3 processor, 2.2 GHz, Windows 7 Operating System. Three GUIs were tested: the first one is a simple GUI which consists of two buttons and two textboxes, the second one is the GUI of the classic Calculator application from Windows and the last one is a complex GUI which consists of more than twenty user controls.

For test cases generation we used both the GA and the SA algorithms. The selection method used for GA algorithm was the roulette wheel method. For each specific parameter, for each algorithm, the values presented in Table 1 were used in order to generate the test cases. These values were chosen to be used for running EAs based on our empirical studies done before. All the EAs' specific parameters' values were setup after we have tried hundred of runs with different values for these parameters. The values for which we have obtained the best results were chosen.

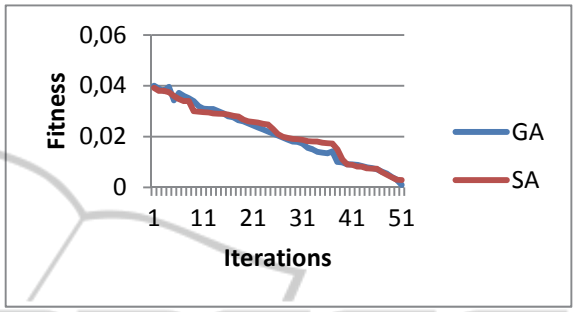

Figure 6: Test case generation for the simple GUI.

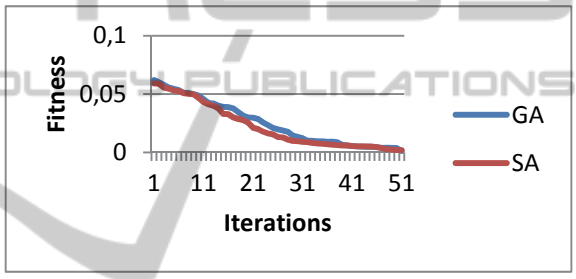

Figure 7: Test case generation for the Calculator GUI.

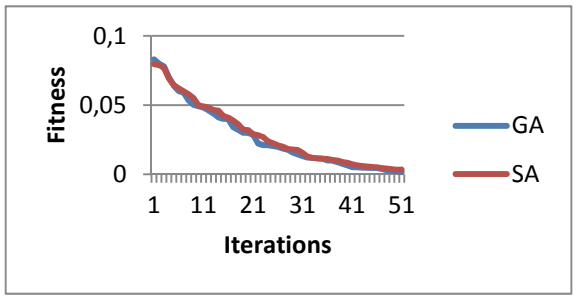

Figure 8: Test case generation for the complex GUI.

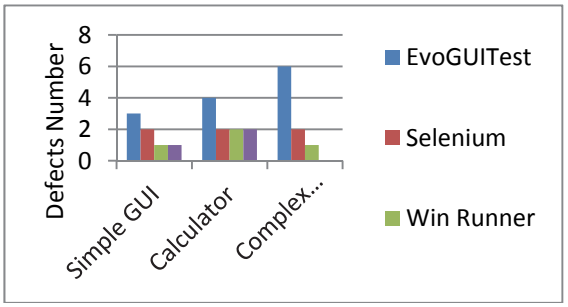

Figure 9: Number of defects discovered by different testing frameworks.

Figures 6, 7 and 8 present the test results obtained for each of the three GUIs using the GA and the SA algorithms for evolving the test cases suite. In Figure 9 there is presented a comparison between four test suites which are composed of ten test cases. The test suites were generated with EvoGUITest, Selenium, WinRunner and Rational Robot. They were used in regression testing phase for detecting errors inside Web application.

The performance of the best for both GA and SA is presented in Table 2.

Table 2: Best individuals' performance for the GA and SA algorithms.

GUI	GA Perfor-mance	No. of GUI Changes	SA Perfor-mance	No. of GUI changes
Best individual for simple GUI testing	0.001	14	0.0029	11
Best individual for calculator GUI testing	0.0012	19	0.0019	15
Best individual for complex GUI testing	0.0015	27	0.0034	23

From Figures 6, 7, 8 and Table 2 one can notice that the GA is able to find better test data in comparison with the SA algorithm. GA manages to find out the sequence of events which produces more changes inside GUI in comparison with SA. The individual which produces the biggest number of changes inside the GUI is the one which has the smallest value for fitness function, because the testing problem is transformed into a minimization problem. It shows that individuals have evolved from the first generation to the last one. The best individual from the last generation produces the biggest number of changes inside the GUI; therefore, it has the smallest value of the fitness function.

The mean value of convergence time (in seconds) obtained from ten runs of each algorithm is presented in Table 3.

Table 3: Convergence time(s) for the GA and SA algorithms.

GUI type	GA Convergence (s)	SA Convergence (s)
Simple GUI testing	30	46
Calculator GUI testing	40	57
Complex GUI testing	60	78

The convergence time for GA algorithm is smaller than the convergence time obtained for SA algorithm.

From Figure 9 can be noticed that the test suite generated using EvoGUITest is able to find more defects in comparison with the other test suites even if they have the same amount of test. This illustrates the fact that the test suite generated with EvoGUITest is better than the others test suites.

5 CONCLUSIONS AND FUTURE WORK

This paper presents EvoGUITest, an original framework for automatically testing graphical user interfaces of Web applications based on EAs techniques. The main features of the EvoGUITest framework are the following:

- It tests the GUI separately from the application source code itself.
- It automatically generates and executes the test suite.
- It is able to find the sequence of events which produces the biggest number of changes inside the GUI, so it verifies a biggest number of controls inside the GUI.

The EvoGUITest framework is original because it runs on client side, being developed in Javascript and it tests the GUI of the application separately from the software application itself. It is the first GUI testing application developed only using JavaScript. The advantage of using JavaScript is that it is platform-independent and it can test GUI components developed in any programming language. The extension of the framework is very easy because there is no need of any extra tools to write JavaScript code. This can be done using any plain text or HTML editor.

EvoGUITest has the objective to find out the most important sequence of events which produces the biggest number of changes inside the GUI. By producing the biggest number of changes, the sequence is able to verify as many components as possible inside the GUI.

EvoGUITest is able to find out the most important sequence of GUI events in about 50 iterations.

Future work will involve using EvoGUITest framework for testing larger projects. We will also focus on using EvoGUITest for regression testing. The test cases suite will be used to check if the GUI still functions correct after each development change is performed. The framework will be extended with other evolutionary algorithms: Particle Swarm Optimization (PSO) and Ant Colony Optimization (ACO) algorithms.

A complete automated testing framework based

on EAs could be designed and implemented, for completely automating the GUI testing process.

ACKNOWLEDGEMENTS

This work was supported by a grant of the Romanian National Authority for Scientific Research, CNDI-UEFISCDI, project number 47/2012.

REFERENCES

Belli F., 2001. Finite-State Testing and Analysis of Graphical User Interfaces, *International Symposium on Software Reliability Engineering*, China.

Bergen S., Ross J., 2011, Evolutionary art using summed multi-objective ranks, Genetic Programming Theory and Practice VIII, Springer Science.

Bernard J., 1998. The Graphical User Interface: An Introduction, *Seminal works in computer human interaction*, 30(3), 24-28.

Bertsimas D., Tsitsiklis J., 1993. Simulated Annealing, *Statistical Science*, vol. 8, no.1, 10-15.

Ganov S., Killmar C., Khurshid S., Perry D., 2008, Test Generation for Graphical *User Interfaces Based on Symbolic Execution*, AST.

Jones G., 1990. Genetic and Evolutionary Algorithms, University of Sheffield, CGA04.

Nascimento V., Carvalho V., Castilho C., Soares E., Bittencourt C., Woodruff D., 1999. The Simulated Annealing Global Search Algorithm Applied to the Crystallography of Surfaces by Leed, Surface Review and Letters, vol. 6, no. 5, 651-661.Nyman N., 2000. Using Monkey Test Tools, Software Testing and Quality Engineering Magazine.

NUnit Framework, http://nunit.org, online documentation. Abbot, http://abbot.sourceforge.net, online documentation

Nyman N., 2006. In Defense of Monkey Testing, Software Testing and Quality Engineering Magazine.

Qureshi I.A., Nadeem A., 2013. GUI Testing Techniques: A Survey, *International Journal of Future Computer and Communication,* vol. 2, no.2.

Pimenta A., 2006. Phd. Thesis, Automated Specification-Based Testing of Graphical User Interfaces, Department of Electrical and Computer Engineering, FEUP.

Pohlheim H., 2006. Evolutionary Algorithms: Overview, Methods and Operators.

Prabhu J., Malmurugan N., 2011. A Survey on Automated GUI Testing Procedures, European Journal of Scientific Research, no. 3, pp. 456-462.

Rational Robot Framework, http://www-01.ibm.com/software/awdtools/tester/robot/, online documentation.

Rauf A., 2010. Coverage Analysis for GUI Testing, Phd. Thesis, Department of Computer Science, National University of Computer and Emerging Sciences, Pakistan.

Rutenbar R., 1989. Simulated Annealing Algorithms: An Overview, IEEE Circuits and Devices Magazine.

Selenium Framework, http://seleniumhq.org, online documentation.

Streichert F., 2007. Evolutionary Algorithms in Multi-Modal and Multi-Objective Environments, Phd. Thesis, University of Tubingen, Germany.

Valdez-Garcia M. et al., 2013, EvoSpace-Interactive: A Framework to Develop Distributed Collaborative-Interactive Evolutionary Algorithms for Artistic Design, *Evolutionary and Biologically Inspired Music, Design, Sound Art and Design,* vol. 7834, pp. 121-132.

WinRunner Framework, http://mercury.com, online documentation.

Yang X., 2011. Phd. Thesis, *Graphic User Interface Modelling and Testing Automation*, Victoria University.