Automated Graphical User Interface Testing Framework—Evoguitest—Based on Evolutionary Algorithms

Gentiana Ioana Latiu, Octavian Augustin Cret and Lucia Vacariu

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 mobile devices. In the last years GUIs have become more and more complex and interactive, their testing process requiring an interaction with the GUI components, mainly by generating mouse, keyboard and touch events. Given their increased importance, GUIs verification for correctness contributes to the establishment of the correct functionality of the corresponding software application. The current research on GUI testing methodologies primarily focuses on automated testing. This paper presents EvoGUITest, a novel automated GUI testing framework based on evolutionary algorithms which tests the GUI independently from the application code itself. The framework is designed for testing GUIs of web applications. Results have been compared, based on specific metrics, with others existing frameworks.

Keywords Graphical user interface testing · Evolutionary algorithms · Automated testing framework

1 Introduction

GUI is a specification for the look and feel of the software application [1]. A GUI consists of graphical elements such as windows, icons, menus, buttons, textboxes. A well designed GUI must be intuitive and user friendly, being the image of the

G.I. Latiu (✉) · O.A. Cret · L. Vacariu
Computer Science Department, Technical University of Cluj-Napoca,
26-28 Baritiu Street, Cluj-napoca, Romania
e-mail: Gentiana.Latiu@cs.utcluj.ro

O.A. Cret
e-mail: Octavian.Cret@cs.utcluj.ro

L. Vacariu
e-mail: Lucia.Vacariu@cs.utcluj.ro

© Springer International Publishing Switzerland 2016

K. Madani et al. (eds.), *Computational Intelligence*,
Studies in Computational Intelligence 613,
DOI 10.1007/978-3-319-23392-5_3

application. A good quality of the GUI is necessary and the diminishing of the testing cost becomes an important requirement. The GUI's set of components can be a crucial point in the users' decisions to either use or not use that specific software application [2].

While GUIs have become ubiquitous and increasingly complex, their testing remains largely ad-hoc. Due to its complexity, the testing process is problematic and time-consuming [3].

During the manual GUI testing process, each test case needs a long time to execute (tens of seconds, for a medium complexity GUI). The manual checking process of the results 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) [4].

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 [4]—that is why the research mainly focuses on automatic GUI testing.

Different frameworks were built to automate the testing process for Web applications GUIs, to eliminate the human tester involvement, etc., but many of these were made either for some particulary GUI software systems, or for the systems at a very general level.

A survey by Al-Zain et al. on automation testing tools for web applications shows, using different criteria (the effectiveness of recorder/playback tools, handling of page waits, cross browser compatibility, technical support, and the number of different techniques available to programmatically locate elements on web pages), that free and simple tools can be more powerful and time saving, compared to commercially sophisticated and expensive tools [5]. The authors have also summarized the best practices and guidelines to be considered when adopting automated GUI functional tests for web applications [5].

A comparative study of automated testing tools was conducted in [6]. Based on criteria such as the efforts involved with generating test scripts, capability to playback the scripts, result reports, speed and cost, Mercury QuickTest Professional and the AutomatedQA TestComplete have been compared. Analyzing the features supported by these two functional testing tools, which help minimizing the resources in script maintenance and increasing the efficiency of script reuse, the authors conclude that both tools are good, but for data security needs the QuickTest Professional is better.

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.

To only mention some of the most spectacular applications of EA in real life, we could say that in the last years the Evolutionary Art was used in a lot of applications, with interactive EAs in which the user assigns scores to images based on their suitability [7]; also, the EvoSpace framework is used for developing interactive algorithms for artistic design [8].

The rest of this paper is organized as follows: Sect. 2 describes the automatic process for GUI testing, Sect. 3 provides a detailed description of the EA process, Sect. 4 describes our novel proposed Web GUI testing framework (EvoGUITest). In this Section the framework architecture and the experimental results are also presented. Section 5 concludes the paper, summarizing the future work planned.

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 the way the software application handles mouse and keyboard events [9].

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 is 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 the 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 [10], WinRunner [11] and Rational Robot [12].

2.2 Random Input Testing

This testing technique is also referred in the literature as stochastic testing or monkeys testing [13]. 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 [2].

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 [13].

2.3 Unit Testing Frameworks

Unit testing technique for GUI testing requires programming the test cases. Unit testing frameworks like NUnit [14] 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 [15] 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 Fig. 1.

The model based testing process starts with the construction of the GUI's 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 [4]:

- *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 [16].
- *Event Flow Graph* (EFG) and *Event Interaction Graph* (EIG)—inside the 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

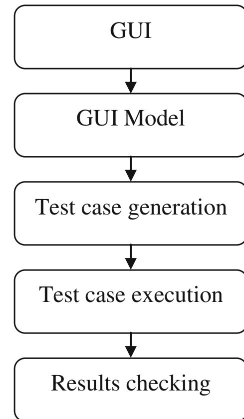

Fig. 1 Model based testing

the EFG represents a sequence of GUI events and can be considered a test case. EIG is the later version of the EFG. The EIG's structure is composed by all the GUI events which represent the GUI nodes and all relationships between events which represent the graph edges.

The model-based testing technique is usually used to test the structural representation of a GUI [17].

Some of the frameworks used in the testing process of the GUIs of Web applications are WebGuitar [18], Artemis [19], Atusa [20] and Kudzu [21]. Web Guitar uses the ESG model; Atusa and Kudzu use the EIG model, all being based on functional testing. Artemis has a grey-box testing style, both structural and functional.

The EvoGUITest framework that was developed by our team 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 an evolutionary process. The aim of the EvoGUITest framework is to determine the longest sequence of events which tests as many GUI controls as possible. The EvoGUITest framework will be further detailed in Sect. 4.

3 Evolutionary Algorithms

EAs are software programs that attempt to solve complex problems by mimicking the processes of Darwinian evolution [22]. They operate on a population of possible solutions by applying the principle called *survival of the fittest* to produce better approximations to a solution [23].

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

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, both Genetic Algorithm and the Simulated Annealing (SA) algorithm 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 [24].

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 [25].

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 [25]. The crossover process joins two selected individuals at a crossover point, thus producing two new offsprings. During crossover, for instance the first parent's right half genes can be 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 [26]. 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 [27]:

- *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 [28]. 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 [28].

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 novel GUI automatic 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. A bigger number of changes inside the GUI guarantee a better coverage of the search space, i.e. capturing a greater number of situations for testing the GUI's functionality.

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.

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 Fig. 3 shows such an initial population for the classical Calculator application running under Windows.

Each individual consists of an array of genes, each corresponding to a GUI control. In Fig. 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 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. 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

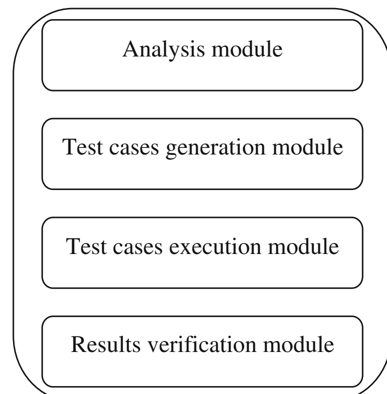

Fig. 2 The EvoGUITest architecture

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

Fig. 3 Randomly generated individuals for testing GUI of a calculator 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

Fig. 4 First generation of individuals for testing a complex GUI component

contains more button controls; therefore it is the one which produces the biggest number of changes inside the GUI. 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:

Individual1 : 2 Fitness : 0.5
Individual2 : 1000 Fitness : 0.3
Individual3 : 65536 Fitness : 0.2

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

The objective function which evaluates each individual is presented in formula (1):

$$
\begin{aligned}
&Objective = (1/\text{no_of_changes}) + \\
&1/(100 \times \text{no_of_similar_states}) + \\
&1/(100 \times \text{no_of_useless_states})
\end{aligned} \tag{1}
$$

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:

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	–	

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 exist in each population while 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 refers to the application of the 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.

Fig. 5 GA parameters settings area

4.2 The EvoGUITest Experimental Results

All the experiments were performed on a computing system 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.

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.

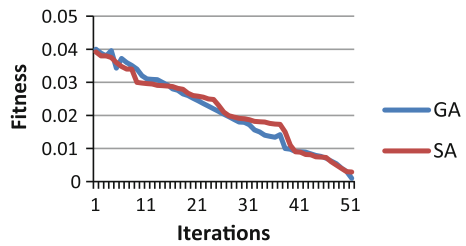

Fig. 6 Test case generation for the simple GUI

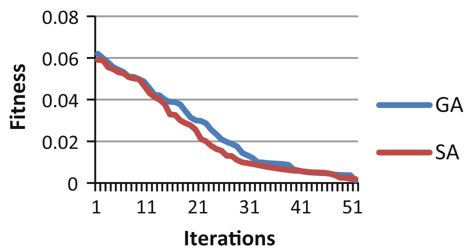

Fig. 7 Test case generation for the Calculator GUI

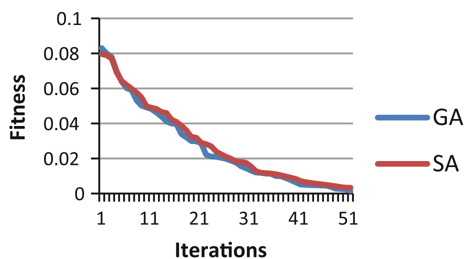

Fig. 8 Test case generation for the complex GUI

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

GUI	GA Performance	No. of GUI Changes	SA Performance	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

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

From Figs. 6, 7, 8 and Table 2 one can notice that the GA is able to find better test data compared to 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 of its 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.

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

4.3 Performance Metrics

To evaluate the testing results obtained with the EvoGUITest framework we made a series of tests using other available open source frameworks.

To facilitate frameworks comparisons and provide information about the frameworks' performances, some metrics had to be defined. The defined metrics are as general as possible, in order to be applicable on any testing software that might be used for testing the GUIs of Web applications.

The metrics we defined are:

- Metrics #1: Number of HTML content errors *per* number of source code lines (NECL)

This metric represents the number of HTML content errors found over the number of source code lines of the tested application. The metric offers an idea about the density of HTML errors in the application source code. Measuring this parameter gives an image of the application's quality.

- Metrics #2: Average number of HTML content errors *per* number of source code lines (ANECL)

This metric represents the average number of HTML content errors over the number of source code lines of the tested applications. This parameter gives an idea about the density of HTML errors compared to the average size of the tested application. Measuring this parameter also gives an image of the tested application's quality.

- Metrics #3: Number of HTML content errors *per* test suite (NETS)

This metric indicates the number of HTML content errors discovered after testing the software application over the number of tests in the test suite. It can be extended for any other applications and test suites. Its extension refers to the total number of HTML errors over the total number of tests run for all the tested applications.

- Metrics #4: Average number of HTML content errors *per* test suite (ANETS)

The metric represents the average number of HTML content errors found over the number of tests in the test suite. This parameter offers an idea about the density of HTML errors discovered by each test from the test suites. It gives an image of the quality of the tests used in the testing process.

- Metrics #5: Number of HTML content errors *per* test (NET)

The metric represents the number of HTML content errors found after testing the software application with one single test from the suite of tests.

- Metrics #6: Average number of HTML content errors *per* test (ANET)

This metric represents the average number of HTML content errors found by a certain testing scenario. It shows the average abilities of a testing scenario to find errors in the tested graphical interface.

4.4 Experimental Results

Figure 9 presents a comparison between four test suites composed of ten test cases each. The test suites were generated with EvoGUITest, Selenium [8], WinRunner [9] and Rational Robot [10]. They were used in the regression testing phase for detecting errors inside a set of benchmark Web applications.

From Fig. 9 one can notice that the test suite generated using EvoGUITest is able to find more defects in comparison with other test suites based on the two functioning modes (i.e. capture and replay), even if they contain the same number of tests. This illustrates the fact that the test suite generated with EvoGUITest is better than those generated with Selenium, WinRunner and Rational Robot frameworks.

Artemis [17], Atusa [18] and Kudzu [19] frameworks for testing the GUIs of Web applications were used to observe the similarities and differences with our EvoGUITest framework.

The number of benchmark applications that were tested in order to make comparisons between EvoGUITest and Artemis, respective Atusa frameworks, was 30, and the number of tests from the tests suite was 100. These Web applications were selected among popular Web applications available on the Internet.

Figure 10 shows the differences between EvoGUITest and Artemis when they were used for finding out errors in different Web graphical interfaces.

The number of errors discovered in the HTML code by EvoGUITest and Artemis are presented in Table 4. In case of just two applications out of 30, the Artemis framework had better results than our EvoGUITest framework.

Using the NECL, NETS and NET metrics defined before, in Table 5 we present the better results obtained by EvoGUITest, compared to those obtained using the Artemis framework. The results clearly show better results for EvoGUITest.

Figure 11 shows the differences between EvoGUITest and Atusa when looking for errors in the 30 benchmark applications tested.
The results of the number of errors discovered in the HTML code by EvoGUITest and Atusa are presented in Table 6. In all the 30 of tested cases, EvoGUITest framework had better results than Atusa framework.

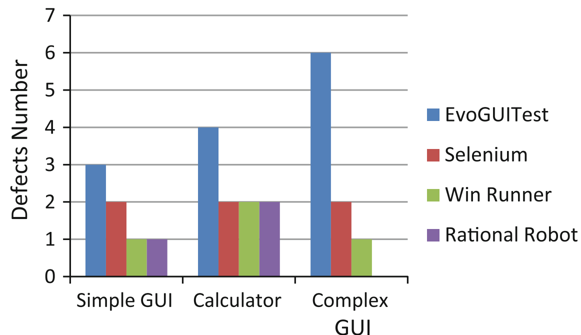

Fig. 9 Number of defects discovered by different testing frameworks

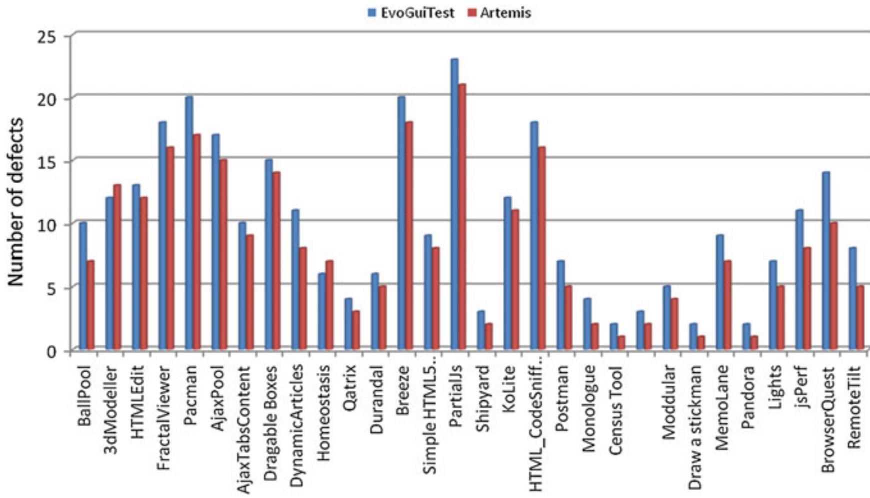

Fig. 10 Errors find out by EvoGUITest and Artemis

Table 4 Errors in HTML code discovered by EvoGUITest and Artemis

Tested Applications	Lines code number	EvoGuiTest (HTML errors number)	Artemis (HTML errors number)
3dModeller	393	12	13
BallPool	256	10	7
FractalViewer	750	18	16
HTMLEdit	568	13	12
Pacman	1857	20	17
AjaxPool	250	17	15
AjaxTabsContent	156	10	9
DragableBoxes	697	15	14
DynamicArticles	156	11	8
Homeostasis	2037	6	7
Qatrix	1712	4	3
Durandal	2159	6	5
Breeze	14730	20	18
Simple HTML5 Drawing Application	439	9	8
PartialJs	5857	23	21
Shipyard	73	3	2
KoLite	381	12	11
HTML CodeSniffer	1433	18	16
Postman	199	7	5

(continued)

Table 4 (continued)

Tested Applications	Lines code number	EvoGuiTest (HTML errors number)	Artemis (HTML errors number)
Monologue	215	4	2
Census Tool	1567	2	1
Computer Language Benchmarks Game	1678	3	2
Moddular	945	5	4
Draw a stickman	785	2	1
MemoLane	567	9	7
Pandora	1128	2	1
Lights	957	7	5
jsPerf	589	11	8
BrowserQuest	1368	14	10
RemoteTilt	688	8	5
Average	1486	10	8

Table 5 Metrics results for EvoGUITest and Artemis

Applications	EvoGuiTest			Artemis		
	NECL	NETS	NET	NECL	NETS	NET
3dModeller	0.03	12	0.12	0.033	13	0.13
BallPool	0.039	10	0.1	0.027	7	0.07
FractalViewer	0.024	18	0.18	0.021	16	0.16
HTMLEdit	0.022	13	0.13	0.021	12	0.12
Pacman	0.01	20	0.2	0.009	17	0.17
AjaxPool	0.068	17	0.17	0.06	15	0.15
AjaxTabsContent	0.064	10	0.1	0.057	9	0.09
DragableBoxes	0.021	15	0.15	0.02	14	0.14
DynamicArticles	0.07	11	0.11	0.05	8	0.08
Homeostasis	0.002	6	0.06	0.003	7	0.07
Qatrix	0.002	4	0.04	0.002	3	0.03
Durandal	0.003	6	0.06	0.002	5	0.05
Breeze	0.001	20	0.2	0.001	18	0.18
Simple HTML5 Drawing Application	0.02	9	0.09	0.018	8	0.08
PartialJs	0.004	23	0.23	0.003	21	0.21
Shipyard	0.04	3	0.03	0.027	2	0.02
KoLite	0.031	12	0.12	0.028	11	0.11
HTML CodeSniffer	0.012	18	0.18	0.011	16	0.16

(continued)

Table 5 (continued)

Applications	EvoGuiTest			Artemis		
	NECL	NETS	NET	NECL	NETS	NET
Postman	0.0351	7	0.07	0.025	5	0.05
Monologue	0.0186	4	0.04	0.009	2	0.02
Census Tool	0.001	2	0.02	0.001	1	0.01
Computer Language Benchmarks Game	0.001	3	0.03	0.001	2	0.02
Moddular	0.005	5	0.05	0.004	4	0.04
Draw a stickman	0.002	2	0.02	0.001	1	0.01
MemoLane	0.015	9	0.09	0.012	7	0.07
Pandora	0.001	2	0.02	0.001	1	0.01
Lights	0.007	7	0.07	0.005	5	0.05
jsPerf	0.018	11	0.11	0.0135	8	0.08
BrowserQuest	0.010	14	0.14	0.007	10	0.1
RemoteTilt	0.011	8	0.08	0.007	5	0.05
	ANECL	**ANETS**	**ANET**	**ANECL**	**ANETS**	**ANET**
Average for NECL, NETS, NET	0.0195	10	0.1	0.0159	8	0.084

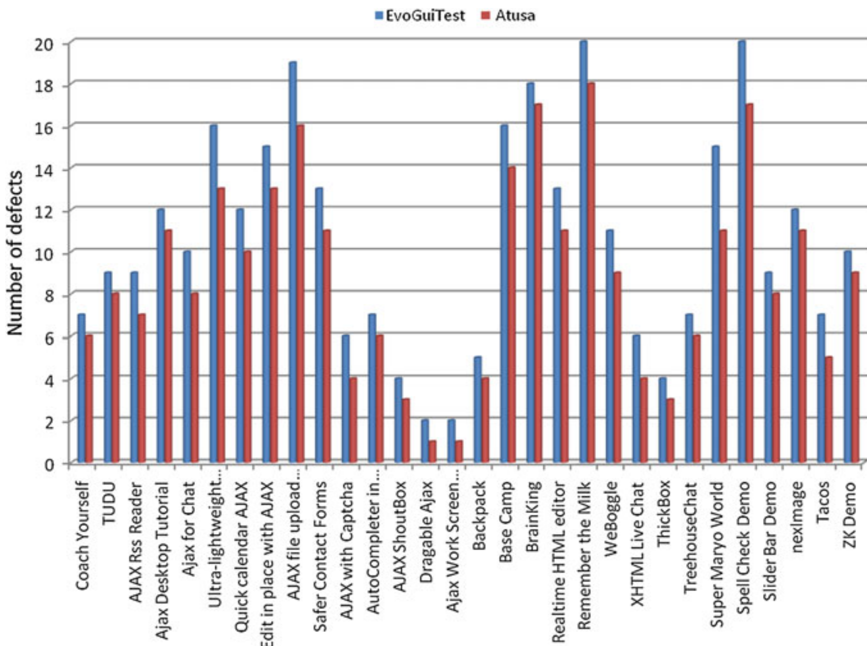

Fig. 11 Errors discovered by EvoGUITest and Atusa

Table 6 Errors in HTML code find out by EvoGUITest and Atusa

Tested applications	Lines code number	EvoGuiTest (HTML errors number)	Atusa (HTML errors number)
TUDU	580	9	8
Coach Yourself	150	7	6
AJAX Rss Reader	600	9	7
Ajax Desktop Tutorial	856	12	11
Ajax for Chat	678	10	8
Ultra-lightweight charts for AJAX	1234	16	13
Quick calendar using AJAX	453	12	10
Edit in place with AJAX	745	15	13
AJAX file upload tutorial	897	19	16
Safer Contact Forms without Captcha	1145	13	11
Using AJAX with Captcha	459	6	4
AutoCompleter in AJAX	239	7	6
AJAX ShoutBox	712	4	3
Dragable Ajax	324	2	1
Making Ajax Work with Screen Readers	123	2	1
Backpack	890	5	4
Base Camp	1256	16	14
BrainKing	2367	18	17
Realtime HTML editor	543	13	11
Remember the Milk	967	20	18
WeBoggle	1145	11	9
XHTML Live Chat	678	6	4
ThickBox	564	4	3
TreehouseChat	343	7	6
Super Maryo World	756	15	11
Spell Check Demo	897	20	17
Slider Bar Demo	235	9	8
nexImage	123	12	11
Tacos	378	7	5
ZK Demo	456	10	9
Average	693	10	8

Table 7 Metrics results for EvoGUITest and Atusa

Aplications	EvoGuiTest			Atusa		
	NECL	NETS	NET	NECL	NETS	NET
TUDU	0.015	9	0.09	0.013	8	0.08
Coach Yourself	0.046	7	0.07	0.04	6	0.06
AJAX Rss Reader	0.015	9	0.09	0.011	7	0.07
Ajax Desktop Tutorial	0.014	12	0.12	0.013	11	0.11
Ajax for Chat	0.0147	10	0.1	0.0118	8	0.08
Ultra-lightweight charts for AJAX	0.013	16	0.16	0.010	13	0.13
Quick calendar using AJAX	0.0264	12	0.12	0.022	10	0.1
Edit in place with AJAX	0.020	15	0.15	0.017	13	0.13
AJAX file upload tutorial	0.021	19	0.19	0.0178	16	0.16
Safer Contact Forms without Captcha	0.011	13	0.13	0.009	11	0.11
Using AJAX with Captcha	0.0130	6	0.06	0.008	4	0.04
AutoCompleter in AJAX	0.0292	7	0.07	0.0251	6	0.06
AJAX ShoutBox	0.005	4	0.04	0.004	3	0.03
Dragable Ajax	0.006	2	0.02	0.003	1	0.01
Making Ajax Work with Screen Readers	0.0162	2	0.02	0.008	1	0.01
Backpack	0.005	5	0.05	0.004	4	0.04
Base Camp	0.0127	16	0.16	0.011	14	0.14
BrainKing	0.007	18	0.18	0.007	17	0.17
Realtime HTML editor	0.024	13	0.13	0.020	11	0.11
Remember the Milk	0.008	20	0.2	0.007	18	0.18
WeBoggle	0.009	11	0.11	0.007	9	0.09
XHTML Live Chat	0.008	6	0.06	0.005	4	0.04
ThickBox	0.007	4	0.04	0.005	3	0.03
TreehouseChat	0.020	7	0.07	0.017	6	0.06
Super Maryo World	0.019	15	0.15	0.014	11	0.11
Spell Check Demo	0.022	20	0.2	0.018	17	0.17
Slider Bar Demo	0.038	9	0.09	0.034	8	0.08
nexImage	0.097	12	0.12	0.089	11	0.11
Tacos	0.018	7	0.07	0.013	5	0.05
ZK Demo	0.021	10	0.1	0.019	9	0.09
	ANECL	**ANETS**	**ANET**	**ANECL**	**ANETS**	**ANET**
Average for NECL, NETS, NET	0.0193	10	0.1	0.0154	8	0.088

Using the NECL, NETS and NET metrics in Table 7 we present the results obtained by EvoGUITest and compare them to those obtained by Atusa framework. The results show that EvoGUITest had better results than Atusa for all metrics.

The third comparison was made with the Kudzu framework. For 28 benchmark applications that were tested, Fig. 12 shows the errors obtained by EvoGUITest and by Kudzu frameworks.

The number of errors discovered in the HTML code by EvoGUITest and Kudzu for 28 benchmark applications are presented in Table 8. In the tested cases when errors were discovered, EvoGUITest framework had better or same results than Kudzu framework. In just one single case (the TVGuide application), Kudzu discovered one error while EvoGUITest was unable to find any error.

Using the NECL, NETS and NET metrics in Table 9 we present the results obtained by EvoGUITest framework, compared to Kudzu framework. The results show that EvoGUITest had better results than Kudzu.

All the comparisons made until now show that the results obtained with EvoGUITest framework in automated testing of graphical user interfaces for Web applications have a better quality. The EvoGUITest framework managed to find out more errors in the HTML code and to generate more performing test suites than others similar frameworks.

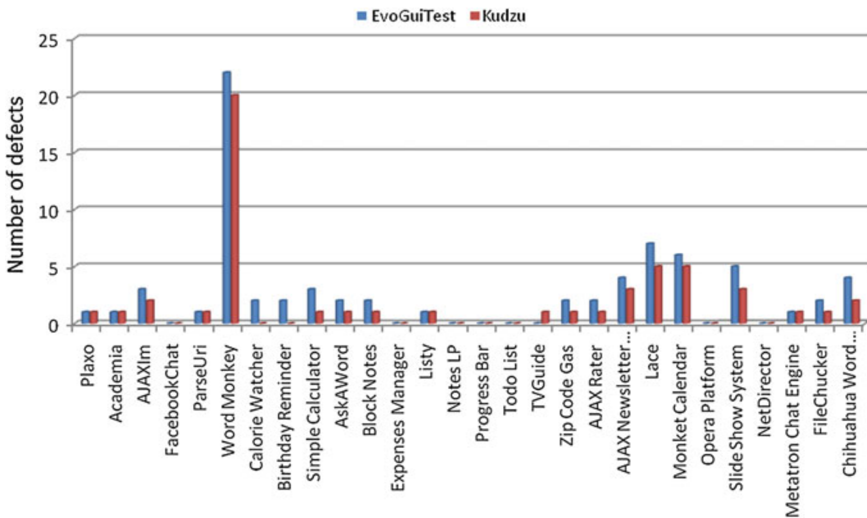

Fig. 12 Errors discovered by EvoGUITest and Kudzu

Table 8 Errors in HTML code discovered by EvoGUITest and Kudzu

Tested Applications	Lines Code Number	Tests Suite Dimension	EvoGuiTest(HTML errors number)	Kudzu (HTML errors number)
Plaxo	17854	178	1	1
Academia	1604	16	1	1
AJAXIm	9328	93	3	2
FacebookChat	15789	144	–	–
ParseUri	601	6	1	1
Word Monkey	5437	93	22	20
Calorie Watcher	2808	28	2	–
Birthday Reminder	1678	16	2	–
Simple Calculator	2340	93	3	1
AskAWord	2690	93	2	1
Block Notes	1798	28	2	1
Expenses Manager	3608	32	–	–
Listy	2564	26	1	1
Notes LP	3516	30	–	–
Progress Bar	1439	15	–	–
Todo List	2076	20	–	–
TVGuide	3789	32	–	1
Zip Code Gas	5587	54	2	1
AJAX Rater	4312	47	2	1
AJAX Newsletter Signup	4789	87	4	3
Lace	3912	76	7	5
Monket Calendar	2945	34	6	5
Opera Platform	12347	120	–	–
Slide Show System	5489	67	5	3
NetDirector	3675	104	–	–
Metatron Chat Engine	4012	81	1	1
FileChucker	2934	72	2	1
Chihuahua Word Puzzle	5476	55	4	2
Average	4799	62	3	2

Table 9 Metrics results for EvoGUITest and Kudzu

Aplications	EvoGuiTest			Kudzu		
	NECL	NETS	NET	NECL	NETS	NET
Plaxo	0	1	0.005	0.00005	1	0.005
Academia	0.0006	1	0.062	0.0006	1	0.062
AJAXIm	0.0003	3	0.032	0.0002	2	0.021
FacebookChat	0	0	0	0	0	0
ParseUri	0.0016	1	0.166	0.0016	1	0.166
Word Monkey	0.004	22	0.23	0.0036	20	0.21
Calorie Watcher	0.0007	2	0.071	0	0	0
Birthday Reminder	0.0011	2	0.12	0	0	0
Simple Calculator	0.0012	3	0.032	0.0004	1	0.01
AskAWord	0.0007	2	0.021	0.0003	1	0.01
Block Notes	0.0011	2	0.071	0.0005	1	0.035
Expenses Manager	0	0	0	0	0	0
Listy	0.0003	1	0.038	0.0003	1	0.038
Notes LP	0	0	0	0	0	0
Progress Bar	0	0	0	0	0	0
Todo List	0	0	0	0	0	0
TVGuide	0	0	0	0.0002	1	0.031
Zip Code Gas	0.0003	2	0.037	0.0001	1	0.018
AJAX Rater	0.0004	2	0.042	0.0002	1	0.021
AJAX Newsletter Signup	0.0008	4	0.045	0.0006	3	0.034
Lace	0.0017	7	0.092	0.0012	5	0.065
Monket Calendar	0.002	6	0.176	0.0016	5	0.147
Opera Platform	0	0	0	0	0	0
Slide Show System	0.0009	5	0.074	0.0005	3	0.044
NetDirector	0	0	0	0	0	0
Metatron Chat Engine	0.0002	1	0.012	0.0002	1	0.012
FileChucker	0.0006	2	0.0277	0.0003	1	0.013
Chihuahua Word Puzzle	0.0007	4	0.0727	0.0003	2	0.0363
	ANECL	**ANETS**	**ANET**	**ANECL**	**ANETS**	**ANET**
Average for NECL, NETS, NET	0.0006	3	0.050	0.0004	2	0.034

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 checks the biggest possible 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. To the best of our knowledge, 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 to make 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 discover the most important sequence of GUI events.

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 correctly 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.

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

References

1. Jansen, B.J.: The Graphical User Interface: an introduction. In: Seminal Works in Computer Human Interaction, vol. 30(2), pp. 24-26. ACM, New York (1998)
2. Pimenta, A.: Automated Specification-based Testing of Graphical User Interfaces. Ph.D. Thesis, Department of Electrical and Computer Engineering, Porto University, Portugal (2006)
3. Ganov, S., Killmar, C., Khurshid, S., Perry, D.: Test generation for graphical user interfaces based on symbolic execution. In: Proceedings of the 3rd International Workshop on Automation of Software Test , pp. 35-40. ACM, New York (2008)
4. Yang, X.: Graphic User Interface Modelling and Testing Automation. Ph.D. Thesis, School of Engineering and Science, Victoria University Melbourne, Australia (2011)
5. Al-Zain, S., Eleyan, D., Hassouneh, Y.: Comparing GUI automation testing tools for dynamic web applications. Asian J. Comput. Inf. Syst. **01**(02), 38–48 (2013)
6. Kaur, M., Kumari, R.: Comparative study of automated testing tools: testcomplete and quicktest pro. Int. J. Comput. Appl. **24**(1), 1–7 (2011)

7. Bergen, S., Ross, J.: Evolutionary art using summed multi-objective ranks. In: Genetic Programming Theory and Practice VIII, vol. 8, pp. 227–244. Springer Science, Berlin (2011)
8. Valdez-Garcia, M., Trujillo, l., Fernandez de Vega, F., Guervos, J.M., Olague, G.: EvoSpace-Interactive: a framework to develop distributed collaborative-interactive evolutionary algorithms for artistic design. In: Evolutionary and Biologically Inspired Music, Design, Sound Art and Design, LNCS, vol. 7834, pp. 121–132. Springer (2013)
9. Prabhu, J., Malmurugan, N.: A survey on automated GUI testing procedures. Eur. J. Sci. Res. **64**(3), 456–462 (2011)
10. Selenium Framework. http://seleniumhq.org
11. WinRunner Framework. http://mercury.com
12. Rational Robot Framework. http://www-01.ibm.com/software/awdtools/tester/robot/
13. Nyman, N.: In defense of monkey testing. In: Software Testing and Quality Engineering Magazine, pp. 18–21 (2000)
14. NUnit Framework. http://nunit.org
15. Abbot GUI libraries. http://abbot.sourceforge.net
16. Belli, F.: Finite-State testing and analysis of Graphical User Interfaces. In: Proceedings of the 12th International Symposium on Software Reliability Engineering, pp. 34–43. IEEE Xplore (2001)
17. Qureshi, I.A., Nadeem, A.: GUI testing techniques: a survey. Int. J. Future Comput. Commun. vol. 2(2), pp. 142–146 (2013)
18. Web Guitar. http://www.cs.umd.edu/~atif/GUITAR-Web
19. Artzi, S., Dolby, J., Jensen, S. H., Moller, A., Tip, F.: A framework for automated testing of JavaScript web applications. In: Proceedings of the 33rd International Conference on Software Engineering, pp. 571–580. ACM, New York (2011)
20. Mesbah, A., Van Deursen, A.: Invariant-Based automatic testing of AJAX user interfaces. In: Proceedings of the 31st International Conference on Software Engineering, pp. 210–220. IEEE Computer Society Washington, DC (2009)
21. Saxena, P., Akhawe, D., Hanna, S., McCamant, S., Song, D., Mao, F.: A symbolic execution framework for JavaScript. In: Proceedings of 31st IEEE Symposium on Security and Privacy, pp. 513–528. IEEE Computer Society Washington, DC (2010)
22. Jones, G.: Genetic and evolutionary algorithms. In: Encyclopedia of Computational Chemistry, pp.1–10. Wiley (1990)
23. Pohlheim, H.: Evolutionary algorithms: overview, methods and operators. In: Geneticand Evolutionary Algorithm Toolbox for Matlab (2006)
24. Streichert, F.: Evolutionary Algorithms in Multi-Modal and Multi-Objective Environments, Ph.D. Thesis, Department of Computer Architecture, University of Tubingen, Germany (2007)
25. Rauf, A.: Coverage Analysis for GUI Testing, Ph.D. Thesis, Department of Computer Science, National University of Computer and Emerging Sciences Islamabad, Pakistan (2010)
26. Bertsimas, D., Tsitsiklis, J.: Simulated annealing. Stat. Sci. **8**(1), 10–15 (1993)
27. Ruthenbar, R.: Simulated Annealing algorithms: an overview. IEEE Circuits Devices Mag. **5**, 19–26 (1989)
28. Nascimento, V., Carvalho, V., Castilho, C., Soares, E., Bittencourt, C., Woodruff, D.: The simulated annealing global search algorithm applied to the crystallography of surfaces by Leed. In: Surface Review and Letters, vol. 6(5), pp. 651–661 (1999)