การประชุมวิชาการทางวิศวกรรมไฟฟ้า ครั้งที่ 45
The 45th Electrical Engineering Conference (EECON-45)
วันที่ 16-18 พฤศจิกายน 2565 ณ ภูสักธาร รีสอร์ท อำเภอเมือง จังหวัดนครนายก

GUI Testing Using Genetic Algorithm for Web Applications

Pipat Sookavatana[1] and Nipat Liampisan[2]

[1]Department of Computer Engineering, King Mongkut's Institute of Technology Ladkrabang, Thailand
pipat.so@kmitl.ac.th
[2]Department of Computer Engineering, King Mongkut's Institute of Technology Ladkrabang, Thailand
nliampisan@gmail.com

Abstract

Graphical user interfaces (GUI) testing has always been a challenging and labor-intensive part of software development. Previous methods of GUI testing have involved manual test execution or test script development and maintenance. The problem with test scripts for web pages is that the attributes of DOM elements are always changing, therefore test scripts have to be frequently fixed. A simpler solution for GUI testing of web applications is to use random testing approaches. In this paper, two random testing approaches are compared. The baseline model is the monkey testing approach and the proposed approach is through a genetic algorithm. After experimentation, on average the genetic algorithm had a 12% higher statement coverage than monkey testing.

Keywords: GUI testing, test generation, web applications, genetic algorithms

1. Introduction

GUIs are the way users interact with the software. Thus, it is important to thoroughly test it. Such testing can also elicit faults in the business logic of the software as well. A typical Graphical User Interface (GUI) gives an enormous amount of freedom of interaction to the user, so a test designer needs to develop test cases that examine the interaction space. Due to the large space, test designers usually generate test cases to traverse a subspace that maximizes fault detection. This is a very challenging issue because even a very small program can have a large number of unique sequences, i.e., a 5 event GUI can have 125 unique three length sequences.

Another issue with GUI testing of web applications is that the application is constantly changing. Therefore, it is hard to maintain test cases. GUI testing depends on locating an element and performing an action on the element. Element locators are not reliable since any small change can cause the whole test to fail. Locators are not robust enough to deal with slightly changes in a software. As a software is being developed, locators need to be repaired so that it points to the new elements. This is a major cost in web application testing. Diving by aforementioned issues, we proposed a method to generate test scripts for GUIs of web applications, employing genetic algorithm.

The remainder of this paper is organized as follows: in Section 2, a review on published works in relation with web GUI and software testing, Section 3 introduces our proposed method to generate web GUI test scripts, Section 4 presents experimental results, and Section 5 concludes this paper and presents future works.

2. Background and related work

2.1 Monkey Testing

One way to avoid writing automated test scripts for web GUIs is to use a process called Monkey Testing to generate test scripts. The term "monkey testing" was presented by Nyman [1]. Nyman described a practical application of automating exploration of GUI applications with random tests. Monkey testing is simple and easy to use, but one of its issues is that it is a random interaction. Therefore, sometimes it executes tests that a typical human user is unlikely to perform. However, due to its randomness it also has a high chance of discovering new defects [2].

Wetzlmaier et al [3] implemented a framework for random "monkey" GUI testing. The framework randomly interacts with the GUI, while also monitoring the system under test's responses and behaviors to detect failures. To randomly interact with the GUI, the framework first extracts GUI elements, i.e., buttons and text fields from the window, and then randomly chooses an action on an element from a set of preprogrammed interactions. Wetzlmaier et al [4] combined this framework with prewritten GUI test cases to create a hybrid approach. Their approach was evaluated on open-source application KeePass 2, a Windows password manager. They found that on average the added random interactions increased the number of visited application windows per test by 23.6% and code coverage by 12.9%.

2.2 Genetic Algorithms

In previous research papers, genetic algorithms have been used to find optimized test cases that maximize code coverage and repair element locators. Genetic algorithm (GA) is a population-based search or optimization algorithm inspired by biological evolution. GA aims to create genes that maximize or minimize the fitness function given some constraints. First, genes are randomly generated. Then genes that have good fitness are selected and reproduced for the next phase. This is done until a termination point is reached [5]. M. Leotta et al [6] proposed a genetic algorithm to solve the problem of automatically generating robust XPath locators for DOM elements. Each chromosome is a random XPath expression. The mutation of a chromosome can occur by adding or removing a tag name, a predicate and adding or removing a level. A crossover of a chromosome can only be applied, if there are two or more levels in the XPath expression. One point or two-point crossover occurs at the level separators producing two child chromosomes. The

fitness function is the number of web page elements selected by the XPath expression in the chromosome. The aim is to minimize the fitness function, so the chromosome selects only one element [7]. H. M. Eladawy et al [8] also proposed a genetic algorithm to repair locators, which are used to select elements from web pages. The biggest difference here is the fitness function is a comparison between the gene of the current element and the old element. This is measured by calculating the *Levenshtein distance* between the genes of the old element and the corresponding genes of that individual.

2.3 Statement Coverage

One way to show that there has been enough testing is statement coverage. This is a measure of how much of the system has been exercised during testing. However, high statement coverage is hard to attain. This is true especially if there are nested conditions and exception handling. To cover all these cases many test scripts must be written. In their experiment, the tool that is used to measure statement coverage is *Coverage.py*. This tool is written for measuring statement coverage of Python programs [9]. By default, it measures line (statement) coverage, but it can also measure branch coverage. To get the results, the tool first keeps track of the parts of code that were executed, then analyzes the parts of code that could have been executed but was not.

3. Proposed Approach

The aim of our model is to generate a test case that maximizes test coverage. The reason behind maximization is to expand the subspace to search for bugs/faults as much as possible. To accomplish this, we employ a genetic algorithm.

3.1 Chromosome/Population

The chromosome for this model is a sequence of user actions. To be more specific, it is a sequence of actions on web elements [10]. A chromosome is meant to represent one test case. A chromosome is a set of genes, and a set of chromosomes is a population.

3.2 Gene

Each gene in our model is a web element. Possible elements that will be interacted include links, buttons, input texts, and text areas. Each element has a specific action that is related to it. Links and buttons only have the action click associated with it, while the text area and input text have the action send keys associated with it. A gene's element is located using the Xpath expression or Tag. This may lead to a return of more than one web element. If this happens, then one element is randomly chosen from the set of returned elements. If the locator does not find any elements, then the gene will point to nothing.

3.3 Selection

The strategy used in the model for selection is the Tournament selection strategy. In this, the fittest candidate is chosen from a random set of k-individuals. This will lead to the fittest candidates being passed on to the next generation. Tournament selection is used for exploration of test cases to happen.

3.4 Mutation

A mutation in this model means that the web element associated with the gene will change. In this paper, the gene will either change to a link, a button, an input text or a text area. All elements are equally possible to be chosen. The mutation is meant to slow down the convergence of the algorithm.

3.5 Crossover

In our model a one-point crossover is applied. A crossover creates offspring chromosomes by combining a pair of parent chromosomes. In our model the crossover points will be at the subdivision between web pages. A constraint to the crossover is that both genes after the crossover point must be located at the same web page. This is because the action of the gene before the crossover point might change the current webpage to a new webpage. This means that the gene after the crossover point must also be located on the webpage or the action will fail, decreasing the overall fitness. Decreasing overall fitness does lead to a convergence, thus the constraint is added.

3.6 Fitness Function

The fitness function will be based on two criteria: (1) the number of successful actions and (2) the amount of different web pages converged. The fitness of each chromosome is calculated by executing the test case on the system that is being tested and keeping count of the two criteria. By trying to maximize these two criteria, it will indirectly lead to an increase in total coverage. Given a chromosome (test case) "s", the fitness function is defined as

$$fitness(s) = \sum_{i=1}^{n} W_i X_i + bY \quad (1)$$

The parameters b and weights W_i associated with X_i are non-negative numbers. Parameter b can be adjusted to how important it is to check new webpages. X_i represents the total number of successful actions on an element of type $i-1$, i.e., X_0 represents the number of times that clicks on an element of type link was successful. Each X_i has an associated weight and its default is one, but it can also be adjusted to a non-negative number to promote certain element actions. The number of element types is represented by the variable n. In this experiment, the n is set to 4, as there are 4 element types (link, button, input text and text area). Each element type has a weight of 1 and b is set to 10.

3.7 Initial Population

The chromosome is generated by starting at the web application's homepage. The starting gene will be a web element that is randomized and acted upon. The next gene will be assigned and acted upon after the first gene. This will happen iteratively until the length of the chromosome is reached. When a new chromosome is

created, the web application resets at the homepage again and the same process begins.

3.8 Termination Point

The termination point for the algorithm is when the number of generations has exceeded the set maximum. The algorithm starts at generation zero and runs until the set maximum number of generations.

4. Development /Implementation

4.1 Selenium

To develop code that interacts with the web application's GUI, Selenium WebDriver is used. In this study, Google Chrome was used as the browser. To find the elements to interact with, WebDriver's element locator was used. The locator can take different attributes such as class, tag, id, name or Xpath and return elements that fit the given value. In this study, Xpath was used to locate the elements. For each gene, an element would be located randomly, and the corresponding action would be used.

4.2 A. Monkey Testing Implementation

In this experiment, the monkey testing is implemented by extracting certain elements from the web page. The elements extracted are links, buttons, input text, text area, images, paragraph, and headings. These are extracted using Xpaths and Tags. After the elements are extracted, then one is randomly chosen from the set and the appropriate user action is tried on the element. For input text and text area it is send keys or all the other elements it is the click action. The amount of user actions it does is controlled by the variable N.

4.3 Genetic Algorithm Implementation

The genetic algorithm itself is designed and developed using object-oriented programming. There are five main classes used to implement the genetic algorithm: GA, ParameterSet, Breeder, Population, and Evaluator. The GA class is responsible of running the genetic algorithm and plotting the results. This is where the loop happens, and the termination point is checked. The GA class contains the parameter set class, the population class, the evaluator class, and the breeder class. The parameter set class contains the parameters that are needed to run the genetic algorithm. The parameters contained are maximum generations, chromosome length, population size, tournament size, crossover probability and mutation rate. The GA contains two populations. The parent population and the offspring population. Each chromosome's fitness in the population can be calculated using the class Evaluator. Finally, the Breeder class takes care of the reproduction process where individuals are selected to either be crossed or mutated. After the termination point, the fittest individual can be returned by the GA class.

4.4 Genetic Algorithm Evaluator Algorithm

The evaluator keeps track of the number of successful actions for each element type and the number of web pages traversed. The evaluator loops through the entire length of the chromosome. For each element in the chromosome, the evaluator tries to find all elements of a web element type. Then the evaluator randomly acts on the web element. If it is successful, then it increments the number of successful actions for that element type by one. If the same action also changes the webpage, then the number of web pages traversed is also incremented by one. If the evaluator does not find any element of the selected element type on the page, it skips the current iteration of the loop. At the end of the loop, the fitness is calculated using the fitness function.

5. Experimental Results

5.1 Environment

To evaluate our proposed work, we employed two web applications, which are a portfolio and a blogging web application from the website realpython.com. The web application was run locally. The testing code is directly connected to the localhost through selenium. The experientials were performed on the following criteria, HP Envy Notebook (i7-8550u, 8GB, Windows 10 64-Bit OS), PyCharm IDE and Python V 3.8.1.

5.2 Parameters

The selected parameters described in Table 1, represent the parameters that best maximize the coverage for the genetic algorithm. The population size, tournament selection size, chromosome length and number of generations are set to optimal amounts, so the algorithm can still run properly in the programming environment without freezing. With these constraints, the experiment was initially conducted with values of 0.95 for crossover probability and 0.05 for mutation probability. The initial value for crossover probability converged to one test case too quickly, so the test cases no longer changed much in the later generations. Therefore, the crossover probability was later decreased to 0.90 so that the algorithm could converge while generating sufficiently different test cases. The initial mutation probability made mutation a non-factor in the algorithm. This was due to the small population size and number of generations, causing mutations to happen too infrequently. Therefore, the mutation probability was increased to 0.1 so that mutations can occur more often so that exploration of the optimal test case is possible.

Table 1 Genetic Algorithm Parameters

Parameter	Value
Crossover Probability	0.9
Mutation Probability	0.1
Tournament Selection Size	2
Chromosome Length	10
Population Size	10
No. of Generations	12

5.3 Fitness of Chromosomes

Fig. 1 shows that the chromosome's fitness can vary greatly in the first few generations. However, after a while, the fitness values of the chromosome will start to

converge because only the fittest chromosomes are chosen to reproduce. The difference in fitness value is still a good thing, because it allows for exploration as well so that the genetic algorithm doesn't get stuck to a suboptimal solution.

As seen in Fig. 2, the fitness value increases as the generation increases. At first the fitness value increases quickly but it starts to plateau at higher generations. Because only the fittest chromosome of each generation is chosen, the next generation can never have a lower fitness level than the previous generation. The execution time of 12 generations was 18 mins 33 secs.

```
Generation: 0
gene = [1][0][2][0][3][3][3][1][0], fit =    13.0000
gene = [3][0][3][3][0][3][2][1][0][2], fit =    23.0000
gene = [0][0][0][0][3][1][3][0][1][3], fit =    35.0000
gene = [3][1][2][1][1][3][2][0][3][0], fit =    12.0000
gene = [1][2][0][2][3][1][2][2][3][3], fit =     1.0000
gene = [0][3][1][3][3][1][2][2][0][3], fit =     2.0000
gene = [0][1][3][2][3][0][3][0][2][3], fit =    35.0000
gene = [1][1][1][0][1][1][3][2][2][3], fit =    11.0000
gene = [2][0][3][1][1][3][0][3][2][1], fit =    22.0000
gene = [3][3][2][3][2][0][2][3][0][1], fit =    22.0000
Best-so-far: gene = [0][0][0][0][3][1][3][0][1][3], fit =    35.0000
```

Fig. 1 Fitness of Chromosomes in Population

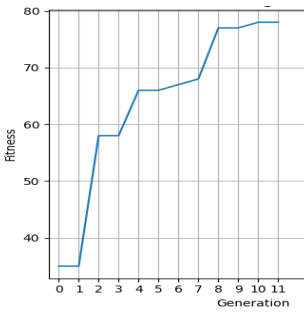

Fig. 2 Best Fitness of Each Generation

5.4 Coverage Results

The As shown in Table 2 and 3, monkey testing on average elicited coverage of the Blog folder and Projects folder that was lower compared to the genetic algorithm. There was a 13.9% increase in Blog and a 10.9% increase in coverage for Projects by using the genetic algorithm. The average of these two increases is 12.4%. A lot of the time, monkey testing doesn't explore all possible webpages. The monkey testing methodology often gets stuck on the starting web page, locating and interacting on web elements that do not result in a transition to a new page. On the other hand, the genetic algorithm produced a higher test coverage. In one instance, it covered 100% of the views.py of in one Project application. From these results, it can be said that even though the fitness function does not directly calculate statement coverage, it can be indirectly used to maximize coverage. This proves that the proposed fitness function is a viable option.

Table 2 Full coverage of one complete run

Method	Coverage Results
Monkey Testing	Blog: 77% files, 40% lines covered Personal portfolio: 66% files, 16% lines Projects: 83% files, 66% lines
Genetic Algorithm	Blog: 77% files, 48% lines covered Personal portfolio: 66% files, 16% lines Projects: 83% files, 75% lines

Table 3 Average coverage results (Number of runs = 10)

Method	Average Line Coverage			Average Fittest Gene Value
	Blog	Personal portfolio	Projects	
Monkey Testing	40.3	16	66	None
Genetic Algorithm	45.9	16	73.2	73.3

6. Conclusion

It is critical for software products to be thoroughly tested before delivery. This is especially true for web applications that are rolled out to users. Rigorous GUI testing is needed to verify that the user can interact with the web application. However, there is a problem time needed to create test cases and fix test cases when web applications are slightly changed. The proposed genetic algorithm implementation solves this problem. It replaces the manual development of test cases with automatic generation of test cases. To maximize the coverage of the test case, the fitness function is evaluated using the number of successful element actions and number of webpages visited. This proposed Genetic Algorithm has a higher coverage when compared to Monkey Testing. However, there is a tradeoff of execution time since it takes a longer time to run the Genetic Algorithm. In the future, we expect to study a method to automatically generate initial parameters b *and* X_i which associate with our equation 1.

References

[1] N. Nyman. 2000. Using monkey test tools: How to find bugs cost effectively through random testing. Software Testing & Quality Engineering Magazine (STQE), (January/February 2000), 18–21.

[2] TY Chen, FC Kuo, RG Merkel, and TH Tse. 2010. Adaptive random testing: The art of test case diversity. Journal of Systems and Software. 83(1), 60-66.

[3] T. Wetzlmaier, R. Ramler, and W. Putschögl. 2016. A framework for monkey GUI testing. In Proceedings of the 2016 IEEE International Conference on Software Testing, Verification and Validation (ICST). IEEE, 416-423.

[4] Thomas Wetzlmaier and Rudolf Ramler. 2017. Hybrid monkey testing: enhancing automated GUI tests with random test generation. In Proceedings of the 8th ACM SIGSOFT International Workshop on Automated Software Testing (A-TEST 2017). Association for Computing Machinery, New York, NY, USA, 5–10. https://doi.org/10.1145/3121245.3121247

[5] M. Alshammari, M. A. Mezher and K. Al-utaibi, "Automatic Test Data Generation Using Genetic Algorithm for Python Programs," 2022 2nd International Conference on Computing and Information Technology (ICCIT), 2022, pp. 197-205, doi: 10.1109/ICCIT52419.2022.9711607.

[6] M. Leotta, A. Stocco, F. Ricca and P. Tonella, "Meta-heuristic Generation of Robust XPath Locators for Web Testing," 2015 IEEE/ACM 8th International Workshop on Search-Based Software Testing, 2015, pp. 36-39, doi: 10.1109/SBST.2015.16.

[7] Hongyu Zhai, Casey Casalnuovo, and Prem Devanbu. 2019. Test coverage in python programs. In Proceedings of the 16th International Conference on Mining Software Repositories (MSR '19). IEEE Press, 116–120. https://doi.org/10.1109/MSR.2019.00027

[8] H. M. Eladawy, A. E. Mohamed and S. A. Salem, "A New Algorithm for Repairing Web-Locators using Optimization Techniques," 2018 13th International Conference on Computer Engineering and Systems (ICCES), 2018, pp. 327-331, doi: 10.1109/ICCES.2018.8639336.

[9] Coverage.py. https://coverage.readthedocs.io/en/v4.5.x/

[10] A. Arora and M. Sinha, "Applying variable chromosome length genetic algorithm for testing dynamism of web application," 2013 International Conference on Recent Trends in Information Technology (ICRTIT), 2013, pp. 539-545, doi: 10.1109/ICRTIT.2013.6844260.