A New Algorithm for Repairing Web-Locators Using Optimization Techniques

Hadeel Mohamed Eladawy
Department of electronics, communication, and computers
Faculty of Engineering, Helwan University
Cairo, Egypt
hadeel.eladawy@h-eng.helwan.edu.eg

Amr E. Mohamed
Department of electronics, communication and computers
Faculty of Engineering, Helwan University
Cairo, Egypt
amr_mohamed@h-eng.helwan.edu.eg

Sameh A. Salem
Department of electronics, communication and computers
Faculty of Engineering, Helwan University
Cairo, Egypt
sameh_salem@h-eng.helwan.edu.eg

Abstract— **The importance of test automation in the software industry has received a growing attention in recent years and it is continuously increasing. Unfortunately, the maintenance of the test cases is a major problem that faces those who use test automation. This problem becomes bigger when dealing with Graphical User Interface (GUI) tests.**

In this paper, an algorithm is introduced to maintain GUI tests for web applications. The Genetic algorithm is adopted to automatically repair the locators used to select elements from web pages. The algorithm is evaluated using several applications and results show that the proposed algorithm improves the repair percentage to 87% of the used locators compared to the previous result which was 73%.

Keywords—GUI testing; web testing; DOM selectors; XPath; XPath locator; genetic algorithm

I. Introduction

In recent years, test automation and writing tests to a certain software product have proven their importance in the software life cycle. These tests not only help in finding defects in the software system, but they also help in better understanding the business domain and the system to be developed; specifically, when these tests are written in the early stages of the software life cycle [1]. The concept of early testing is adopted in new agile methodologies like Test Driven Development (TDD), Behavior Driven Development (BDD) and Acceptance Test Driven Development (ATDD). Although the early definition of the test cases helps in better understanding the software, it increases the need for maintaining these test cases because of increasing their chance to be changed over time. Dudekula et al. [2] stated that the most challenging task in test automation is the maintenance of the test cases. Software products are frequently exposed to changes over time. These changes often lead to the invalidity of test cases and their failure when being run. This limitation becomes clearer when dealing with Graphical User Interface (GUI) tests. GUI tests depend on the Arrange-Act-Assert principle, where first an element is located, an action is performed on the element and finally, the result is checked. In this context, GUI tests are extremely fragile because the used locator easily becomes obsolete with any small change in the application's GUI and thus causing the whole test to fail.

The fragility of GUI tests has been a major problem that disturbs those who use test automation so much [1], [2] and [3]. From a study that was made on Adobe's Acrobat Reader, it was found that 74% of its test cases fail between two successive releases [4]. Moreover, the simplest individual GUI change can cause failure to 30% - 70% of a system's test cases [5]. To repair these defects, test engineers must debug and rewrite those test cases [6]. This maintenance process was found to be so expensive costing, for example, the Accenture company $120 million per year [5]. As reported in [6] and [7], the main cause of fragility of the GUI tests for web applications was the failure of the used locator to obtain the right element.

In this paper, a solution is proposed to help in decreasing the fragility of GUI tests for web applications by automatically repairing the used locators; in order to select the right element; using Genetic Algorithm (GA) [9].

The rest of the paper is organized as follows: Section II presents a background about different ways of locating elements and shows the related work. In Section III, we define the problem and present the proposed algorithm. The evaluation and results are presented in Section IV. Finally, Section V summarizes our conclusions.

II. Background and Related Work

In web applications, GUI tests are concerned with performing actions on certain elements in different pages of the application; e.g. finding a button to click on, finding a text field to write in, etc. and comparing the output of these actions with a predefined expected result.

To perform an action on any element, the element must be firstly located. Elements of web applications could be located using different types of locators such as Coordinate-based, Visual and Document Object Model (DOM)-based locators. *Coordinate-based locators* find an element based on its XY coordinates in the page. *Visual locators* use image recognition techniques to find the element. *DOM-based locators* search for an element using information about its attributes and location in the DOM structure. Leotta et al. [10] stated that DOM-based locators have more robustness than both coordinate-based and visual locators.

A. Document Object Model [11]

The Document Object Model (DOM) is a World Wide Web Consortium (W3C) standard, which defines a programming interface used to manipulate HTML and XML documents. It describes the logical structure of documents and their access and control methods. By using the DOM, documents can be constructed and accessed through their structure. In addition, actions such as adding, changing or deleting elements can be achieved. DOM-based locators use the DOM information for an element to locate and perform any action on it. The most famous DOM-based locator is the XML Path Language (XPath) [12].

B. XPath

XPath is a W3C recommendation that defines a way to select nodes from a DOM structure using path expressions. It locates an element according to its position from the root element (absolute XPath), or relative to another element within the DOM tree (relative XPath). An XPath expression may fail to locate the required element if the DOM structure changes.

The fragility of the used locators is a major problem facing anyone dealing with automated GUI tests [7]. Researches have been made to improve the maintainability of the used locator so that it will not fail at later versions of the Application Under Test (AUT).

C. Related work

In the field of maintaining GUI test cases, most of the current researches are concerned with fixing the XPath locators that have failed to capture the required element and trying to make them less fragile as possible.

Inigo Aldalur and Oscar Diaz [13] provided a solution to fix broken locators used by web extensions and locate the element using its attributes. If there is no single element obtained, then it tries all combinations of attributes of the old element one by one. If there is no unique node located using these attribute combinations, the attributes of the ancestor node of this element are used. This process is successively repeated by considering more ancestors until a single node is detected. If no unique node is located and all the ancestor elements were used, it is considered a failure.

To decrease the amount of work needed to repair a test case, Thummalapenta et al. [14] provided a test automation tool called ATA where a locator is manually fixed in one test case and this fix will be propagated to all other test cases where this locator is used.

Choudhary et al. [6] try to repair a broken locator by also using the attributes that exist in the old. If an element is located using one of these attributes, the test case is executed. If the test case passes, the locator is updated to match that node. If not, the algorithm finds the most similar node by computing the Levenstein's distance between the XPath of the old node and nodes in the new DOM tree.

Leotta et. al. [15] compared the maintainability of different locators used in web applications and found that the use of ID locators; that find elements by their HTML ID; is much better than XPath locators that depend only on the location of the element in the DOM tree. These locators can also be improved by adding the text displayed in the element to them. Leotta et. al. [8] then developed an algorithm that uses multi locators generated by different tools to select an element. Each of these locators is given a weight to indicate its ability to survive among different versions of the web application. If any of these locators could still obtain a unique element, then the one with the highest weight is selected. In [16] ROBULA+ tool was developed. This tool aims to generate a robust locator for a web element using attribute prioritization and blacklists. In this work, an algorithm is developed to uniquely define an element using its attributes. These attributes are prioritized as reported in [15]. In addition, Leotta blacklisted some attributes as he noted that they always fail to locate an element after some changes have occurred to a web page.

III. THE PROPOSED ALGORITHM

In this paper, an algorithm that helps to maintain GUI tests is proposed. The proposed algorithm tries to repair GUI test cases which failed as a result of not finding the required element.

To locate an element, its HTML attributes are used. In the proposed algorithm, a combination of thirteen attributes is considered for each element (*id, name, text, value, title, placeholder, alt, src, class, type, x-location, y-location,* and *tag name*). The stated attributes were selected based on the study addressed and [15] and [17] which prioritize their ability to successfully locate an element among different versions of an application. For each element, the mentioned attributes are acquired during the first time of executing a test case. In this context, the locator used in the test case is assumed to be correct and can successfully locate the element. At this time, all the attributes of the located element are acquired to be introduced to the suggested repairing algorithm whenever the locator fails to locate the element in later versions of the AUT.

To define the average number of elements that exist in a single web page, twenty-three pages from nine different websites and applications *Amazon, BBC, Claroline, Film Affinity, Fox News, Goodreads, NY Times, Wikipedia and YouTube* were considered. The average number of elements in a single web page was nearly 1200 elements. To find the correct element, a similarity measure is used to obtain the matching between the attributes of the original element and the corresponding attributes of all other elements. In that case, all different combinations of these attributes should be considered. This will give 2^ν combinations, where ν is the number of attributes. Thus, the search space shrinks and expands according to the number of elements in the page and the number of the considered attributes. The maximum number of comparisons to find the required element is $2^\nu * \mu$ where μ is the number of elements in the page. In this study, by considering 1200 elements with 13 attributes, this will give about 9,830,400 comparisons where $\nu = 13$, and $\mu = 1200$.

To avoid this large number of comparisons, some randomness can be used for determining which attributes to be considered, and determining which elements are the most likely to be the right one. Therefore, locating the right element is considered as an optimization problem; where there is a need to find the most appropriate element from all the elements in an HTML page, using the least number of attributes within the least time. In that context, the problem was formulated as an optimization problem so that robust optimization algorithms can be applied. In this paper, the Genetic Algorithm is selected as it is one of the most popular optimization algorithms.

To apply the GA, the following assumptions were made to prepare the environment:

- Each element within the HTML page is an *Individual* or a *Chromosome*
- All elements in the HTML page represent the *Population*
- The HTML attributes of each element are the *Genes* of that individual
- Each attribute has a predefined weight that is used when calculating the *fitness function*

When applying any GA, the population should be first initialized, crossover is performed among some randomly selected individuals, mutation is then applied on some individuals and finally, the best individuals are selected; according to a fitness function; to be transferred to the next generation. This process continues until the solution is found, or for a specific number of iterations. These steps are followed in the suggested algorithm to repair the failing locators as shown in the following pseudo-code:

The proposed algorithm

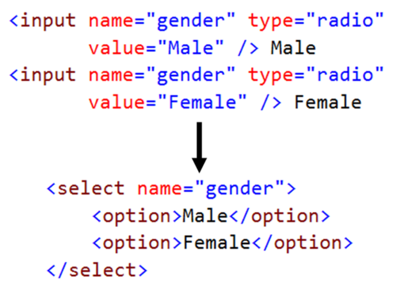
```
Input: DOM_Locator
Output: The best matching element

1.  BEGIN
2.      element = find_element(DOM_Locator)
3.      IF (element is NULL)
4.          population = initialize_population ()
5.          iterations = N
6.          FOR (i = 0 to iterations) DO
7.              population.Do_Crossover(crossoverPercent)
8.              population.Do_Mutation(mutationPercent)
9.              population.Do_Selection()
10.         ENDFOR
11.         return the fittest individual in the population
12.     ELSE
13.         save_element_attributes(element)
14.     ENDIF
```

A. Population initialization

The initial population size is tried to be reduced as possible to fasten the process of finding the element. To do that, we select all elements from the page that exactly match the old element in only one gene (attribute). If no similar elements found, we initialize the population with all elements in the page excluding some elements that never can be used in a test case e.g. html, script, style, link, head, body, …etc.

B. Crossover

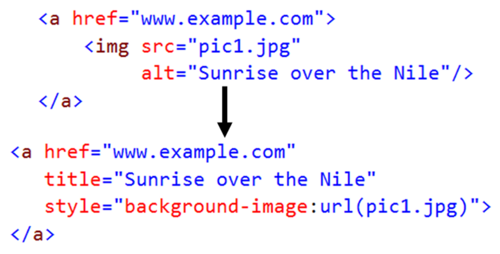
In new versions of web applications, an element may change its type or exchange the values of its attributes. For example, radio buttons can be changed to a drop-down list *"Fig. 1,"* or an image could be changed to an anchor with the image set as a background and the image alt becomes the title or a class of the anchor *"Fig. 2"*. *"Fig. 3"* shows an example of an element that exchanged the values of two of its attributes. These changes cause failure when locating the required element. Trying to imitate this situation in order to solve this problem, crossover is used. To perform crossover, several individuals are randomly chosen from the population according to a specified percentage. Then, randomly two genes are chosen within each individual and exchange their values. Some constraints are applied here. First, if the selected gene represents the tag name, we exchange the value to an appropriate one, e.g. (anchors and buttons) are exchangeable and (checkboxes, radio buttons, drop-down lists) are exchangeable. Another constraint is that some genes are excluded from the crossover process like the location, href and



```
<input name="gender" type="radio"
       value="Male" /> Male
<input name="gender" type="radio"
       value="Female" /> Female
```

```
<select name="gender">
    <option>Male</option>
    <option>Female</option>
</select>
```

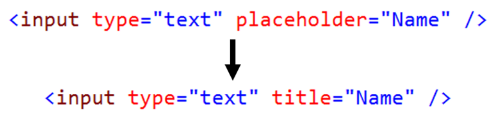
Fig. 1 Radio buttons that changed to drop-down list



```
<a href="www.example.com">
    <img src="pic1.jpg"
         alt="Sunrise over the Nile"/>
</a>
```

```
<a href="www.example.com"
   title="Sunrise over the Nile"
   style="background-image:url(pic1.jpg)">
</a>
```

Fig. 2 An image that changed to an anchor



```
<input type="text" placeholder="Name" />
```

```
<input type="text" title="Name" />
```

Fig. 3 A textfield that exchanged its placeholder attribute with the title attribute

src because their values weren't exchangeable with other genes.

C. Mutation

Mutation helps to keep diversity in the population from one generation to the next. This is applied in the proposed algorithm using what we called the *isConsidered* feature. This feature determines which attributes to be considered when calculating the fitness of each individual. By neglecting (isConsidered=0) or considering (isConsidered=1) the values of some genes, new individuals could be given a chance to arise in the next generation. So, some individuals are randomly chosen from the population according to a specified percentage and the value of their isConsidered is changed from 0 to 1 or vice versa.

D. Fitness function

The fitness of an individual is determined by the degree of similarity between that individual and the old element. This is measured by calculating the Levenstein's distance between the genes of the old element and the corresponding genes of that individual. The following pseudo-code shows the details of calculating the fitness of an individual.

Calculating the fitness function

```
Input: old_element, Individual
Output: fitness of the Individual

1.  BEGIN
2.    fitness = 0
3.    FOREACH gene IN Individual.Genes DO
4.      IF gene.isConsidered DO
5.         d = Levenstein_Distance(gene, old_element.gene)
6.         fitness = fitness + gene.weight * (1-d)
7.      ENDIF
8.    ENDFOREACH
9.  END
```

From the pseudo-code, we can see that only genes with isConsidered = 1 are taken into account. Then we measure the normalized Levenstein's distance between the value of the gene in the individual and the old element. Each gene has a weight that resembles the importance of that gene and how probable it's suggested to change. These weights are prioritized according to the results obtained by [16]. The value of the weight ranges from 0.1 to 1. For example, the id gene has weight 1 but the location gene has a weight 0.1 because the location of an element is more likely to change.

E. Selection

After doing crossover and mutation, new individuals have been added to the population. The next generation should have the same population size, so the fittest individuals are chosen from the population and the rest are deleted. The roulette wheel selection policy was adopted to select individuals for the next generation.

F. Stopping criteria

Any GA runs until the solution is found or for a specific number of iterations. In our case, setting a stopping criterion is difficult because we can't set a value for the fitness that when reached the algorithm can stop. So, for now, the suggested GA stops after a predefined number of iterations. Defining a stopping criterion would be done as a future work.

IV. EVALUATION

The purpose of this study is to analyze the ability of the implemented algorithm to repair a locator to find the required element in a later version of an application. The suggested algorithm was implemented in C# and Selenium WebDriver and tests were run on a laptop Lenovo Intel® Core™ i5-7200U CPU (2.5 GHz), 8 GB RAM.

A. Environment Setup

To evaluate the proposed algorithm, it was tested on a number of the applications suggested by [13] and [16]. These applications differ in the number of elements per page, the complexity of the HTML structure, and the amount of attribute usage per element as shown in *"Table I."* The complexity of the HTML structure defines the number of sublevels existing in the HTML DOM tree or how far elements are nested in the DOM tree. Older versions of these web applications were obtained using Wayback Machine [18] and SourceForge [19]. The versions used are mentioned in *"Table II"*.

In order to test the proposed algorithm on the mentioned applications, these steps were followed. Firstly, an element is selected from the old version of the AUT using the absolute XPath locator for that element. After that, the locator is tried on the new version. If the locator could successfully obtain a unique element, return that element. If no unique element could be found, the GA is called. The GA runs for the specified number of iterations and finally returns the fittest element found. For each test case, the proposed GA was executed 20 times. Each time, the returned element was recorded. The number of times the correct element has been returned was calculated and the average was taken.

It should also be noted that when choosing elements to make our tests, we didn't consider repeated ones; for example, if the page contains a list of clickable elements, only choose

Table I. A COMPARISON BETWEEN THE APPLICATONS USED TO TEST THE PROPOSED ALGORITHM

Application / Property	No. of elements per page	HTML structure	No. of attributes per element
Claroline	Few	Simple	Moderate
Wikipedia	Moderate	Complex	Moderate
Goodreads	Many	Simple	Moderate
Amazon	Many	Simple	Many
Fox news	Moderate	Complex	Few

Table II. DATES OF THE OLD AND NEW VERSIONS OBTAINED FOR EACH APPLICATION

	The old version	The new version
Claroline	v.10 29/12/2010	v.11 11/6/2012
Wikipedia	1/1/2014	30/12/2014
Goodreads	1/1/2014	30/12/2014
Amazon	1/1/2014	30/12/2014
Fox news	1/1/2014	30/12/2014

one of them is chosen, or if there is a calendar object in the page, only one day is chosen and so on.

B. Results

"Table III" shows the results. For each application, the number of pages considered, the average number of elements per page, the number of failing locators selected for test, the number of failing locators after applying the proposed algorithm and the repair percentage are recorded.

Table III. Results of applying the PROPOSED algorithm on the mentioned applications.

	No. of pages	**Average no. of elements per page**	**No. of failing locators recorded**	**No. of failing locators after using the GA**	**Repair percent**
Claroline	50	258	623	43	**93%**
Wikipedia	3	1530	66	5	**92.4%**
Goodreads	2	2090	61	16	73.8%
Amazon	2	2200	50	9	82%
Fox news	6	1420	118	49	58.4%
Total	63	7498	918	122	86.7%

Results indicate an obvious improvement in the number of locators that fail after changes have been done in the GUI. From the results, it could be shown that applications with more attributes per elements and fewer elements per page have higher repairing percentage. The complexity of the HTML structure and how nested the elements are in the DOM tree have no effect on the efficiency of the algorithm. The lowest repair percentage was on the Fox News website. This is because when the GA was applied to fix the locators, the "text" attribute which is an important attribute and has a high weight in the fitness of the element, was causing confusion to the algorithm. This confusion was due to the fact that the content of this website was changing daily. The best repair percentage was in Claroline application due to its significant small number of elements per page.

V. CONCLUSION

In this paper, an algorithm was proposed to repair GUI test cases for web applications that failed due to broken locators. In the suggested algorithm, the problem was formulated as an optimization problem and the Genetic Algorithm was adopted to solve it. Experiments were done on more than 900 test case from several applications. The proposed algorithm could repair about 87% of the locators. This result is a significant improvement compared to the result obtained by [13] who could repair only 73% of the tested locators.

REFERENCES

[1] S. Berner, R. Weber, and R. K. Keller, "Observations and lessons learned from automated testing," in Proceedings of the 27th international conference on Software engineering - ICSE '05, 2005, p. 571.

[2] D. M. Rafi, K. R. K. Moses, K. Petersen, and M. V. Mantyla, "Benefits and limitations of automated software testing: Systematic literature review and practitioner survey," 2012 7th Int. Work. Autom. Softw. Test, pp. 36–42, Jun. 2012.

[3] K. Karhu, T. Repo, O. Taipale, and K. Smolander, "Empirical observations on software testing automation," in Proceedings - 2nd International Conference on Software Testing, Verification, and Validation, ICST 2009, 2009, pp. 201–209.

[4] A. M. Memon and M. Lou Soffa, "Regression testing of GUIs," Proc. 9th Eur. Softw. Eng. Conf. held jointly with 10th ACM SIGSOFT Int. Symp. Found. Softw. Eng. - ESEC/FSE '03, vol. 28, no. 5, p. 118, Sep. 2003.

[5] M. Grechanik, Q. Xie, and C. Fu, "Maintaining and evolving GUI-directed test scripts," in Proceedings - International Conference on Software Engineering, 2009, pp. 408–418.

[6] S. R. Choudhary, D. Zhao, H. Versee, and A. Orso, "WATER : Web Application TEst Repair," First Int. Work. EndtoEnd Test Scr. Eng., pp. 24–29, 2011.

[7] L. Christophe, R. Stevens, C. De Roover, and W. De Meuter, "Prevalence and maintenance of automated functional tests for web applications," in Proceedings - 30th International Conference on Software Maintenance and Evolution, ICSME 2014, 2014, pp. 141–150.

[8] M. Leotta, A. Stocco, F. Ricca, and P. Tonella, "Using multi-locators to increase the robustness of web test cases," in 2015 IEEE 8th International Conference on Software Testing, Verification and Validation, ICST 2015 - Proceedings, 2015, pp. 1–10.

[9] K. F. Man, K. S. Tang, and S. Kwong, "Genetic algorithms: Concepts and applications," IEEE Trans. Ind. Electron., vol. 43, no. 5, pp. 519–534, 1996.

[10] M. Leotta, D. Clerissi, F. Ricca, and P. Tonella, "Visual vs. DOM-based web locators: An empirical study," Lect. Notes Comput. Sci. (including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics), vol. 8541, pp. 322–340, 2014.

[11] "W3C Document Object Model." [Online]. Available: https://www.w3.org/DOM/. [Accessed: 25 Aug. 2018].

[12] "XML Path Language (XPath) 3.1." [Online]. Available: https://www.w3.org/TR/xpath-31/. [Accessed: 25 Aug. 2018].

[13] I. Aldalur and O. Diaz, "Addressing web locator fragility," in Proceedings of the ACM SIGCHI Symposium on Engineering Interactive Computing Systems - EICS '17, 2017, pp. 45–50.

[14] S. Thummalapenta, P. Devaki, S. Senha, S. Chandra, S. Gnanasundaram, D. D. Nagaraj, S. Kumar, and S. Kumar, "Efficient and change-resilient test automation: An industrial case study," in Proceedings - International Conference on Software Engineering, 2013, pp. 1002–1011.

[15] M. Leotta, D. Clerissi, F. Ricca, and C. Spadaro, "Comparing the maintainability of selenium WebDriver test suites employing different locators: a case study," Proceedings of1st International Workshop on Joining AcadeMiA and Industry Contributions to testing Automation, no. Jamaica, pp. 53–58, 2013.

[16] M. Leotta, A. Stocco, F. Ricca, and P. Tonella, "Robula+: An algorithm for generating robust XPath locators for web testing," J. Softw. Evol. Process, vol. 28, no. 3, pp. 177–204, Mar. 2016.

[17] M. Leotta, D. Clerissi, F. Ricca, and P. Tonella, "Capture-replay vs. programmable web testing: An empirical assessment during test case evolution," in 2013 20th Working Conference on Reverse Engineering (WCRE), 2013, pp. 272–281.

[18] "Wayback Machine." [Online]. Available: web.archive.org. [Accessed: 25 Aug. 2018].

[19] "SourceForge - Download, Develop and Publish Free Open Source Software." [Online]. Available: https://sourceforge.net/. [Accessed: 27 Aug. 2018].