Using Multi-Locators to Increase the Robustness of Web Test Cases

Maurizio Leotta[1], Andrea Stocco[1], Filippo Ricca[1], Paolo Tonella[2]
[1] Dipartimento di Informatica, Bioingegneria, Robotica e Ingegneria dei Sistemi (DIBRIS), Università di Genova, Italy
[2] Fondazione Bruno Kessler, Trento, Italy
maurizio.leotta@unige.it, andrea.stocco@dibris.unige.it, filippo.ricca@unige.it, tonella@fbk.eu

***Abstract*—The main reason for the fragility of web test cases is the inability of web element locators to work correctly when the web page DOM evolves. Web elements locators are used in web test cases to identify all the GUI objects to operate upon and eventually to retrieve web page content that is compared against some oracle in order to decide whether the test case has passed or not. Hence, web element locators play an extremely important role in web testing and when a web element locator gets broken developers have to spend substantial time and effort to repair it.**

While algorithms exist to produce robust web element locators to be used in web test scripts, no algorithm is perfect and different algorithms are exposed to different fragilities when the software evolves. Based on such observation, we propose a new type of locator, named *multi-locator*, which selects the best locator among a candidate set of locators produced by different algorithms. Such selection is based on a voting procedure that assigns different voting weights to different locator generation algorithms. Experimental results obtained on six web applications, for which a subsequent release was available, show that the multi-locator is more robust than the single locators (about –30% of broken locators w.r.t. the most robust kind of single locator) and that the execution overhead required by the multiple queries done with different locators is negligible (2-3% at most).

***Keywords*—Web Testing, Testware Evolution, Test Case Robustness, Web Element Locators, XPath Locators.**

I. Introduction

The cost of software testing is impressively high and estimated between 40% and 80% of the total development cost [1]. Test automation plays a key role in reducing such cost [5]. Software testers implement the testing logics by writing scripts that provide input data, set the values of GUI components, operate on such components by changing their state and retrieve information to be compared with oracles, to determine if the program behaves correctly. Automated testing tools, that run such scripts, interact with the application in a similar way as the real user does. Reuse of testing logics for the same functionalities across successive releases (i.e., for regression purposes) is the main benefit of adopting a test automation technique. Unfortunately, new releases of the application with modified GUIs can easily break the corresponding test scripts, hindering the benefits of test automation [2]. This problem is particularly dramatic in the context of the web applications, because these are subject to a tremendous pressure for change [20]. New releases are continuously produced, often accommodating just style improvements or presentation changes.

Typically, DOM-based web test automation tools (e.g., Selenium toolkit) are used to test a web application. Thanks to their rich APIs, testers can easily implement the test cases by invoking commands that operate on web elements localised by means of DOM properties (attributes, textual information, XPaths, etc). The choice of appropriate web element *locators* is fundamental, because it impacts enormously the test script resilience to change (i.e., their *robustness*) when the application evolves. Previous works [8], [9] show that often minor changes between releases, resulting in changes in the DOM structure, are responsible for most of the cases when test cases are broken and cannot be executed any more. The manual effort to repair such test scripts is tedious, time-consuming and intellectually frustrating, so that often existing test suites are abandoned, despite their potential value for catching regressions.

For this reason, in the literature several researchers [11], [16] have attacked the test script fragility problem by proposing algorithms able to compute locators that are resilient to the evolution of the software. These *robust locators*, based on the XPath language, have been shown to be more resilient to changes than those available from state of the practice tools, such as for example FirePath [11]. When web pages change because of a new release of the web application, they continue to select the target web element correctly.

While experimenting with different locator generation algorithms, we have noticed that locators are resilient to different types of changes and that they tend to be fragile individually, not collectively. Even locators produced by algorithms with the highest robustness performance can occasionally be broken (i.e., they do not locate the target web element correctly in the new DOM) by specific code changes, while other locators, based on different web page properties, may remain valid. In other words, different locators, built by different algorithms, tend to be complementary with each other. The idea of this work is to compensate for one locator's fragility by resorting to the capabilities of another locator.

In this paper, we overcome the potential weaknesses of a single locator by means of a novel locator type, which we call **multi-locator** and which is capable of aggregating the results produced by a set of different locators (generated by different algorithms) into a single web element localisation, the most voted one. In the test cases, we replace each single locator with a multi-locator, i.e., a set of locators all selecting the same web element and all automatically generated by different algorithms. When the web application evolves, some locators in this set become broken while others may work correctly and return the right element. By applying a voting decision procedure, the multi-locator will select the web element receiving the highest number of weighted votes from all locators. We expect that the multi-locator will be more robust than each individual locator taken in isolation. On the other hand, the advantage of adopting the multi-locator is also in its *automatic repair* capability: when the multi-locator is able to locate the desired web element, the broken locators belonging to the set and generated by the various algorithms can be automatically repaired. This accounts just for re-running the locator creation algorithms with the web element returned by the multi-locator as target. In this way, the test scripts are continuously evolved by the automated repair procedure, hence better accommodating the future changes occurring in the next software releases.

The paper is organised as follows: Section II introduces the problems associated with web testware evolution and shows how locators are generated by state of the art algorithms. Section III describes the multi-locator, our novel contribution. Empirical results about the robustness and execution time overhead of the multi-locator as compared to single locators are reported in Section IV, followed by related works (Section V) and conclusions (Section VI).

II. Background

Software testers are required to execute manual repair actions on the test cases, whenever these are affected by the changes that have been performed on the web application under test (WAUT). For the sake of simplicity, the changes to the WAUT can be categorised into two families: logical and structural [9], [10]. A logical change involves the modification of the web application logics for the introduction of new features or the modification of existing features. On the tester side, this means for example creating new test cases, or modifying the existing ones. A structural change, instead, impacts the web page structure, modified to beautify the web page appearance or to reorganise its content (e.g., switching from a table-based to a table-less layout). In the test suite, the tester has to modify one or more test script lines containing locators that are affected by the structural changes.

In this paper, we focus on reducing the web test suite maintenance effort due to structural changes, since such effort is heavily affected by the fragility of the locators. On the other hand, logical changes require manual interventions on the test suite that go beyond the creation of robust locators. Structural changes are indeed quite important, since web site re-styling, a frequently occurring activity, tends to affect the DOM structure, leaving the application logics unaffected. In the following, we assume that XPath locators are used to retrieve the web elements required by the test cases (form fields, buttons, check boxes, textual output, etc.) and that the tester's effort can be reduced when structural changes occur, by making such locators robust.

A. XPath Locators

Considering only XPath locators is by no means restrictive, for the following reasons:

– *XPath is a powerful and expressive language.* If properly generated, XPath locators can be highly expressive and compact. To the best of our knowledge, most of the localisation methods provided by DOM-based tools can be easily rewritten as an XPath locator with no substantial impact on its understandability. For example, the Selenium WebDriver locator: By.name("xy") is equivalent to By.xpath("//*[@name='xy']").

– *XPath locators are sometimes the only option.* Selenium WebDriver offers different localisation methods[1] beyond XPath, mostly tailored on specific DOM attributes and properties (e.g., id or text). A tester may prefer the use of these methods, instead of generating/writing XPath expressions that may be perceived as more complex. However, sometimes these methods cannot be employed, since no unique attribute value or textual information are available to uniquely identify the web element of interest. In such cases, the only way to get a locator is by specifying a navigational path on the DOM tree. As an example, in our previous work [9], we considered six Selenium WebDriver test suites and we were forced to use XPath locators (i.e., no specific localisation methods can be used) for about one-third of the considered web elements.

B. Algorithms/Tools for Generating XPath Locators

We populate our multi-locator with the most used and most promising XPath locators (to the best of our knowledge), generated by state of the practice tools and by state of the art research algorithms. In particular we considered:

FirePath Absolute: FirePath[2] is a browser-integrated plugin for XPath expressions generation. For each web element, it is able to generate a corresponding absolute XPath locator. An absolute XPath consists of the full navigational path from the root of the DOM (i.e., the html tag) to the target web element. Only when strictly necessary, element position values are used to select the correct node among a set of siblings.

FirePath Relative ID-based: When a unique value for the id attribute exists for the target element or one of its ancestors, FirePath can also generate a relative ID-based XPath locator. Otherwise an absolute XPath is returned. In case a unique value for the id attribute exists, the XPath locator starts by selecting the node (closest to the target) that contains id and then navigates the remaining portion of the DOM to the target element.

Selenium IDE[3] is a capture/replay tool for quick development of web test cases. During the test case recording phase, it is able to generate locators for the web page elements on which the tester is performing actions. Selenium IDE contains an advanced XPath locators generator algorithm[4] that generates locators using different strategies and that ranks them depending on an internal robustness heuristic estimate.

Montoto et al. [16] proposed an algorithm for identifying the target elements during the navigation of AJAX websites. The algorithm starts from a simple XPath expression, progressively augmented with textual and attribute information. The algorithm first tries to identify the element according to its associated text (if the element is a leaf node) and the value of its attributes. If the XPath produced does not uniquely identify the element, every ancestor (and the value of their attributes) is considered until the root of the DOM is reached.

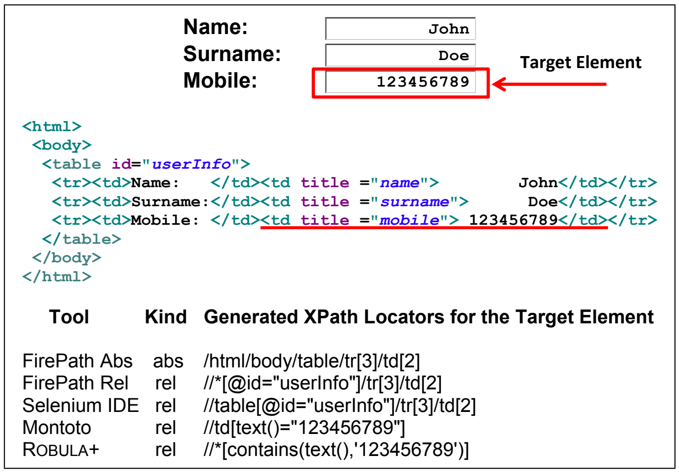
Robula+ is an extension of our previous algorithm Robula (ROBUst Locator Algorithm) [11]. Basically, Robula starts with a generic XPath expression that returns all nodes ("//*"). It then iteratively refines the expression until only the element of interest is selected. In such iterative refinement, the algorithm applies four refinement transformations, according to a set of heuristic XPath specialisation steps [11]. Robula has been developed in order to create very short and simple XPath

[1] http://docs.seleniumhq.org/docs/02_selenium_ide.jsp#locating-elements

[2] https://addons.mozilla.org/firefox/addon/firepath/

[3] http://seleniumhq.org/projects/ide/

[4] https://code.google.com/p/selenium/source/browse/ide/main/src/content/locatorBuilders.js



Name: John
Surname: Doe
Mobile: 123456789

```
<html>
 <body>
  <table id="userInfo">
   <tr><td>Name:   </td><td title ="name">       John</td></tr>
   <tr><td>Surname:</td><td title ="surname">      Doe</td></tr>
   <tr><td>Mobile: </td><td title ="mobile"> 123456789</td></tr>
  </table>
 </body>
</html>
```

Tool	Kind	Generated XPath Locators for the Target Element
FirePath Abs	abs	/html/body/table/tr[3]/td[2]
FirePath Rel	rel	//*[@id="userInfo"]/tr[3]/td[2]
Selenium IDE	rel	//table[@id="userInfo"]/tr[3]/td[2]
Montoto	rel	//td[text()="123456789"]
Robula+	rel	//*[contains(text(),'123456789')]

Fig. 1. showInfo.php – Ver. 1 – Page, Source, Locators

expressions, with the goal to increase their resilience to changes. Robula+ enhances Robula with: *(i)* a prioritisation strategy, to rank candidate XPath expressions by heuristically estimated attribute robustness, when multiple attributes are available; *(ii)* a blacklisting technique, to exclude attributes that are intrinsically fragile; and *(iii)* textual information, potentially a reliable anchor when the web application evolves [19]. A technical report describing Robula+ is available on our web site: http://sepl.dibris.unige.it/TR/Robula+.pdf.

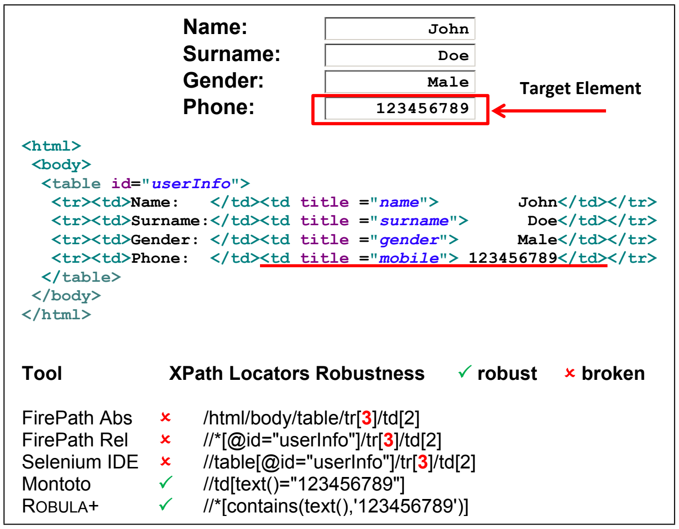
The outputs of the five algorithms considered in our work are usually different, as depicted in the example in Fig. 1. Focusing only on the two algorithms proposed by the research community, Montoto and Robula+, they both adopt a top-down approach in the construction of the XPaths. However, remarkable differences exist thus, the XPath expressions generated by Robula+ are usually very different from the ones generated by Montoto. For instance, to localise the target div element in the web page used as example in the Montoto et al. paper [16], their algorithm generates //td/a[@href="#"]/div[@class="c1" and text()="More Info"] while Robula+ generates the following simpler XPath expression //td/*/div.

C. XPath Locators and Software Evolution: an Example

Let us consider Ver. 1 of a simplified web application composed of two web pages — insertInfo.php and showInfo.php — that allow users to insert and visualise some personal information previously stored in a database. A test case for this functionality may open the insertInfo.php page, fill a form, submit the information and verify that the inserted data are correctly displayed in the resulting showInfo.php page, shown in Fig. 1 (top).

For the test case implementation, it is necessary to locate some web page elements as, for instance, the field of the table showing the mobile phone number (see the underlined td in Fig. 1 (center)). Fig. 1 (bottom) lists the XPath locators provided by the algorithms considered in this work. With the exception of the absolute (abs) XPath locator generated by FirePath, the others are relative XPaths. Different XPath generation strategies are adopted resulting in different expressions.

We now consider a new version of the web application (Ver. 2), in which a new text box is present, allowing the user to insert gender information (see Fig. 2 (top)). Depending on the robustness of the XPath locator used to select the target element, the test case described above will be broken (and will have to be repaired) or will work without problems. Looking at Fig. 2 (bottom), we can see that only the locators generated by Robula+ and Montoto work, while all the other locators are broken. Indeed, all of them include node tr[3] that in the new version becomes tr[4]. Hence, they locate the wrong element (i.e., the "gender" field).



Name: John
Surname: Doe
Gender: Male
Phone: 123456789

```
<html>
 <body>
  <table id="userInfo">
   <tr><td>Name:   </td><td title ="name">        John</td></tr>
   <tr><td>Surname:</td><td title ="surname">       Doe</td></tr>
   <tr><td>Gender: </td><td title ="gender">      Male</td></tr>
   <tr><td>Phone:  </td><td title ="mobile"> 123456789</td></tr>
  </table>
 </body>
</html>
```

Tool		XPath Locators Robustness ✓ robust × broken
FirePath Abs	×	/html/body/table/tr[3]/td[2]
FirePath Rel	×	//*[@id="userInfo"]/tr[3]/td[2]
Selenium IDE	×	//table[@id="userInfo"]/tr[3]/td[2]
Montoto	✓	//td[text()="123456789"]
Robula+	✓	//*[contains(text(),'123456789')]

Fig. 2. showInfo.php – Ver. 2 – Page, Source, Locators

III. The Multi-Locator Approach

In this section we describe the multi-locator approach, which selects a web element using a candidate set of locators performing a vote decision procedure (weighted or unweighted). We also show that the multi-locator can be employed to automatically repair the broken locators in the candidate set, so as produce a better candidate set of locators, to be used by the multi-locator on the successive versions of the application.

A. Multi-locator Definition

Let us assume that a candidate set L (with $|L| > 1$) of alternative locators can be obtained to extract the web element e from the DOM D. Such alternative locators can be generated in different ways: by alternative algorithms or tools that help web testers to produce robust locators for their test cases, manually, or they can be defined according to simple rules (e.g., use the absolute XPath or the web element identifier/name). When they are initially defined, all such locators select element e uniquely:

$$\forall l \in L : \text{query}(l, D) = \{e\} \qquad (1)$$

i.e., all XPath queries using such locators return a result set containing exactly one entity, element e.

When the web application evolves, some locators in L may become unusable because they return more than one element or no element in the new DOM D'. Among those that return a single web element (i.e., $|query(l, D')| = 1$), there might be disagreement. The idea of the multi-locator is to establish a voting procedure that involves all locators still returning exactly one element from the new DOM D'. The multi-locator will select the web element receiving the highest weighted vote from all locators that uniquely select a single element in the DOM D'. Since different locators may have different "reputations" (e.g., the absolute XPath locator is known to be quite fragile [8], [11]), it makes sense to assign different weights to the voters. The web element returned by the multi-locator will be the one with the highest weighted vote. We decided to compute the aggregate vote v for the element e' using the following formula:

$$v[e'] = 1 - \Pi_{l \in L_{e'}}(1 - \text{nw}[l]) \quad (2)$$

where $L_{e'}$ is the set of locators returning uniquely the element e' in the new DOM D', while nw$[l]$ is the normalised weight assigned to locator l. Weights are normalized between 0 and 1, and each of them is interpreted as the *locator reliability*, i.e., probability of correct localisation. A reliable locator, generated by an algorithm with high robustness performance, will have a high weight (i.e., close to 1). Such interpretation justifies formula (2): the result of the formula gives the probability that the aggregate vote localises correctly element e'.

Algorithm 1: Multi-locator DOM Selection

```
Input:
$D'$: DOM of the evolved web application.
$L$: set of locators selecting uniquely web element $e$ in the initial
    DOM $D$. The information about the algorithms that generated each
    locator $l \in L$ is stored in an auxiliary datastructure
Result:
$e'$: a web element from DOM $D'$ or null if no web element can be
    located
1  begin
2      $L'_c := \{l \in L : |\text{query}(l, D')| = 1\}$
3          // candidate locators
4      $E' := \{e' \in D' : e' \in \text{query}(l', D'), l' \in L'_c\}$
5          // candidate target web elements
6      if $|E'| = 0$ then return null
7
8      foreach $e' \in E'$ do $v[e'] := 1$
9
10     foreach $l' \in L'_c$ do
11         $e' := \text{elementOf}(\text{query}(l', D'))$
12             // elementOf returns the unique element of the set
13         $v[e'] := v[e'] \cdot (1 - \text{nw}(l'))$
14             // nw returns the weight, normalised between 0 and 1,
                relative to the algorithm used to generate $l'$
15     foreach $e' \in E'$ do $v[e'] := 1 - v[e']$
16
17     return $e' \in E' : v[e'] = \max_{k \in E'} v[k]$
```

The pseudocode for the multi-locator procedure is shown in Algorithm 1. At lines 2-4, candidate web elements are determined as the results of all XPath queries that return one element. Formula (2) is implemented from line 8 to line 15. In particular, the loop at line 10 attributes the weight of each locator l' to the selected web element e' according to formula (2). At line 17, the multi-locator returns the web element that was attributed the highest vote. In case of parity, a randomly selected element among those with highest weight is chosen.

An important component of Algorithm 1 is the voting weight assigned to each locator (loop at line 10). We suggest three strategies to determine such weights: (1) uniform weights; (2) learned weights; (3) heuristic weights. *Uniform weights* are obtained by trivially assigning the same weight (e.g., 0.5) to each method used to generate the locators in L, hence, to each locator. *Learned weights* are obtained by training them on a corpus of web applications for which successive versions are available. Weights can be optimised so as to minimise the number of broken locators that is measured when the multi-locator algorithm is applied to the next versions of the web applications in the corpus. Otherwise, a simpler method consists of measuring the robustness of the locators (number of non-broken locators) on the corpus and using such measurement as the weight for the algorithm that generated such locators. We implemented the latter method. As usual with training, the training corpus must be different from the web applications on which the multi-locator performance is assessed. This can be achieved, for instance, through the cross-validation (aka, leave-one-out) procedure. *Heuristic weights* are produced manually, based on a-priori knowledge about the expected fragility of the locators created by the various locator generation methods [11].

In Fig. 1 (bottom), there are five different XPath locators generated using different algorithms. When evaluated on the new version of the web page (see Fig. 2 (bottom)) three of them are broken, while two select the correct target element (i.e., the phone number field). It should be noticed that in the new version of the web page all three broken locators select the same web element, i.e., the gender field. Thus, in this case, if the unweighted (i.e., uniformly weighted, with weight=0.5 for all the considered algorithms) version of the multi-locator is adopted, the result will select the wrong element. In fact, the gender field obtains three votes (corresponding to v[gender]=0.875, see equation (2)) while the phone number field only two (corresponding to v[phone]=0.75). On the other hand, using a weighted version of the multi-locator, depending on the weights assigned, it is possible to select the correct target element. For instance, we can assign the following weights based on a-priori knowledge about the expected robustness of the locators created by the various locator generation methods: (1) for absolute XPaths, weight is 0.25, since they are known to be quite fragile; (2) for locators obtained by tools for ID-based DOM navigation, weight is 0.50, since they are probably more robust than absolute XPaths, but they are not designed specifically to make test cases resilient to the evolution of the web application (e.g., FirePath Relative ID-based); (3) for locators obtained by algorithms specifically designed for producing robust locators for web testing, weight is 0.90 (e.g., Selenium IDE, Montoto and ROBULA+). Using these weights the result is different. Indeed, the gender field obtains the lowest weighted vote (corresponding to v[gender]=0.9625) while the phone number field the highest (corresponding to v[phone]=0.99). Thus, this version of the weighted multi-locator adopting "knowledge based" weights is able to select the correct element.

When the web application evolves, there is a *theoretical limit* to the capabilities of *any* multi-locator, defined as an arbitrary procedure to select among the XPaths of the set L of locators. In fact, if all locators in L are broken when used to query the evolved DOM D', the multi-locator has no chance of being able to select the right element, since any selection of a locator $l \in L$ will result in a broken locator. This sets an upper bound to the robustness achievable by any multi-locator.

B. Automated Multi-locator Repair

When the multi-locator procedure succeeds, by selecting the most voted locator and returning the web element identified by such locator, it is possible to automatically repair all other broken locators. In fact, the algorithms that have been used to produce the locator set L for the initial DOM D can be re-executed on the new DOM D' to locate the web element e' returned by the multi-locator. Each algorithm whose locator is considered broken on D' by the multi-locator is re-executed with e' (e.g., the element selected by the multi-locator) as target. The new locators that these algorithms produce replace the locators considered broken. When the web application evolves to a new version, the multi-locator will be able to use the automatically repaired locators instead of the original ones, hence further increasing its chances of robustly identifying the web element used by the test cases.

Algorithm 2: Multi-locator Repair

```
Input:
$D'$: DOM of the evolved web application.
$e'$: the web element in the evolved DOM $D'$ selected by multi-locator.
$L$: set of locators selecting uniquely web element $e$ in the initial
    DOM $D$. The information about the algorithm that generated each
    locator $l \in L$ is stored in an auxiliary datastructure.
Result:
$L'$: the repaired set of locators selecting web element $e'$ from $D'$
1  begin
2      $L' := \emptyset$
3      foreach $l \in L$ do
4          if $query(l, D') = \{e'\}$ then
5              $L' := L' \cup \{l\}$
6          else
7              $nl$ := createNewLocator($algoForLocator(l)$, $D'$, $e'$)
8                  // algoForLocator uses information from the
                      auxiliary datastructure associated with $L$
9              $L' := L' \cup \{nl\}$
10     return $L'$
```

Algorithm 2 shows the pseudo-code of the automated repair method. The input web element e' is uniquely identified by the Algorithm 1. The repaired set of locators L' is constructed iteratively at lines 3-9, by just keeping unmodified the locators that uniquely identify e' in the new DOM D' (line 5), and by creating a new locator nl (lines 7-9) in the other cases. To generate such new locators, the same algorithm originally used to produce the locator l is run (such algorithm is returned by function *algoForLocator*(l) on the new DOM D', with web element e' as target).

For example, in the case of the XPath locators shown in Fig. 2 (bottom), using the weighted version of the multi-locator described above (i.e., using weights 0.25, 0.50 and 0.90), we obtain that the phone number field has the highest vote. Thus, the three locators considered (in this case correctly) broken (i.e., FirePath Absolute, FirePath Relative ID-based and Selenium IDE) are re-generated in order to select the phone number field. The related repair operation affects only the locator fragment tr[3], which in the new release becomes tr[4].

Of course, in case the web element e' returned by the multi-locator is wrong, the automated repair operation will also produce a set of *incorrectly repaired locators*. Hence, the possibility to perform correct repair operations is strictly related to the correctness of the multi-locator, when it returns a non-null element. One of the research questions investigated in our empirical evaluation deals with the correctness of the automated repair actions.

C. Setting up the Multi-locator

For setting-up the multi-locator it is necessary to execute two steps: **(1)** generating the set L (i.e., a list of XPath locators) for each target element employed in the original test suite (i.e., located by a single locator) and **(2)** defining the set of weights in case the weighted multi-locator is applied. Step **(1)** can be completed by resorting to aspect oriented programming: during the execution of the test suite an aspect will intercept each locator invocation on the current page DOM D, that selects the target element e, and will generate the corresponding set L by using the XPath generation algorithms implementations (i.e., L={FirePathAbs(D, e), FirePathID(D, e), Montoto(D, e), SeleniumIDE(D, e), RobulaPlus(D, e)}). Step **(2)** can be completed by estimating the robustness of the various kinds of XPath locators on the considered web application. Alternatively, heuristic weights can be used. Indeed, in Section IV-E, we show that for a practical adoption of the multi-locator it is not even necessary to execute the cross-validation procedure to obtain the weights.

D. Deploying the Multi-locator

For deploying the multi-locator the test code has to be changed so as to replace single locator invocations on the DOM D (e.g., By.xpath(xp,D)), with invocations to the multi-locator, which requires a set of XPaths L={xp1,...,xp5} instead of a single XPath (e.g., By.multiLocator({xp1,...,xp5}, D)) and a list of weights in case non-uniform weights are to be applied (e.g., By.multiLocator({xp1,...,xp5}, {w1,...,w5}, D)). This can be done automatically by means of code transformations. In this way, the multi-locator can be used to locate the target element in the evolved DOM D' using the previously computed set L. We are currently developing a tool able to automatically migrate existing test cases to the multi-locator approach.

E. Evolving Multi-locator based Test Cases

When a new version of the web application is available, in case the multi-locator (Algorithm 1) is able to correctly select the target web element e', the set of alternative locators L is automatically repaired (Algorithm 2) by invoking the XPath generation algorithms with the web element selected by the multi-locator and the evolved DOM D' as input (e.g., if the multi-locator selects e' on D', the other locators that are not able to select e' are updated as follows: xp3 = Montoto(D', e'), xp5 = RobulaPlus(D', e')). Otherwise, the multi-locator is unable to select the target element, thus the test case is broken, and the web tester has to manually repair *only one* of the locators; the other locators can be automatically re-generated using the implementations of the various XPath generation algorithms. Finally, the test suite can be automatically updated with the new generated locators set L' similarly as described in Section III-D.

IV. Experimental Results

This section presents the design, objects, research questions, metrics, procedure, quantitative and qualitative analysis, and threats to validity of the empirical study conducted to evaluate the effectiveness and execution time overhead of the multi-locator. We follow the guidelines by Wohlin *et al.* [22] on designing and reporting empirical studies in software engineering.

The *goal* of this study is to analyse the effectiveness and performance of the multi-locator in selecting the correct target element, with the purpose of understanding the strengths and the

TABLE I. Objects: Web Applications from *SourceForge.net*

	Description	Web Site	1st Release				2nd Release			
			Release	Date	File[a]	kLOC[b]	Release	Date	File[a]	kLOC[b]
MantisBT	bug tracking system	http://sourceforge.net/projects/mantisbt/	1.1.8	Jun-09	492	90	1.2.0	Feb-10	733	115
PPMA[c]	password manager	http://sourceforge.net/projects/ppma/	0.2	Mar-11	93	4	0.3.5.1	Jan-13	108	5
Claroline	collaborative learning environment	http://sourceforge.net/projects/claroline/	1.10.7	Dec-11	840	277	1.11.5	Feb-13	835	285
Address Book	address/phone book, contact manager, organizer	http://sourceforge.net/projects/php-addressbook/	4.0	Jun-09	46	4	8.2.5	Nov-12	239	30
MRBS	system for multi-site booking of meeting rooms	http://sourceforge.net/projects/mrbs/	1.2.6.1	Jan-08	63	9	1.4.9	Oct-12	128	27
Collabtive	collaboration software	http://sourceforge.net/projects/collabtive/	0.65	Aug-10	148	68	1.0	Mar-13	151	73

[a] Only PHP source files were considered - [b] PHP LOC - Comment and Blank lines are not considered - [c] Without considering the source code of the framework used by this application (Yii framework)

weaknesses of the proposed approach. The results of this study are interpreted according to the *perspective* of: (1) developers and project managers, interested in data about the benefits of adopting the multi-locator in an industrial context, in order to increase the test suite robustness; (2) *researchers*, interested in empirical data about the impact of the multi-locator on web testing. The *software objects* used in the experiment are six web applications already used in a different work [10].

A. Web Applications

We conducted our experiments over a sample of six open-source web applications from *SourceForge.net*. We considered only applications that: (1) are quite recent, so that they can work without problems on the latest versions of Apache, PHP and MySQL, technologies we are familiar with (since the XPath locators localise web elements in the HTML code processed by the client browser, the server side technologies do not affect the results of the study); (2) are well-known and used (some of them have been downloaded more than one hundred thousand times last year); (3) have at least two major releases (we have excluded minor releases because with small differences between versions the majority of the locators — and, thus, of the corresponding test cases — are expected to work without problems); (4) belong to different application domains.

Table I reports some information about the selected applications. We can notice how all of them are quite recent (ranging from 2009 to 2013) and different in terms of number of source files (ranging from 46 to 840) and number of lines of code (ranging from 4 kLOC to 285 kLOC, considering only the lines of code contained in the PHP source files, comments and blank lines excluded).

B. Research Question and Metrics

Our study aims at answering the following research questions:

RQ1: *What is the robustness of the unweighted multi-locator as compared to that of single locators? What is the effect of the random selection occurring when different elements obtain the same number of votes?*

The goal of the first research question is to compare the robustness of the unweighted multi-locator with the robustness of five single locators, generated by state of the practice tools (FirePath, release 0.9.7; Selenium IDE, release 2.8.0) and by research algorithms (Montoto, ROBULA+). The aim is to give developers and project managers a precise idea of the benefits coming from the adoption of the multi-locator. The metrics used to answer RQ1 is the number of broken XPath locators in the next software release.

RQ2: *What is the robustness of the weighed multi-locator as compared with the unweighted multi-locator?*

The second research question is about the influence of the weights used in the multi-locator. In particular, with this research question we want to compare unweighted and weighted multi-locators. The aim is to give developers, project managers and researchers an idea of the importance of a good calibration of the used weights. The metrics used to answer RQ2 is the same used for RQ1.

RQ3: *How far is the robustness of the multi-locator from the theoretical limit?*

The third research question aims at understanding whether the multi-locator strategy has any margin of further improvement. This aims at giving researchers an idea about the contributions possibly coming from more complex multi-locator strategies. The metrics used to answer RQ3 is the distance between our algorithm and the theoretical limit explained in the previous section.

RQ4: *What is the amount of correct repair actions triggered by the multi-locator?*

The fourth research question deals with the correctness of the automated repair actions operated on the broken locators. In particular, we are interested in the total number of correctly repaired broken locators w.r.t. the incorrectly repaired ones. The metrics used to answer RQ4 is the number of correctly repaired locators over the total number of repair actions.

RQ5: *What is the performance overhead of the multi-locator on test case execution?*

The last research question is about the additional time required for executing a test suite when the multi-locator selection (i.e., Algorithm 1) is adopted, as experienced in practical cases. This gives developers and project managers an idea of the penalty in terms of execution time coming from the adoption of the multi-locator. The metrics used to answer RQ5 is the execution time expressed in seconds.

C. Procedure

To answer our RQs we proceeded as follows:

(I) We selected six open-source web applications from *SourceForge.net* as explained in Section IV-A.

(II) For each application and for each web page we manually selected all the web elements: (1) on which it is possible to perform actions (e.g., links, input fields, submit buttons); (2) which report information that can be used to evaluate assertions (e.g., the number of rows in a table or a confirmation message); (3) which belong to pages related to core functionalities of the application (e.g., we did not consider the configuration and installation pages); and, (4) which are present in both releases of the applications. This last requirement is particularly important for computing the number of broken locators.

In order to avoid biased results, we excluded multiple instances of the same web element present in different pages, or different web elements that can be considered the same. In

TABLE II. Robustness of the various XPath Locators and of different kinds of Multi-Locator Algorithms

	Address Book			Collabtive			MRBS			Claroline			PPMA			Mantis			All Apps	
Total Number of Target Web Elements	**80**			**125**			**102**			**235**			**30**			**103**			**675**	
Locators	**Broken**	**%**	**Weight**	**Broken**	**%**	**Weight**	**Broken**	**%**	**Weight**	**Broken**	**%**	**Weight**	**Broken**	**%**	**Weight**	**Broken**	**%**	**Weight**	**Broken**	**%**
FirePath Absolute	45	56	0,32	125	100	0,41	102	100	0,39	69	29	0,14	30	100	0,35	78	76	0,35	**449**	**67**
FirePath Relative ID-based	43	54	0,52	34	27	0,46	102	100	0,60	55	23	0,37	19	63	0,52	78	76	0,56	**331**	**49**
Selenium IDE	12	15	0,84	22	18	0,85	35	34	0,87	23	10	0,81	11	37	0,85	4	4	0,82	**107**	**16**
Montoto	10	13	0,76	7	6	0,74	39	38	0,80	68	29	0,81	16	53	0,79	11	11	0,76	**151**	**22**
ROBULA+	10	13	0,89	3	2	0,86	30	29	0,92	22	9	0,87	10	33	0,89	3	3	0,87	**78**	**12**
Unweighted Multi-Locator (Worst Order)	9	11		3	2		18	18		24	10		9	30		5	5		**68**	**10**
Unweighted Multi-Locator (Best Order)	7	9		3	2		20	20		18	8		9	30		2	2		**59**	**9**
Weighted Multi-Locator (CrossValidation)	3	4		3	2		20	20		18	8		9	30		2	2		**55**	**8**
Theoretical Limit	1	1		3	2		16	16		15	6		9	30		2	2		**46**	**7**

%: percentage of broken locators over the total number of locators of this kind - **Weight**: represents the average robustness of this kind of locator computed on the other five applications

detail, we excluded multiple instances of: (1) the same web element repeated in different web pages as part, for instance, of the header or the footer (e.g., the link to the home page of the web application can be found in every page and has exactly the same locator), and (2) similar web elements from common groups (e.g., for a calendar with a check box for each day we selected only one of the check boxes).

(III) For each selected web element in the first release (located by the absolute XPath abs, obtained from FirePath) we manually defined a mapping (abs→abs') that associates it with its counterpart in the second release (located by the absolute XPath abs', also obtained from FirePath). The absolute XPath locators defined on the second release of the applications are used as oracle to verify the robustness of the generated XPath locators for the elements of the first release of the applications.

(IV) For the first release of each web application and for each web element (located by the absolute XPath defined above), four additional XPath locators have been created by using respectively: (1) FirePath Relative ID-based, (2) Selenium IDE, (3) Montoto, and (4) ROBULA+.

(V) For each web element, we applied the unweighted and weighted variants of the multi-locator. We have defined the weights for the weighted variant of multi-locator using k-fold cross validation [6]. In particular, we used a leave-one out cross validation with k = 6, where 6 is the size of the original data set (i.e., the number of web applications selected for this experiment). Thus, we split the original data set into five applications used for training and one application used for testing, with the testing application rotated so as to test the multi-locator on each of the six available applications. Finally, for each test application, we evaluate the robustness of the multi-locator using weights proportional to the average robustness of the single-locators algorithms, measured when these are applied to the other five training applications (see columns "Weight" in Table II).

To answer RQ1, for each web element, the robustness of the multi-locator is automatically evaluated against the oracle on the next release of the web application by verifying whether it is still able to locate the web element of interest. To this end, we verify if the web element selected by the multi-locator and by the absolute locator abs' is the same. The voting procedure of the unweighted multi-locator could generate ties and in these cases we decided to randomly select one element. Thus, to measure the performance boundaries associated with such non-deterministic choice, we considered the two extreme cases that may happen when the random selection is done: the best case and the worst case. In particular, based on the results collected in our experiments, we defined the best order as: ROBULA+, Selenium IDE, Montoto, Relative ID-based, Absolute. The worst order is the reverse one. Instead of considering a random element, in case of parity, we report the results obtained in two cases: *best order* and *worst order*.

To answer RQ2, we evaluated the robustness of the multi-locator produced by the weighted multi-locator as done in the previous step.

To answer RQ3, for each application, we compared the theoretical limit — computed as described in Section III-A — with the results of the multi-locator.

To answer RQ4, for each application, we counted the number of correctly repaired broken locators in the multi-locators and analysed all the actions performed by the repair algorithm described in Section III-B.

To answer RQ5, only for three different applications (Claroline, AddressBook, PPMA) we have built two Selenium WebDriver test suites: one that localizes the web elements using the absolute XPath and the other that uses the multi-locator selection algorithm. We have re-executed three times these test suites comparing the mean times of the single locator versions with the ones of the multi-locator, so as to measure the average overhead of the multi-locator for each application.

D. *Results*

Table II reports the data used to answer RQ1, RQ2, and RQ3. For each application and for each target web element, it reports the number of broken locators and the corresponding breakage percentage over the total number of locators. In the last two columns, aggregate results over all the six web applications are also reported.

Based on these results we can notice that absolute XPath locators are the most fragile in the considered set of locators. In three cases (i.e., Collabtive, MRBS, PPMA) out of six, all absolute locators are broken. Considering all six applications, 449 over 675 absolute locators (i.e., 67%) are broken.

The results of FirePath relative XPath locators are better than those of absolute XPath locators. Still, in MRBS all relative locators are broken and over the six applications, 331 out of 675 absolute locators (i.e., 49%) are broken.

The locators generated by state of the art XPath generator algorithms targeting web testware evolution are more robust than the previous ones. In particular, the most robust is ROBULA+, whose XPath locators are broken only in 12% of the cases (78 out of 675). The locators produced by Selenium IDE achieve also a very high level of robustness (16% of broken locators). The algorithm proposed by Montoto et al. is also quite good, with 151 broken locators out of 675 (i.e., 22%).

In order to answer our research questions, we analyse the experimental results quantitatively. The reasons and implications of the results are further analysed qualitatively in Section IV-E.

TABLE III. Actions performed by the repair algorithm (Correctly Repaired, Incorrectly Repaired, Unmodified and Unrepairable)

			Address Book	Collabtive	MRBS	Claroline	PPMA	Mantis	All Apps
Correct	C1	Correct Locators - No Repair Triggered	279	434	198	936	64	341	**2252**
	C2	Correct Locators - Incorrectly Repaired	1	0	4	2	0	0	**7**
	CT	**Total Correct Locators (C1+C2)**	**280**	**434**	**202**	**938**	**64**	**341**	**2259**
Broken	B1	Broken Locators - Correctly Repaired	106	176	212	149	41	164	**848**
	B2	Broken Locators - No Repair Triggered	10	0	11	31	5	2	**59**
	B3	Broken Locators - Incorrectly Repaired	4	0	20	22	10	3	**59**
	B4	Broken Locators - Unrepairable	0	15	65	35	30	5	**150**
	BT	**Total Broken Locators (B1+B2+B3+B4)**	**120**	**191**	**308**	**237**	**86**	**174**	**1116**
	T	**Total Locators (CT+BT)**	**400**	**625**	**510**	**1175**	**150**	**515**	**3375**

RQ1: The unweighted multi-locator (worst order) is more or equally robust as single locators in almost all the cases, with the only exception of Selenium IDE and Robula+ in the cases of Mantis and Claroline. Overall, multi-locator (worst order) is able to outperform Robula+, the algorithm that produces the most robust locators, globally reducing the number of broken locator from 78 to 68 (12.8% reduction). Multi-locator (best order) improves the results of multi-locator (worst order) and further reduces the number of broken locators, to 59, corresponding to reduce by 24.4% the number of broken locators w.r.t. Robula+. Only in the case of MRBS, multi-locator (worst order) is able to perform slightly better (i.e., -2 broken) than multi-locator (best order).

RQ2: The adoption of the weights (see columns "Weight" in Table II) allows multi-locator to further improve the results provided by multi-locator (worst and best order). Indeed, weighted multi-locator is able to reduce the number of broken locators to 55, corresponding to reduce by 29.5% the number of broken locators w.r.t. Robula+, and to achieve an overall percentage of only 8% broken locators. It is interesting to notice that in this case, there is no single locator algorithm, among the ones we considered, which is able to outperform weighted multi-locator in any case. Only in the case of MRBS, multi-locator (worst order) is able to perform slightly better (i.e., -2 broken) than weighted multi-locator.

RQ3: The results in Table II show that weighted multi-locator is able to reach the theoretical limit in 3 cases out of 6, i.e., for web applications Collabtive, PPMA, and Mantis. In the other cases, its results are quite close to the best achievable ones (with the considered algorithms). Indeed, applying the multi-locator using the outputs (i.e., the locators) of the five selected algorithms, it is not possible to have less than 46 broken locators out 675 and weighted multi-locator has only 55 broken locators in total (i.e., it is only 1.3% from the best possible results, corresponding to only 9 locators out of 675).

RQ4: Table III reports the data about the execution of the Multi-locator Repair algorithm described in Section III-B. The analysis is conducted considering each locator composing the multi-locator, thus in our case five for each target web element (e.g., AddressBook has 80 web elements thus there are 400 locators). The locators composing the multi-locator set L can be *correct* or *broken* depending on whether they are able to locate the correct element on D' or not. The *correct locators* in L can be: (C1) simply copied in L' if the multi-locator is correct; or (C2) incorrectly repaired, if the multi-locator selects the wrong element. The *broken locators* in L can be: (B1) correctly repaired if the multi-locator is correct; (B2) simply copied in L' if the multi-locator is broken and selects the same wrong element; or (B3) incorrectly repaired, if the multi-locator selects a different wrong element. When the multi-locator is not able to select any element on D', i.e., when all the locators in L are not able to select any element, no repair action is possible (B4). From the data, it is possible to notice that in the majority of the cases (92%), our algorithm performs the correct action (see the rows C1 and B1 in green), i.e., for 3100 locators out of 3375. Overall, 848 locators are correctly repaired (C1) over a total 1116 broken locators (76%). The incorrect repairs of correct locators (C2) are only 7 out of 2259 (0.31%). To answer RQ4, we focus only on the repair actions, i.e., when the locators are modified (914 cases, B1, C2, B3), and we can observe that 848 locators (i.e., 93%) are correctly repaired (B1), while only 66 locators are repaired incorrectly (C2, B3).

RQ5: The overhead of the multi-locator selection algorithm on test case execution is very low in general. Indeed, in the worst case, Claroline, the test suite is composed by 18 test cases and a complete execution of the entire test suite requires on average 84.6 seconds when using one locator per web element, which corresponds in total to 132 XPath locators being evaluated, and 87.8 seconds when using the multi-locator selection, which corresponds in total to 660 XPath locators being evaluated. Thus, the increment of the time required for a complete execution of the test suite is only 3.8%. With AddressBook and PPMA, composed respectively by 13 and 21 test cases, the overhead due to the introduction of the multi-locator is even lower, being respectively 2.9% (from 84 to 420 XPaths are evaluated) and 2.8% (from 143 to 715 XPaths are evaluated).

E. Qualitative Analysis and Discussion

Weighted multi-locator has the best performance, but even the locators selected by the *unweighted* multi-locator are more robust that the ones generated by the best algorithm considered in this work (i.e., Robula+). In case of a tie, the unweighted multi-locator selects randomly the locator to return. Hence, the robustness of the unweighted multi-locator is intermediate between the worst order and the best order of selection, considered in our experiment. Since unweighted multi-locator (worst order) is already better than Robula+, we conclude that the multi-locator is beneficial even when uniform weights are used. Analysing the results, we discover that often 2 or 3 locators (generally the ones generated by Robula+, Montoto and Selenium IDE) select the correct target element. Using the unweighted multi-locator, the votes assigned to the correct web elements range typically from 2 to 5, thus there are no cases in which an element is chosen since voted only by one algorithm and the unweighted variant is often enough to improve the performance of single locators.

The difference between multi-locator (best order) and multi-locator (worst order) is due to the cases in which there is parity in the votes assigned to the candidate web elements (in the other cases their behaviours are exactly the same). Looking at Table II, we can see that the absolute locators generated by FirePath are usually broken in the highest number of cases (67%), while the ones generated by ROBULA+ are broken in the lowest number of cases (12%). Thus, in case of parity, multi-locator (worst order) selects the locator generated by FirePath, which has a good chance of being broken, while multi-locator (best order) selects the element voted by ROBULA+, having many more chances of being correct. Since the unweighted multi-locator makes a random choice in case of parity, its actual performance will be intermediate between multi-locator (best order) and multi-locator (worst order) which means that, overall, it performs better that any single locator.

Weighted multi-locator has the best performance. Its improvement over the unweighted multi-locator is a reduction of broken locators between 4 and 13 (best/worst order, respectively). It is interesting to notice that making an optimally ordered choice (best order) in case of parity leads to very similar results as those obtained with carefully determined weights. On the other hand, such optimal ordering is unknown when the unweighted multi-locator is used, so it represents just the upper bound for the performance expected when the multi-locator has to make a random choice, in case of parity.

The locator repair algorithm is able to perform the correct repairs in most of the cases. In Collabtive it repairs all the broken locators without making any error. In fact, in this case every time the multi-locator is broken (3 cases) it does not select any element. Thus the repair algorithm is not triggered. Overall, the multi-locator is broken in 55 cases of which in 30 cases (*false negatives*) it does not select any element – hence, no incorrect repair is performed – in 25 cases (*false positives*) it selects the wrong element – hence, the repair algorithm is executed using the wrong element to repair the other locators.

If we consider the weights assigned to the five locators creation algorithms by the cross-validation procedure (see columns "Weight" in Table II), we can see that they agree on the ranking of the algorithms, independently of the web application left out by the cross-validation procedure. ROBULA+ is always assigned the highest weight, followed by Selenium IDE and Montoto. FirePath Relative ID-based and FirePath Absolute close the ranking. This means that for a practical adoption of the multi-locator it is not even necessary to execute the cross-validation procedure to obtain the weights, since any reasonable weight assignment that respects the order ROBULA+, Selenium IDE, Montoto, FirePath Relative ID-based and FirePath Absolute is expected to work fine. For instance, the following weights can be used in practice: ROBULA+ 0.90, Selenium IDE 0.85, Montoto 0.80, FirePath Relative ID-based 0.50, and FirePath Absolute 0.33. On our subjects, these weights provide exactly the same result of the weighted multi-locator with cross validated weights.

The other issue potentially affecting the adoption of the multi-locator is the test case execution overhead, but our experimental data show that this is negligible, even in the worst case. Hence, we conclude that: the proposed approach (1) is very easy to adopt (e.g., weights can be assigned heuristically, without using cross-validation), and (2) offers major benefits, in terms of robustness of the locators during software evolution.

F. Threats to Validity

One threat to the internal validity of our study is associated with the approach used to select the target web elements. To remove this threat, we adopted the procedure described in Section IV-C. While the choice of the releases considered in this study may have affected the results of RQ1, RQ2, RQ3, we have no reason to believe that the ranking of the algorithms in terms of broken locators, as reported in Table II, would vary significantly considering different releases, although the magnitude of our findings might change. Concerning the strategy used to assign weights to the weighted multi-locator, it is clear that other choices (e.g., minimising the number of broken locators selected by the multi-locator, instead of just measuring the robustness of each algorithm) are possible. However, from the obtained results it is clear that this choice is not so critical, since a simple weight assignment based on each algorithm's robustness gives already results close to the theoretical limit. Finally, concerning the generalization of results, we selected real open source web applications belonging to different domains, which makes the context realistic, even though further studies with other applications are necessary to corroborate the obtained results.

V. RELATED WORK

The problem of test script maintenance and repair has been extensively studied by the research community. Related to our work are a number of test case and test script repair techniques, and some others using voting mechanisms in different testing activities.

Test case/script repair. *Grechanik et al.* [7] describe an approach for maintaining and evolving test scripts by means of GUI-tree diffs, in order to find altered GUI objects. *Choudhary et al.* [3] propose WATER, a tool that suggests changes that can be applied to repair test scripts for web applications. It compares the test executions of two successive releases of a web application. By analysing the difference between the two executions, it suggests repairs to the script code. *Thummalapenta et al.* [19] present ATA, a tool to automatically repair test scripts. For certain types of applications or environment changes, they are able to repair the XPath on-the-fly. *Yandrapally et al.* [23] show a novel solution to the problem of test-script fragility based on what they define as "contextual clues". Their approach is promising because it takes advantage of multiple aspects of the representation of the application, including textual information used as anchor point. *Fard et al.* [14] mine an existing test suite to gather input and assertion information, and extend it to the uncovered portions of the web application by means of automated crawling and test generation techniques. *Mirzaaghaei et al.* [15] present TestCareAssistant, a technique that automatically repairs test cases broken due to changes in method declarations. *Daniel et al.* [4] use GUI change refactoring information to repair the test code accordingly. *Memon et al.* [13] describe a method to repair GUI test scripts by means of user-specified transformations. *Zhang et al.* [24] present FlowFixer, a technique able to automatically migrate scripts towards a new and evolved GUI.

Differently from the works aforementioned, our work aims at strengthening the test scripts by means of a multi-locator, which is useful in two respects: *(I)* to make the test script more robust, by taking advantage of redundant information; *(II)* to repair the locators set itself, for the successive releases of the software. To the best of our knowledge, no previous work proposed and evaluated the effectiveness of a voting procedure within such context, designed to make the script more resilient to software changes. Our repair mechanism is also different from the existing ones, because it resorts on voting to select the repair action among those provided by the alternative algorithms aggregated by the multi-locator.

Voting mechanisms. *Tsai et al.* [21] use a voting mechanism to help with failure detection and oracle creation. The voting mechanism aims at solving the test oracle problem statistically, by analysing the outputs of the web services in group testing. During execution of the voting mechanism, the reliability of web services is calculated and test cases are ranked according to their ability to reveal faults. *Nguyen et al.* [17] use an IR-based ranking technique for test case prioritisation.

Differently from these works, we use a voting mechanism to choose the best locator among those in the multi-locator set.

VI. Conclusions and Future Work

The main sources of fragility for web test scripts are web element locators that must be repaired manually when the software evolves and locators get broken. Algorithms for the creation of robust locators have different strengths and weaknesses; they often exhibit complementary performance. For this reason, we proposed the multi-locator, a novel approach that uses a voting decision procedure to aggregate the results of multiple, alternative locators for producing a consolidated locator. Adoption of the multi-locator requires minimal effort and has minimal impact: (1) its parameters (weights) can be easily approximated heuristically; (2) the automated repair actions it performs are correct most of the times; (3) the execution overhead introduced by the multi-locator selection algorithm is negligible, and (4) existing single locator test suites can be migrated to the multi-locator approach automatically. Experimental results show that the multi-locator is substantially more robust than single locators, since it reduces by 29.5% the number of broken locators w.r.t. the best single locator algorithm (ROBULA+). This may represent a substantial saving of test case repair effort in case, e.g., of large industrial test suites.

In our future work, we plan to: (1) experiment with more web applications, (2) analyse the effectiveness of the repair algorithm across more than two releases of the applications, (3) analyse the contribution of the various algorithms to the creation of the multi-locators. Moreover, we plan to complete the development of the tool for automating the migration of existing DOM-based test suite to the multi-locator approach using a technique similar to the one we adopted in a previous work [12], [18]. Once the tools will be completed, we plan to make it available for download together with a Java implementation of each XPath generation algorithm used by the multi-locator. We will also extend our work beyond the area of structural locators, considering visual locators [10], which take advantage of image recognition to identify the web elements.

References

[1] B. Beizer. *Software Testing Techniques (2nd Ed.)*. Van Nostrand Reinhold Co., New York, NY, USA, 1990.

[2] S. Berner, R. Weber, and R. Keller. Observations and lessons learned from automated testing. In *Proceedings of 27th International Conference on Software Engineering*, ICSE 2005, pages 571–579. IEEE, 2005.

[3] S. R. Choudhary, D. Zhao, H. Versee, and A. Orso. WATER: Web application test repair. In *Proceedings of 1st International Workshop on End-to-End Test Script Engineering*, ETSE 2011, pages 24–29. ACM, 2011.

[4] B. Daniel, Q. Luo, M. Mirzaaghaei, D. Dig, D. Marinov, and M. Pezze. Automated GUI refactoring and test script repair. In *Proceedings of 1st International Workshop on End-to-End Test Script Engineering*, ETSE 2011, pages 38–41. ACM, 2011.

[5] M. Fewster and D. Graham. *Software Test Automation: Effective Use of Test Execution Tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[6] S. Geisser. *Predictive Inference*. Chapman & Hall/CRC Monographs on Statistics & Applied Probability. Taylor & Francis, 1993.

[7] M. Grechanik, Q. Xie, and C. Fu. Maintaining and evolving GUI-directed test scripts. In *Proceedings of 31st International Conference on Software Engineering*, ICSE 2009, pages 408–418. IEEE, 2009.

[8] M. Leotta, D. Clerissi, F. Ricca, and C. Spadaro. Comparing the maintainability of Selenium WebDriver test suites employing different locators: A case study. In *Proceedings of 1st International Workshop on Joining AcadeMiA and Industry Contributions to testing Automation*, JAMAICA 2013, pages 53–58. ACM, 2013.

[9] M. Leotta, D. Clerissi, F. Ricca, and P. Tonella. Capture-replay vs. programmable web testing: An empirical assessment during test case evolution. In *Proceedings of 20th Working Conference on Reverse Engineering*, WCRE 2013, pages 272–281. IEEE, 2013.

[10] M. Leotta, D. Clerissi, F. Ricca, and P. Tonella. Visual vs. DOM-based web locators: An empirical study. In *Proceedings of 14th International Conference on Web Engineering (ICWE 2014)*, volume 8541 of *LNCS*, pages 322–340. Springer, 2014.

[11] M. Leotta, A. Stocco, F. Ricca, and P. Tonella. Reducing web test cases aging by means of robust XPath locators. In *Proceedings of 25th International Symposium on Software Reliability Engineering Workshops*, ISSREW 2014, pages 449–454. IEEE, 2014.

[12] M. Leotta, A. Stocco, F. Ricca, and P. Tonella. Automated generation of visual web tests from DOM-based web tests. In *Proceedings of 30th Symposium on Applied Computing*, SAC 2015. ACM, 2015.

[13] A. M. Memon. Automatically repairing event sequence-based GUI test suites for regression testing. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 18(2):4:1–4:36, Nov. 2008.

[14] A. Milani Fard, M. Mirzaaghaei, and A. Mesbah. Leveraging existing tests in automated test generation for web applications. In *Proceedings of 29th International Conference on Automated Software Engineering*, ASE 2014, pages 67–78. ACM, 2014.

[15] M. Mirzaaghaei, F. Pastore, and M. Pezze. Automatically repairing test cases for evolving method declarations. In *Proceedings of 26th International Conference on Software Maintenance*, ICSM 2010, pages 1–5. IEEE, 2010.

[16] P. Montoto, A. Pan, J. Raposo, F. Bellas, and J. Lopez. Automated browsing in AJAX websites. *Data & Knowl. Eng.*, 70(3):269–283, 2011.

[17] C. D. Nguyen, A. Marchetto, and P. Tonella. Test case prioritization for audit testing of evolving web services using information retrieval techniques. In *Proceedings of 9th International Conference on Web Services*, ICWS 2011, pages 636–643. IEEE, 2011.

[18] A. Stocco, M. Leotta, F. Ricca, and P. Tonella. PESTO: A tool for migrating DOM-based to visual web tests. In *Proceedings of 14th International Working Conference on Source Code Analysis and Manipulation*, SCAM 2014, pages 65–70. IEEE, 2014.

[19] S. Thummalapenta, P. Devaki, S. Sinha, S. Chandra, S. Gnanasundaram, D. D. Nagaraj, and S. Sathishkumar. Efficient and change-resilient test automation: An industrial case study. In *Proceedings of 35th International Conference on Software Engineering*, ICSE 2013, pages 1002–1011. IEEE, 2013.

[20] P. Tonella, F. Ricca, and A. Marchetto. Recent advances in web testing. *Advances in Computers*, 93:1–51, 2014.

[21] W.-T. Tsai, Y. Chen, D. Zhang, and H. Huang. Voting multi-dimensional data with deviations for web services under group testing. In *Proceedings of 25th International Conference on Distributed Computing Systems Workshops*, ICDCS Workshops 2005, pages 65–71. IEEE, 2005.

[22] C. Wohlin, P. Runeson, M. Höst, M. Ohlsson, B. Regnell, and A. Wesslén. *Experimentation in Software Engineering - An Introduction*. Kluwer Academic Publishers, 2000.

[23] R. Yandrapally, S. Thummalapenta, S. Sinha, and S. Chandra. Robust test automation using contextual clues. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ISSTA 2014, pages 304–314. ACM, 2014.

[24] S. Zhang, H. Ly, and M. D. Ernst. Automatically repairing broken workflows for evolving GUI applications. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, ISSTA 2013, pages 45–55. ACM, 2013.