Reducing Web Test Cases Aging by means of Robust XPath Locators

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*—In the context of web regression testing, the main aging factor for a test suite is related to the continuous evolution of the underlying web application that makes the test cases broken. This rapid decay forces the quality experts to evolve the testware. One of the major costs of test case evolution is due to the manual effort necessary to repair broken web page element locators. Locators are lines of source code identifying the web elements the test cases interact with. Web test cases rely heavily on locators, for instance to identify and fill the input portions of a web page (e.g., the form fields), to execute some computations (e.g., by locating and clicking on buttons) and to verify the correctness of the output (by locating the web page elements showing the results).**

In this paper we present ROBULA (ROBUst Locator Algorithm), a novel algorithm able to partially prevent and thus reduce the aging of web test cases by automatically generating robust XPath-based locators that are likely to work also when new releases of the web application are created. Preliminary results show that XPath locators produced by ROBULA are substantially more robust than absolute and relative locators, generated by state of the practice tools such as FirePath. Fragility of the test suites is reduced on average by 56% for absolute locators and 41% for relative locators.

***Keywords*-Web Testing; Test Cases Aging; Robust Locators.**

I. Introduction

Web applications are subject to a tremendous pressure for change. New releases are continuously produced, to accommodate bugs, security fixes and new functionalities, but often just to update the presentation style and align it with the most recent trends. In fact, the visual appearance of a web application is a major success factor. Within such ultra-rapid development cycles, web testing is an option [13] only if it is strongly supported by automated tools, which reduce the effort required from developers for test suite creation and execution. Moreover, testwares exhibit a significant aging[1] factor, due to the fast evolution of the web applications under test. For this reason, the test suites that accompany a web application must be often evolved, so as to keep test cases and application under test aligned.

Currently, the most widely used tools for web application testing are DOM-based testing tools [2], e.g., Selenium WebDriver [1]. They offer developers a rich programmable API that can be used to define DOM-based test cases. Using this API, developers can for instance *locate* an input text field, insert some text into it, *locate* a button, click on it, *locate* the text that shows the result of the computation and check whether it matches the expected behaviour of the web application. All these steps are programmed using a high level language (e.g., Java) in a similar way as done with JUnit.

Web element locators play a key role in web testing. Among them, XPath locators are remarkably powerful and flexible. They represent the most general choice, since the majority of locators can be specified using properly defined XPath expressions. In practice, manually defining a robust XPath locator turns out to be a difficult task, which requires substantial skills and experience. For this reason, tools and plugins (e.g., FirePath, XPath Checker, XPath Helper) exist which compute a candidate XPath locator for the developer. Such locators are either absolute or relative XPath expressions, which take advantage of heuristically selected tags and attributes (e.g., id or name) to try to increase their resilience to changes. By *robust* XPath locator we mean an XPath expression that continues to select the target web element, even if the web page has changed because of a new release of the web application. Existing tools often create simple and brittle XPath locators [7] and even minor modifications of the DOM structure may cause their failure, so that developers have to correct them when the application evolves.

In this paper, we propose a novel algorithm, called ROBULA (ROBUst Locator Algorithm), able to partially prevent and thus reduce the aging of web test cases by automatically generating robust web element locators. The algorithm starts with a generic XPath locator that returns all nodes ("//*"). It then iteratively refines the locator until only the element of interest is selected. In such iterative refinement, ROBULA applies four refinement transformations, according to a set of heuristic XPath specialisation steps. For all the six web applications considered in our experiment, ROBULA has generated consistently locators that are much more robust than those produced by existing tools, both in the absolute XPath category (with a fragility reduction equal to 56%) and in the relative category (with a 41% fragility reduction). To the best of our knowledge, ROBULA is the first publicly available solution to the problem of web testware aging. To further increase the adoption of ROBULA by practitioners, we are implementing it as a Firefox plugin, in addition to the standalone version.

The paper is organised as follows: Section II introduces the problems associated with web testware evolution and discusses

[1]In this work, with software aging we mean the obsolescence of a software source code and not the performance degradation of a system as it executes.

the way locators are produced and evolved. Section III describes our novel contribution: ROBULA. Preliminary empirical results about the robustness of the locators produced by ROBULA are reported in Section IV, followed by related work (Section V) and conclusions (Section VI).

II. Evolution of Web Test Cases

When a web application evolves to accommodate requirement changes – bug fixes or functionality extensions – test cases may become broken. For instance, test cases may be unable to locate some links, input fields and submit buttons, and software testers have to repair them. This is a tedious and time-consuming task, which has to be performed manually by software testers. Indeed, automatic evolution of test suites is far from being consolidated [9] even if some preliminary approaches have been proposed (e.g., [3]).

In this paper, we focus on reducing the web test suite maintenance effort due to structural changes (i.e., changes impacting the page layout/structure) since these are heavily affected by the robustness of the locators. On the other hand, logical changes (i.e., changes modifying the logic of the web application) 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 logic unaffected. We address the problem of structural changes by automatically generating robust XPath locators that retrieve the web elements required by the test cases.

A. DOM-based Locators

To locate web page elements such as links, buttons, and input fields, different kinds of locators can be employed. In particular, in the context of web application testing, three different categories of locators are used (more details on them can be found in [8]): (1) Coordinate-based locators, nowadays considered obsolete, (2) DOM-based locators, using the information contained in the Document Object Model (DOM) and (3) Visual locators, using image recognition techniques. In this paper, we focus on DOM-based localisation, since this is the most adopted technology in practice [2] and among the various kinds of DOM-based locators, we focus on XPath locators.

B. Why focusing on XPath Locators?

There are three main reasons to focus on XPath locators:
1) **XPath locators are highly expressive.** Actually, most of the other localisation methods provided by DOM-based tools can be simulated using XPath expressions. For example, the Selenium WebDriver method: driver.findElements(By.name('xy')) is equivalent to driver.findElements(By.xpath('//*[@name="xy"]')).
2) **XPath locators are sometimes the only option.** Some localisation methods are applicable only to specific cases (e.g., By.id is not applicable when the identifier is not present). With XPath expressions is always possible to locate every web page element. In our previous work [7], we considered six Selenium WebDriver test suites built for six different web applications. These test suites use specific localisation methods (e.g., By.id, By.name) whenever possible, but they still resort to XPath locators in 177 cases over a total of 487 (36%). On the other hand, all 310 locators that do not make use of XPaths can be easily rewritten as XPath locators with no substantial impact on their understandability.
3) **XPath locators are generally considered fragile, but this strongly depends on how they are created.** This common belief largely depends on how XPath locators are generated by tools. In our previous work [7], we used FirePath[2] to automatically generate XPath locators for the web elements. We found that for the considered test suites: 67% of the 177 XPath locators were broken from a release to the next one, while for the other types of locators the breakage percentages were extremely lower (less than 1% for the ID locators; about 20% for the Name, LinkText and CSS locators). By inspecting the XPath locators, we realised that their quality was largely sub-optimal and better XPath locators could be defined manually. However, careful manual definition of robust XPath locators requires a lot of experience and a big effort.

C. Generating XPath Locators in Practice

A developer can use various tools to build XPath locators. In web testing, a straightforward solution is having a browser-integrated plugin for XPath expression generation. FirePath and XPath Checker[3] are among the most popular XPath expression generators[4] for Mozilla Firefox. FirePath is the most downloaded and provides a development tool to edit, inspect and generate absolute and id-based relative XPath expressions. XPath Checker provides expressions similar to the FirePath ones. For what concerns Google Chrome plugins, among the most used ones are the built-in XPath generator plugin and XPath Helper[5]. The expressiveness of Chrome's built-in plugin is similar to that of the Mozilla add-ons. XPath Helper is more limited, since it is only able to generate absolute XPath expressions, enriched with an attribute (if available), for every tag.

D. 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). In this way it is possible to test the correct saving of the information in the database.

To implement this test case, it is necessary to locate some web page elements as, for instance, the field of the table showing

[2] https://addons.mozilla.org/firefox/addon/firepath/
[3] https://addons.mozilla.org/firefox/addon/xpath-checker/
[4] http://docs.seleniumhq.org/docs/02_selenium_ide.jsp#locating-elements
[5] https://chrome.google.com/webstore/detail/xpath-helper/hgimnogjllphhhkhlmebbmlgjoejdpjl

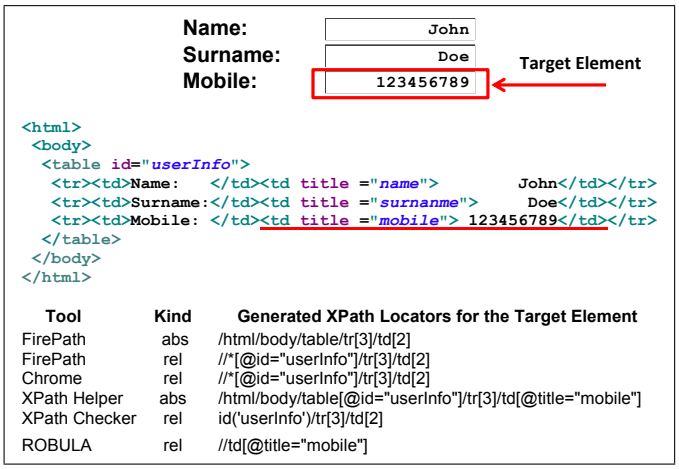

Fig. 1. showInfo.php – Ver. 1 – Page, Source, Locators

the mobile phone number (see the underlined td in Fig. 1 (center)). With Selenium WebDriver, the following methods: By.id, By.name, By.className are not applicable, since the target element has no id, name and className attributes. By.tagName is applicable, but does not allow to create a locator since multiple td elements are present in the page. By.linkText and By.partialLinkText are not applicable since the target element is not a link. Thus, we have to employ an XPath locator and the straight solution is using one of the XPath generator tools mentioned in Section II-C. Fig. 1 (bottom) lists the XPath locators provided by these tools and by ROBULA (the algorithm we propose in the following). The various tools create either relative (rel) or absolute (abs) XPath locators and, to this end, different generation strategies are adopted resulting in different expressions.

We now consider a new version of the web application (Ver. 2) after a maintenance intervention has been executed, in which the user is allowed 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

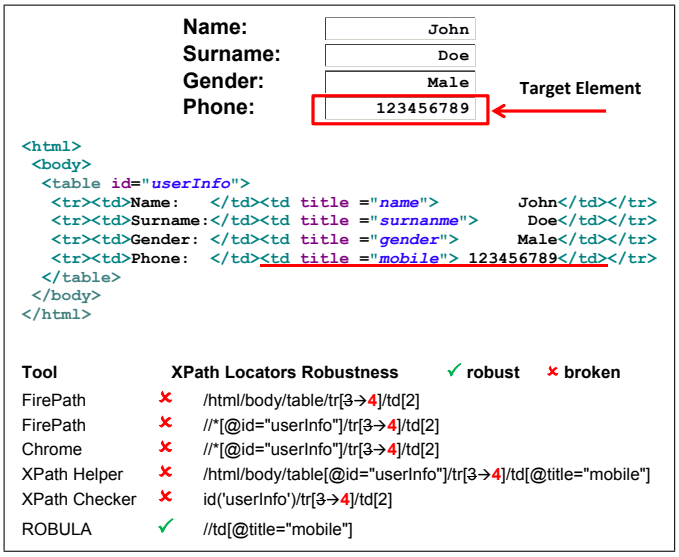

Fig. 2. showInfo.php – Ver. 2 – Page, Source, Locators

be broken (and will have to be repaired) or will work without problems. Looking at Fig. 2 (bottom), we can see that only the locator generated by ROBULA works, while all the other locators are broken. Indeed, all of them include the node tr[3] that in the new release becomes tr[4]. Some of them do not work because they locate another element (i.e., the "gender" field), while others are not able to locate any element (e.g., the locator generated by XPath Helper). Thus, when using a generic XPath generator, the test case must be repaired, while with ROBULA no modifications are needed.

In this regard it is important to highlight two aspects. First, the change to the application shown in this simple example replicates a code evolution pattern that we frequently encountered in the empirical evaluation of ROBULA. In such cases, our algorithm was often able to generate robust XPath locators. Second, even if by looking at the example it might seem quite easy to manually define the XPath locator generated by ROBULA (at least for an expert web tester), this is actually not the case when one works with real web pages containing hundreds of tags. Often, in these complex cases ROBULA finds locators that make use of complex combinations of attributes (e.g., //tr[@class="row-2"]/td[@class="center"]).

III. Robust Localization of Web Page Elements

In this section, we describe our ROBUst Locator Algorithm (ROBULA), which generates robust XPath locators. ROBULA follows a top-down approach, by starting from the most general XPath expression (i.e., "//*", matching all the elements in the document) and specialising it via transformation steps. The pseudo code of ROBULA is shown in Fig. 3. The algorithm takes in input a document d (e.g., an HTML page) and an absolute XPath abs selecting the target web page element e (e.g., an anchor, a text field, a button, etc.). For this web page element, the algorithm returns res (line 26), a robust relative XPath expression (if anyone exists) able to uniquely select the

```
1.  XPath ROBULA(XPath abs, Document d)
2.  {
3.    Element e = eval(abs, d).getFirst();
4.      //abs is a locator in d => eval(abs, d).size() == 1
5.    XPath res = new XPath();
6.    List<XPath> p = ["//*"];
7.    List<XPath> temp = [];
8.    while (res.isEmpty())
9.    {
10.     XPath w = p.removeFirst();
11.     temp = [];
12.     if (w.startsWith("//*"))
13.       temp.addAll(transf1(w));
14.     else
15.     {
16.       temp.addAll(transf2(w));
17.       temp.addAll(transf3(w));
18.     }
19.     temp.addAll(transf4(w));
20.     for (XPath x : temp)
21.     {
22.       if (uniquelyLocate(x, e, d)) {res = x; break;}
23.       else if (locate(x, e, d)) add(x, p);
24.     }
25.   }
26.   return res;
27. }

List<XPath> eval(XPath abs, Document d):
       returns the elements in d selected by the XPath abs
Boolean uniquelyLocate(XPath x, Element e, Document d):
       TRUE iff eval(x, d) contains only e
Boolean locate(XPath x, Element e, Document d):
       TRUE iff eval(x, d) contains e
```

Fig. 3. Pseudocode of ROBULA

The transformations work as follows:
- **transf1** replaces the $*$ in the initial $//*$ with the tag name of the element **L**.get(**N**)
  e.g., $//*/td \rightarrow //tr/td$
- **transf2** adds the predicates (one at time) of the element **L**.get(**N**) to the first node in **w**.
  e.g., $//tr/td \rightarrow //tr[@name = \text{`}data\text{'}]/td$
- **transf3** adds the position of the element **L**.get(**N**) to the first node in **w**
  e.g., $//tr/td \rightarrow //tr[2]/td$
- **transf4** adds $//*$ at the top of **w** (iff **N** < **L**.lenght())
  e.g., $//tr/td \rightarrow //*/tr/td$

Where:
- **w** = the XPath expression to specialize, e.g., $//td$
- **N** = the length (in nodes/levels) of **w**
  e.g., $//td \Rightarrow$ **N**=1; $//*/td \Rightarrow$ **N**=2;
- **L** = the list of elements ancestor of target element **e** in the considered DOM (i.e., web page), starting and including **e**
  e.g., $[td, tr, table, body, html]$
  L.get(2) returns the element tr

Fig. 4. Specialization transformations used by ROBULA

target element, i.e., a robust XPath locator. If a relative XPath expression does not exist, ROBULA returns an absolute XPath similar to the one taken in input.

The algorithm starts its execution by retrieving the element e selected by the absolute XPath abs (line 3) and initialising the list p of XPath expressions with the most general one (i.e., "//*") (line 6).

Then, it iterates until a result (i.e., an XPath locator) is found (line 8). At each cycle, it removes the first XPath expression (w) from the list p and it applies, in an established order, four transformations (transf1, transf2, trans3, and transf4) to specialise w. Precisely, if w starts with "//*" (line 12) then w is specialised using transf1, otherwise transf2, and transf3 (lines 16-17) are applied. Finally, transf4 is always applied (line 19).

The transformations work as shown in Fig. 4. transf1 introduces a specific tag name to replace the wildcard "*" in the XPath expression. transf2 and transf3 add constraints to a tag which is already in the XPath expression, by respectively considering only tags containing specific attribute-value pairs or only tags at specific positions. transf4 adds a new level to the current XPath expression by extending it with a wildcard tag "*" added at the beginning.

All the XPath expressions generated by applying these transformations are inserted into a list named temp. At this point (lines 22 and 23), the algorithm cycles through the XPath expressions contained in temp, considering in turn each XPath expression x. If x is a locator for the target element e (the function uniquelyLocate is used to determine this), the algorithm returns it and terminates. Otherwise, if x selects more elements including the target one, x is inserted into the list p, to be specialised in the next iteration of the algorithm (i.e., the target element is among the elements retrieved by these XPath expressions, whose result set contains more than one element). The remaining XPath expressions, which are not able to locate e, are discarded, since no further specialisation steps can generate a locator for e.

A. Algorithm's Implementation and Analysis

ROBULA has been implemented in Java using JSoup[6] to generate an equivalent XHTML code for the HTML web pages taken in input and JDOM[7] for manipulating XHTML documents and evaluate XPath expressions on them. For the interested reader, ROBULA can be downloaded from http://sepl.dibris.unige.it/2014-ROBULA.php.

Since our implementation returns only the first locator found by the algorithm, it is very important to consider the order of execution of the transformations. If the current XPath expression w starts with //*, only transf1 is executed so we have no problem of order. Otherwise, we can execute two transformations (transf2 and transf3), and an order of execution must be defined. We decided to choose as first transformation transf2 since attribute values are usually more robust than position values. Indeed, attribute values usually include meaningful names (e.g., id="result" or name="username"), while position values are always bound to the web page structure and so they are more fragile [11]. We decided to execute transf4, which adds //* in front of an XPath expression, as the last transformation since we want to maintain the XPath locators as short as possible (in principle, a short locator is less coupled with the page structure than a long one, so it should be more robust).

The algorithm is ensured to terminate, since in the worst case it returns an absolute XPath (similar to the one taken in input). This is actually the case in which it is not possible to generate a shorter locator. In fact, every element in the DOM tree can be uniquely located by an absolute XPath that contains only tag names and positions (this is a straightforward consequence of the correspondence between DOM node names and HTML tag names). Among the transformations in Fig. 4, transf1 can be used to add the tag name and transf3 to add the position. Hence, repeated applications of these two transformations will generate an absolute path consisting of all tags, possibly including element positions, from the root to the element to be located.

The worst-case computational complexity of ROBULA is exponential in the DOM size. Indeed, the heuristics introduced in the algorithm (i.e., the order of execution of transformations) aim at making the execution time acceptable in practical cases.

IV. Preliminary Experimental Results

This section sketches the design, objects, research questions, metrics, procedure, and results of a preliminary empirical study conducted to evaluate the robustness of the XPath locators generated by ROBULA. We follow the guidelines by Wohlin *et al.* [14] on designing and reporting of empirical studies in software engineering.

[6] http://jsoup.org/
[7] http://www.jdom.org/

TABLE I
ROBUSTNESS OF ABSOLUTE, RELATIVE, AND ROBULA

	Address Book		Collabtive		MRBS		Claroline		PPMA		Mantis		All	
	Broken	%	Broken	%	Broken	%	Broken	%	Broken	%	Broken	%	Broken	%
Absolute	45	56	125	100	102	100	72	31	30	100	78	76	452	67
Relative	43	54	34	27	102	100	63	27	19	63	78	76	339	50
ROBULA	8	10	4	3	63	62	93	40	11	37	20	19	199	29
Total # of Locators	80		125		102		235		30		103		675	

A. Study Design

The *goal* of this study is to analyse the robustness of the XPath locators generated by ROBULA with the purpose of understanding the strengths and the weaknesses of the approach it implements. The results of this study are interpreted according to the developers and project managers *perspective*, interested in data about the benefits of adopting ROBULA in an industrial context to create XPath locators. The *software objects* are six open source web applications already used in a different work [8].

B. Research Question and Metrics

Our preliminary study aims at answering the following research question:

RQ1: *Does ROBULA reduce the number of broken XPath locators?*

The goal of the research question is to compare the robustness of the XPath locators generated by ROBULA with the robustness of the absolute and id-based relative XPath locators generated by a state of the practice XPath generator tool (in our experiment we used FirePath, release 0.9.7). This would give developers and project managers a precise idea of the benefits coming from the adoption of ROBULA as XPath locator generator. The metrics used to answer RQ1 is the number of broken XPath locators in the next software release.

C. Procedure

To answer RQ1, we proceeded as follows:

– We selected six open-source web applications from *SourceForge.net*.
– For each application and for each web page we manually selected a set of web elements relevant for the test of the application, while avoiding multiple instances of the same web element from a common group (e.g., in a calendar we selected only the first day link). In particular, we selected web elements: (1) on which it is possible to perform actions (e.g., links, input fields, submit buttons); (2) which report relevant information (e.g., a div containing a string that can be used to evaluate an assertion); (3) which belong to pages related to core functionalities of the application (e.g., we have not considered 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.
– For the first release of each web application and for each web element (located by an absolute XPath), we used FirePath to generate the id-based relative XPath locator (used as baseline) and ROBULA to generate the robust XPath locator. The result of this activity is, for each web element of the first release of each web application, three XPath locators: Absolute, Relative (id-based), and ROBULA.
– For each web element, we evaluated the robustness of the previously computed locators on the next release of the web application by verifying whether they are still able to locate the web element of interest.

D. Results

Table I reports the data used to answer RQ1. For each application and for each kind of locator (i.e., absolute, id-based relative, and ROBULA) it reports the number of broken locators and the corresponding breakage percentage over the total number of locators. In the last columns, we report aggregate results over all six web applications.

As expected, the performance of absolute XPath locators is not good. In three cases (i.e., Collabtive, MRBS, PPMA) out of six, all absolute locators are broken (i.e., they are never able to locate the corresponding web page elements in the new release of the application). In total, considering all six applications, 452 over 675 absolute locators result broken (i.e., 67%). This result confirms what we found in a previous work [6] and what is reported in [5], i.e., absolute XPath locators generated by state of the practice XPath generator tools are generally very fragile.

Results of FirePath id-based relative XPath locators are better than those of absolute XPath locators. Still, in MRBS all relative locators are broken and over the six applications, 339 out of 675 absolute locators are broken (i.e., 50%). These results are consistent with the ones we reported in our previous work [7], where, for the six considered test suites, we found that 67% of the 177 XPath locators were broken from a release to the next one (see Section II-B). Moreover, they confirm the common belief that in general: (1) XPath locators are very fragile; and, (2) relative XPath locators are better than absolute ones.

RQ1: the XPath locators generated by ROBULA are more robust than the absolute and relative locators in all the cases, with the exception of Claroline, see Table I. In five cases, out of six, the adoption of ROBULA results in a significant reduction of fragility of the XPath locators. Specifically, by adopting the locators generated by ROBULA, we obtained, on average, a 56% (computed as (452–199)/452) fragility reduction with respect to the absolute XPath locators, and a 41% reduction w.r.t. id-based relative locators. In the case of MRBS, the advantage of adopting ROBULA locators is slightly reduced, although still significant (38% less broken locators w.r.t. both absolute and relative locators). Only in the case of Claroline we obtained worse results when adopting ROBULA, with an

increment in the number of broken locators of 29% and 48% with respect to absolute and relative locators, respectively. In this case, ROBULA often used the href attributes to build the XPath locators. These attributes proved to be "unreliable" in practice (i.e., they changed often between the considered releases).

ROBULA locator Example: Usually, ROBULA generates XPath locators by far more robust and readable than the id-based relative XPath locators produced by FirePath. As an example, we will consider a locator for PPMA. To locate a link used for updating a password, ROBULA generated the following XPath locator //a[@title="Update"] while Firepath generated the following id-based relative XPath locator //*[@id="yw2"]/table/tbody/tr/td[6]/a[1] and the following absolute XPath locator html/body/div[1]/div[4]/div[1]/div/div[2]/table/tbody/tr/td[6]/a[1]. In this case, only the ROBULA XPath locator worked without any modification on the second release of PPMA while the locators generated by FirePath were broken and required several modifications to locate the web element of interest on the second release of PPMA. These were the resulting locators after manual modification: //*[@id="yw1"]/table/tbody/tr/td[4]/a[2] and html/body/div[1]/div/div/div[3]/table/tbody/tr/td[4]/a[2].

V. Related Work

Montoto et al. [10] propose an automated system for navigating AJAX websites. They use XPath expressions to identify the target elements of user actions on a web page. The XPath expressions they generate are quite different from ours. Indeed, ROBULA has been developed in order to create the simplest XPath expressions (i.e., avoiding to insert unnecessary information), since we think that keeping the expressions short increases their resilience to changes. For instance, to localize the target div element in the web page used as example in the Montoto et al. [10] paper, their algorithm generates //td/a[@href="#"]/div[@class="c1" and text()="More Info"]/ while ROBULA generates the following simpler XPath expression //td/a/div/. Choudhary et al. [3] propose WATER, a tool that suggests changes that can be applied to repair test scripts for web applications. ROBULA, on the contrary, aims at creating robust XPath expressions to be used by practitioners in web test suites. Grechanik et al. [4] describe an approach for maintaining and evolving test scripts by means of GUI-tree diffs in order to find altered GUI objects. ROBULA instead works in the context of web applications and interacts with the GUI elements via the DOM structure. Thummalapenta et al. [12] present ATA, a tool to automatically repair test script automatically for certain types of application or environment changes. ROBULA addresses the same web scenario but aims at strengthening the resilience of XPath locators to the web application evolution and it does not consider yet any automatic repair techniques.

VI. Conclusions and Future Work

This work has proposed and experimented ROBULA, a novel algorithm able to partially prevent and thus reduce the aging of web test cases by automatically generating robust web testing-oriented XPath locators. We have compared the robustness of the XPath locators generated by state of the practice XPath generator tools (i.e., absolute and id-based relative locators) with the ones generated by ROBULA. Results indicate that the locators generated by ROBULA are significantly better in terms of robustness than absolute and id-based relative locators.

In our future work, we intend to fine-tune ROBULA and conduct further studies to corroborate our findings. In particular, we would like to: (1) extend the kind of XPath constructs used by ROBULA with, e.g., text(), to locate web elements by means of the text they contain (i.e., an extended version of the LinkText method provided by Selenium WebDriver [6]); (2) adopt a prioritisation strategy (e.g., among different attributes) and a blacklisting strategy (i.e., excluding some attributes that are generally fragile, e.g., href); (3) extend the empirical study with more web applications and more releases; (4) complete the development of a Firefox plugin implementing ROBULA; (5) compare the robustness of the XPath expressions generated by ROBULA with other approaches/techniques, such as Montoto et al.'s [10], and tools, such as Selenium IDE. Although these tools do not provide developers with a stand-alone solution for creating XPath locators, they indeed include some algorithms to create XPaths that are used in their test cases.

References

[1] A. Bruns, A. Kornstadt, and D. Wichmann. Web application tests with Selenium. *IEEE Software*, 26(5):88–91, 2009.

[2] P. Chapman and D. Evans. Automated black-box detection of side-channel vulnerabilities in web applications. In *Proc. of CCS 2011*, pages 263–274, New York, NY, USA, 2011. ACM.

[3] S. R. Choudhary, D. Zhao, H. Versee, and A. Orso. Water: Web application test repair. In *Proc. of ETSE 2011*, pages 24–29. ACM, 2011.

[4] M. Grechanik, Q. Xie, and C. Fu. Maintaining and evolving GUI-directed test scripts. In *Proc. of ICSE 2009*, pages 408–418. IEEE, 2009.

[5] M. Kowalkiewicz, M. E. Orlowska, T. Kaczmarek, and W. Abramowicz. Robust web content extraction. In *Proc. of WWW 2006*, pages 887–888. ACM, 2006.

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

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

[8] M. Leotta, D. Clerissi, F. Ricca, and P. Tonella. Visual vs. DOM-based web locators: An empirical study. In *Proc. of 14th International Conference on Web Engineering (ICWE 2014)*. Springer, 2014.

[9] M. Mirzaaghaei. Automatic test suite evolution. In *Proc. of ESEC/FSE 2011*, pages 396–399. ACM, 2011.

[10] P. Montoto, A. Pan, J. Raposo, F. Bellas, and J. Lopez. Automated browsing in ajax websites. *Data & Knowledge Engineering*, 70(3):269–283, 2011.

[11] G. Rao and A. Pachunoori. Optimized identification techniques using XPath. Technical Report MSU-CSE-00-2, IBM Developerworks, 2013.

[12] 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 *Proc. of ICSE 2013*, pages 1002–1011. IEEE, 2013.

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

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