ROBULA+: An Algorithm for Generating Robust XPath Locators for Web Testing

Maurizio Leotta, Andrea Stocco, Filippo Ricca, Paolo Tonella

Abstract:

Automated test scripts are used with success in many web development projects, so as to automatically verify key functionalities of the web application under test, reveal possible regressions and run a large number of tests in short time. However, the adoption of automated web testing brings advantages but also novel problems, among which the *test code fragility* problem. During the evolution of the web application, existing test code may easily break and testers have to correct it. In the context of automated DOM-based web testing, one of the major costs for evolving the test code is the manual effort necessary to repair broken web page element *locators* – lines of source code identifying the web elements (e.g., form fields, buttons) to interact with.

In this work, we present ROBULA+, a novel algorithm able to generate robust XPath-based locators – locators that are likely to work correctly on new releases of the web application. We compared ROBULA+ with several state of the practice/art XPath locator generator tools/algorithms. Results show that XPath locators produced by ROBULA+ are by far the most robust. Indeed, ROBULA+ reduces the locators fragility on average by 90% w.r.t. absolute locators and by 63% w.r.t. Selenium IDE locators.

Digital Object Identifier (DOI):

https://doi.org/10.1002/smr.1771

JOURNAL OF SOFTWARE: EVOLUTION AND PROCESS
J. Softw. Evol. and Proc. 2016; **00**:–
Published online in Wiley InterScience (www.interscience.wiley.com).

ROBULA+: An Algorithm for Generating Robust XPath Locators for Web Testing

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*

SUMMARY

Automated test scripts are used with success in many web development projects, so as to automatically verify key functionalities of the web application under test, reveal possible regressions and run a large number of tests in short time. However, the adoption of automated web testing brings advantages but also novel problems, among which the *test code fragility* problem. During the evolution of the web application, existing test code may easily break and testers have to correct it. In the context of automated DOM-based web testing, one of the major costs for evolving the test code is the manual effort necessary to repair broken web page element *locators* – lines of source code identifying the web elements (e.g., form fields, buttons) to interact with.
In this work, we present ROBULA+, a novel algorithm able to generate robust XPath-based locators – locators that are likely to work correctly on new releases of the web application. We compared ROBULA+ with several state of the practice/art XPath locator generator tools/algorithms. Results show that XPath locators produced by ROBULA+ are by far the most robust. Indeed, ROBULA+ reduces the locators fragility on average by 90% w.r.t. absolute locators and by 63% w.r.t. Selenium IDE locators.

Received . . .

KEY WORDS: Web Testing; Test Cases Fragility; Robust XPath Locator; Maintenance Effort Reduction; DOM Selector.

1. INTRODUCTION

Modern web applications are developed at a fast rate to accommodate new functionalities and security/bug fixes, as well as 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 [1] only if it is strongly supported by automated tools, which reduce the effort required from web testers for test suite execution. The fast evolution of the web applications under test requires to evolve continuously the test suites that accompany a web application, 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 [3], even if other interesting proposals are emerging [4, 5, 6]. They offer web testers a rich programmable API that can be used to define DOM-based test scripts. Using this API, web testers can for instance *locate* an input text field, fill it with some text, *locate* a button, click on it, *locate* the text that shows the result of the computation and check whether it matches

* Correspondence to: Maurizio Leotta - Dipartimento di Informatica, Bioingegneria, Robotica e Ingegneria dei Sistemi (DIBRIS), Università di Genova, Genova, Italy.
E-mail: maurizio.leotta@unige.it

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 [7].

Test automation brings several benefits, but also challenges, among which the maintenance of test scripts during software evolution. Such maintenance consists mainly of manual repair of the locator instructions. In fact, test scripts heavily rely on locators, to interact with the elements on the web page – 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). Locators need to be checked for correctness and possibly updated at every new release of the software. Sometimes even a slight modification of the application under test has a massive impact on locators. This problem makes the maintenance of web test suites extremely expensive.

Among the web element locators, 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 browsers' add-ons (e.g., FirePath, XPath Checker, XPath Helper) exist which compute a candidate XPath locator for the web tester.

> 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 [8] and even minor modifications of the DOM structure may cause their failure, so that web testers have to correct them when the application evolves.

In this paper, we propose a novel algorithm, called ROBULA+ (ROBUst Locator Algorithm), a refined version of ROBULA, the algorithm we proposed in a previous work [9], able to automatically generate robust web element locators. The algorithm starts with a generic XPath locator that returns all nodes ("//*") and then it iteratively refines the locator until only the element of interest is selected. In such iterative refinement, ROBULA+ applies seven refinement transformations, according to a set of heuristic XPath specialisation steps. Moreover, ROBULA+ makes use of prioritisation and black listing techniques. The former is used to rank candidate XPath expressions in terms of expected robustness, while the latter to exclude attributes that are considered intrinsically fragile.

Overall, for the eight web applications considered in our experiment, ROBULA+ has generated locators that are much more robust than those produced by both state of the practice/art tools (FirePath* and Selenium IDE†) and algorithms such as Montoto [10]. Moreover, ROBULA+ has also drastically improved the results w.r.t. its predecessor ROBULA [9].
This paper makes the following contributions:

- ROBULA+, a novel algorithm for the automatic creation of robust XPath locators. Our approach extends our previous proposal (ROBULA) in several directions: (1) prioritization of attributes by recognized capability of creating robust XPath locators, and conversely (2) exclusion of attributes that often lead to the creation of fragile XPath locators (i.e., black listing mechanism), (3) usage of the text contained in the DOM nodes for creating XPath locators, (4) usage of multiple attributes for the same node in the XPath locators, (5) ability of creating locators independently from the specific DOM element type (i.e., using "*" to leave the tag name unspecified);
- two implementations of our algorithm. One in Java used for the empirical evaluation reported in this paper. Further, we implemented ROBULA+ as an open source plug-in for Firefox, ready to be used by web testers;
- an empirical evaluation over more than a thousand of web elements from eight real size web applications, assessing the effectiveness and efficiency of ROBULA+, in comparison with the existing state of the practice/art tools/algorithms and with its predecessor ROBULA.

* https://addons.mozilla.org/firefox/addon/firepath/ † http://docs.seleniumhq.org/projects/ide/

Unlike many solutions currently proposed by the research community (e.g., ATA [6]), the major strength of our proposal is that it can be adopted at virtually no additional cost by any web tester developing DOM based test code. Indeed, ROBULA+ does not require to replace the testing tool or framework (e.g., Selenium WebDriver[‡]) used by web testers with a specific one. A web tester who wants to use ROBULA+ in a real industrial context has only to replace the locator generator, usually a browser add-on like FirePath, with the one implementing our algorithm (e.g., by using the Firefox add-on we developed). With this simple change, it is possible to produce more robust locators and thus to reduce the effort and cost needed for repairing them. Indeed, in our experience repairing the broken locators is the major cost factor during web testware evolution [11].

The rest of the paper is organized as follows: Section 2 introduces the problems associated with web testware evolution and discusses the way locators are produced and evolved. Section 3 describes our novel contribution: ROBULA+. The empirical study conducted for evaluating the robustness of the locators produced by ROBULA+ and its efficiency is reported in Section 4. Section 5 reports the related work. Finally, Section 6 summarizes our conclusions and future work.

2. 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 even if some preliminary approaches have been proposed (e.g., [12, 13]).

A software tester executes test case repair actions according to the maintenance task that has been performed on the web application under test (WAUT). The changes to the WAUT can be categorized into two families: logical and structural [11]. A logical change involves the modification of the web application logic pursuant to the introduction of new features or the modification of existing features. On the tester side, this means adjusting the test suite, inserting, deleting or modifying (i.e., changing the scenario) one or more test cases. A structural change impacts only the page layout/structure, modified to beautify the web page appearance or to reorganize its content (e.g., switching from a table-based to a tableless layout). In the test suite, the tester has to modify one or more 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 (i.e., changes impacting the page layout/structure) since such effort depends heavily on the robustness of the locators. On the other hand, logical changes (i.e., changes modifying the logic of the web application) may require manual interventions on the test suite that go beyond the creation of robust locators. Structural changes are indeed quite important. For instance, web site re-styling, a frequently occurring activity [11], 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 when the WAUT evolves.

2.1. 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 [11]:

1. *Coordinate-based locators:* first generation tools just record the screen coordinates of web page elements and use this information to locate such elements during test case replay. This approach is nowadays considered obsolete, because it produces extremely fragile locators. In fact, these locators could break even under imperceptible changes of the web page layout.

[‡] http://docs.seleniumhq.org/projects/webdriver/

2. *DOM-based locators:* second generation tools locate web page elements using the information contained in the Document Object Model (DOM). For example, the tools Selenium IDE and Selenium WebDriver employ this approach and offer different ways to locate web page elements. DOM-based tools have reached a high level of maturity and are widely used both in academia and industry [3, 2, 14, 15, 16].
3. *Visual locators:* third generation tools have emerged in the last years. They make use of image recognition techniques to identify and control GUI components. The tool Sikuli[§] [4] belongs to this category.

In this paper, we focus on DOM-based localization, since (1) it is the most adopted technology in practice [2], (2) DOM-based locators are generally the most robust [11], albeit, in some cases visual locators might be the best choice – e.g., with applications having complex visual components, such as Google Maps. Using the information contained in the DOM, existing tools provide several ways to locate web page elements. For instance, Selenium WebDriver locates a web page element using: (1) the values of attributes id, name, and class; (2) the tag name of the element; (3) the text string shown in the hyperlink, for anchor elements; (4) CSS and (5) XPath expressions. Not all these locators are applicable to any arbitrary web element; e.g., locator (1) can be used only if the target element has a unique value of attribute id, name, or class in the entire web page; locator (2) can be used if there is only one element with the chosen tag name in the whole page; and, locator (3) can be used only for links uniquely identified by their text string. On the other hand, XPath/CSS expressions can always be used. In fact, as a baseline, the unique path from root to target element in the DOM tree can always be turned into a (quite fragile) XPath/CSS locator that uniquely identifies the element. In this paper, we focus on the robustness of XPath locators, leaving CSS locators for future work.

2.2. *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, as discussed before, are applicable only to specific cases (e.g., By.id is not applicable when the identifier is not present). With XPath expressions it is always possible to locate every web page element. In our previous work [8], we developed six Selenium WebDriver and IDE test suites for six different web applications. In these test suites, whenever possible we used Selenium specific localisation methods (e.g., the WebDriver commands By.id, By.name). In such study, we found that 1587 over a total of 2735 locators use such specific localisation methods. The remaining are XPath and CSS-based locators (respectively 791 (29%) and 357 (13%) over 2735 locators). It should be noticed that all the locators that do not make use of XPaths can be easily rewritten as XPath locators with no substantial impact on their understandability. We found similar XPath locator occurrences in an open source test suite[¶] created for a real web application, *Moodle*, where 102 XPath and 43 CSS expressions over 254 locators are used (i.e., respectively 40% and 16% of the total).
3. **XPath locators are generally considered fragile, but this strongly depends on how they are created.** The common belief on the fragility of XPath locators largely depends on how XPath locators are generated by tools. In our previous work [8], we found that: 58% of the 791 XPath locators were broken from a release to a subsequent one, while for the other types of locators the breakage percentages were extremely lower (e.g., less than 2% for the id locators; about 12% for LinkText and 18% for the Name locators). By inspecting the XPath locators, most of which automatically generated by FirePath[‖], we realised that their quality, from the

[§] http://www.sikuli.org/

[¶] https://github.com/moodlehq/functional-test-suite

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

robustness point of view, was largely sub-optimal and that better XPath locators could be defined manually. However, careful manual definition of robust XPath locators requires a lot of experience and a big effort.

2.3. Generating XPath Locators in Practice

Currently there exist several tools that a web tester can use to build XPath expressions. In web testing, it is usually preferable to have a browser-integrated add-on. Among the Mozilla Firefox Add-Ons, FirePath and XPath Checker[**] are the most popular[††]. FirePath is among the most downloaded and used in the practice. It is a Firebug extension that adds a development tool to edit, inspect and generate XPath expressions based on the information contained in the DOM structure. Such expressions include the full path from the root (e.g. /html/body/div[2]/div/a) or make use of id-based expressions (e.g., //div[@id="content"]/a), if an id attribute is available. This creation strategy is quite simplistic to be effective during software evolution, because in real world applications the DOM structure changes frequently. Therefore, FirePath expressions tend to be not resilient even to small changes of the page, resulting in fragile test scripts [9]. XPath Checker is also quite used, but it provides expressions similar to FirePath ones. Among the others, we considered the Google Chrome add-ons: the built-in add-ons and XPath Helper[‡‡]. Chrome built-in add-ons' expressiveness is similar to the Mozilla one, since they produce absolute paths or id-based relative expressions. XPath Helper merges the two approaches, providing absolute expressions enriched with an attribute (if any is available) for every tag of the expression. Our tool ROBULA+ offers a much more reliable solution to robust locator creation (see empirical results in Section 4) and its implementation as a Firefox add-on makes it quite easy to adopt in practice.

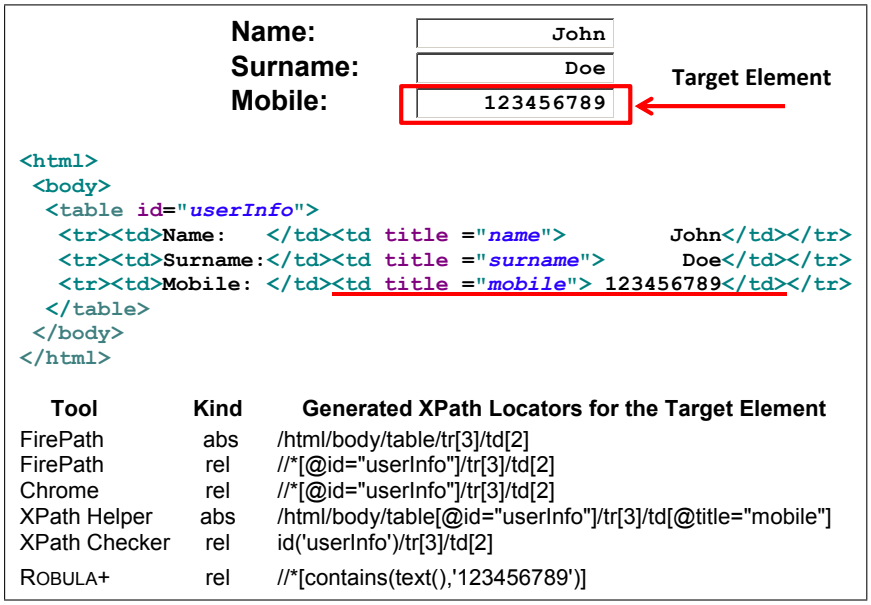
2.4. Are XPath Locators so Fragile?

Despite the common belief, it is not true that XPath locators are generally more fragile than the other kinds of locators, because the expressiveness of the XPath language permits to simulate almost all the other DOM-based localization methods, thus achieving at least the same fragility.

XPath locators are often adopted when the other methods cannot be employed. In these cases, web testers generally rely on sub-optimal XPath locators (from the robustness point of view), provided by the tools mentioned in Section 2.3 (e.g., absolute XPath or id-based relative XPath expressions). On the other hand, building robust locators manually requires strong knowledge of the XPath language and involves a time consuming analysis of the web page structure. For this reason, XPath locators are often perceived, by web testers, as the most fragile kind of locators.

In an industrial case study [17], we found absolute XPath locators to be very fragile since they contain the entire specification of how to traverse the DOM tree, from the root to the target element. The tools mentioned above could generate more robust relative XPath expressions, by relying on ids that are unique in the web page. Indeed, by W3C standards, ids should be unique in the page. "ids are often the safest locator option and should always be considered as the first choice"[§§]. In some cases ids may be less robust, especially if they are automatically re-generated at every new release. In this case their robustness depends on the adopted id-generation algorithm. However, in our experience [17], automatically generated ids have shown a low level of fragility. The robustness of relative locators anchored on ids varies, depending on the position of the first id in the ancestors' chain starting from the target element. If the first id is in the target node itself, the associated XPath locator is probably very robust, because the developer intentionally put a meaningful id that probably is likely to be the same in subsequent releases of the application. If the first id is near to the root of the DOM, the relative XPath locator from such id to the target node becomes very similar to the absolute XPath, thus exhibiting low robustness.

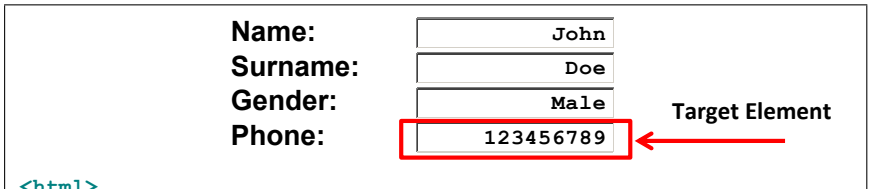
[**] https://addons.mozilla.org/firefox/addon/xpath-checker/ [††] http://docs.seleniumhq.org/docs/02_selenium_ide.jsp#locating-elements
[‡‡] https://chrome.google.com/webstore/detail/xpath-helper/hgimnogjllphhhkhlmebbmlgjoejdpjl
[§§] http://blog.mozilla.org/webqa/2013/09/26/writing-reliable-locators-for-selenium-and-webdriver-tests/




```
<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	/html/body/table/tr[3]/td[2]
FirePath	rel	//*[@id="userInfo"]/tr[3]/td[2]
Chrome	rel	//*[@id="userInfo"]/tr[3]/td[2]
XPath Helper	abs	/html/body/table[@id="userInfo"]/tr[3]/td[@title="mobile"]
XPath Checker	rel	id('userInfo')/tr[3]/td[2]
ROBULA+	rel	//*[contains(text(),'123456789')]

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




```
<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	✗	/html/body/table/tr[~~3~~→4]/td[2]
FirePath	✗	//*[@id="userInfo"]/tr[~~3~~→4]/td[2]
Chrome	✗	//*[@id="userInfo"]/tr[~~3~~→4]/td[2]
XPath Helper	✗	/html/body/table[@id="userInfo"]/tr[~~3~~→4]/td[@title="mobile"]
XPath Checker	✗	id('userInfo')/tr[~~3~~→4]/td[2]
ROBULA+	✓	//*[contains(text(),'123456789')]

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

In this work, we adopt XPath locators for their high expressivity and we aim at automatically building XPath locators that are ideally as robust as the best locators that web testers can possibly choose among the available options.

2.5. 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 Figure 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 the mobile phone number (see the underlined td in Figure 1 (center)). With Selenium WebDriver, the 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 build 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 2.3. Figure 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, in which the user is allowed to insert gender information (see Figure 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 Figure 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, albeit 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 or thousands of tags. Often, in these complex cases ROBULA+ finds instantly locators that make use of complex combinations of attributes (e.g., //*[@class="row-2"]/td[@class="center"]).

3. ROBUST LOCALIZATION OF WEB PAGE ELEMENTS

In this section, we describe our Robust Locator Algorithm, ROBULA+, for the automated generation of robust XPath locators. It consists of an evolution of ROBULA [9], which enhances our previous algorithm with several improvements: *(i)* the adoption of a prioritisation strategy, to rank candidate XPath expressions by heuristically estimated attribute robustness, useful when multiple attributes are available for a DOM element; *(ii)* the adoption of a blacklisting technique, to exclude attributes that are recognized as intrinsically fragile; *(iii)* the usage of textual information as additional information for building the predicates composing the XPath locators – the text can represent a potentially reliable anchor when the web application evolves; *(iv)* the usage of multiple attributes for the same node in the XPath locators; *(v)* the ability of creating locators independently from the specific DOM element type (i.e., using "*" to leave the tag name unspecified).

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.

3.1. Specialisation Transformations for XPath Expressions

ROBULA+ specialises an XPath expression applying seven transformations according to an established order: transfConvertStar, transfAddID, transfAddText, transfAddAttribute, transfAddAttributeSet, transfAddPosition, and transfAddLevel. The transformations work as shown in Figure 3. All the transformations are only applied at the head of the XPath expression (i.e.,

Let:
– **xp** = the XPath expression to specialize, e.g., //td
– **N** = the length (in nodes/levels) of **xp**
e.g., //td ⇒ **N**=1; //*/td ⇒ **N**=2; //table/tr/td ⇒ **N**=3;
– **L** = the list of the ancestors of target element **e** in the considered DOM (i.e., web page), starting and including **e**
e.g., given **L**=[td,tr,table,body,html] the call **L**.get(2) returns the element tr

The transformations work as follows:
– **transfConvertStar**
Precondition: the XPath **xp** starts with //*
Action: replace the initial * with the tag name of the DOM element **L**.get(**N**)
Example: **xp** = //*/td and **L**.get(**2**).getTagName() = tr *output*: //tr/td
Note: if the unspecified tag name "*" is already constrained by a position value (//*[2]), the transformation updates also, if necessary, the position value (e.g., //*[2]/b → //div[1]/b). This happens when an element has preceding siblings of different types.
– **transfAddID**
Precondition: the **N**th level of **xp** does not contain already any kind of predicates
Action: add the predicate based on the id (if available) of the DOM element **L**.get(**N**) to the higher level of **xp**
Example: **xp** = //td and **L**.get(**1**).getID() = 'name' *output*: //td[@id='name']
– **transfAddText**
Precondition: the **N**th level of **xp** does not contain already any predicate on text or any predicate on position
Action: add a predicate on the text contained (if any) in the DOM element **L**.get(**N**) to the higher level of **xp**
Example: **xp** = //td and **L**.get(**1**).getText() = 'John' *output*: //td[contains(text(),'John')]
– **transfAddAttribute**
Precondition: the **N**th level of **xp** does not contain already any kind of predicates
Action: for each available attribute-value pair of the DOM element **L**.get(**N**), generate a candidate locator by adding a predicate based on such value to the higher level of **xp**
Example: **xp** = //tr/td and **L**.get(**2**).getAttributes() = {name='data', class='table-row'}
output: //tr[@name='data']/td, and //tr[@class='table-row']/td
– **transfAddAttributeSet**
Precondition: the **N**th level of **xp** does not contain already any kind of predicates
Action: for each element (with cardinality >1) of the powerset generated from the set of all the attribute-value pairs of the DOM element **L**.get(**N**), generate a candidate locator by adding a predicate based on such element to the higher level of **xp**
Example: **xp** = //tr/td and **L**.get(**2**).getAttributes() = {name='data', class='table-row'}
output: //tr[@name='data' and @class='table-row']/td
– **transfAddPosition**
Precondition: the **N**th level of **xp** does not contain already any predicate on position
Action: add the position of the element **L**.get(**N**) to the higher level of **xp**
Example: **xp** = //tr/td and **L**.get(**2**).getPosition() = {if tag-name=2, if '*'=3} *output*: //tr[2]/td
Note: the transformation considers that the position of the element **L**.get(**N**) that is added in the XPath candidate locator **xp**, could change if such element is selected using a * or the actual tag name. This happens when an element has preceding siblings of different types.
– **transfAddLevel**
Precondition: **N** < **L**.lenght()
Action: add //* at the top of **xp**
Example: **xp** = //tr/td *output*: //*/tr/td

Figure 3. Specialization transformations used by ROBULA+

they work at the higher level of the XPaths). transfConvertStar introduces a specific tag name to replace the wildcard "*" in the XPath expression. transfAddID, transfAddText, transfAddAttribute and transfAddPosition add predicates to the XPath expression, by respectively considering only DOM elements containing specific id values, text values, attribute-value pairs or occupying specific positions among their siblings. transfAddAttributeSet adds elements of the powerset generated from the set of attributes of each DOM element. transfAddLevel adds a new level to the current XPath expression by extending it with the wildcard tag "*" added at the beginning.

Some considerations about the transformations reported in Figure 3 are as follows:

- As already outlined previously, all the seven transformations are applied only to the last level of the XPaths (i.e., on the portion of the XPath next to the initial "//"). On the one hand, this constraint allows to reduce the number of applicable specializations to each XPath expression, with clear advantages on the execution time. On the other hand, this constraint, is not restrictive in terms of generated XPaths. In fact, by means of subsequent applications of the various transformations it still ensures the generation of all possible XPaths for the selected target element. In practice, this constraint avoids to execute many derivations leading to the same XPath by modifying the lower levels of the specializing expression. For instance, with this constraint the XPath //tr/td[2] cannot be generated by means of the following steps //* $\rightarrow_1$ //*/* $\rightarrow_2$ //tr/* $\rightarrow_3$ //tr/*[2] $\rightarrow_4$ //tr/td[2] which would require to apply the transformations also to the lower level of the XPaths (look at the transformation steps 3 and 4 that are executed on the tail of the XPaths). However, the same XPath can be produced by ROBULA+ by applying transformations only to the head of the XPath expression (e.g., //* $\rightarrow_1$ //td $\rightarrow_2$ //td[2] $\rightarrow_3$ //*/td[2] $\rightarrow_4$ //tr/td[2]).
- The goal of the transformations' preconditions is threefold: (1) avoid to add multiple times the same predicate; (2) avoid inadmissible transformations; and, (3) avoid to generate different XPaths having exactly the same behaviour, due to the presence of different permutations of the predicates they contain (e.g., the XPath //td[contains(text(),'John')][2]is equivalent to //td[2][contains(text(),'John')]).
- The transformations shown in Figure 3 generate only XPath expressions that locate the target element, since they add only constraints derived from information contained in the target element or its ancestors.

3.2. The Algorithm

The pseudocode of ROBULA+ is shown in Figure 4. The algorithm takes in input a document d (i.e., an HTML page) and an absolute XPath abs selecting only the target web page element e (e.g., an anchor, a text field, a button, etc.). For this web element, the algorithm returns x (line 18), a robust relative XPath expression (if anyone exists) able to uniquely select the target element, i.e., a *robust*

```
 1.  XPath ROBULA+(XPath abs, Document d)
 2.  {
 3.    Element e = eval(abs, d);
 4.    List<XPath> xpList = ["//*"];
 5.    while (true)
 6.    {
 7.       XPath xp = xpList.removeFirst();
 8.       List<XPath> temp = [];
 9.       temp.append(transfConvertStar(xp));
10.       temp.append(transfAddID(xp));
11.       temp.append(transfAddText(xp));
12.       temp.appendAll(transfAddAttribute(xp));
13.       temp.appendAll(transfAddAttributeSet(xp));
14.       temp.append(transfAddPosition(xp));
15.       temp.append(transfAddLevel(xp));
16.       for (XPath x : temp)
17.       {
18.          if (uniquelyLocate(x, e, d)) return x;
19.          else xpList.append(x);
20.       }
21.    }
22.  }

List<XPath> eval(XPath abs, Document d):
       returns the element in d selected by the XPath locator abs

Boolean uniquelyLocate(XPath x, Element e, Document d):
       TRUE iff eval(x, d) contains only e
```

Figure 4. Pseudocode of ROBULA+

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 initializing the list xpList of XPath expressions with the most general one (i.e., "//*") (line 4).

Then, it iterates (line 5) until a result (i.e., an XPath locator) is found (line 18). At each cycle, it removes the first XPath expression (xp) from the list xpList and it applies, in the established order, seven transformations (transfConvertStar, transfAddID, transfAddText, transfAddAttribute, transfAddAttributeSet, transfAddPosition, and transfAddLevel) to specialize xp.

All the XPath expressions generated by applying these transformations are inserted into a list named temp. At this point (line 16), the algorithm cycles through the XPath expressions contained in temp, considering in turn each XPath expression x. If x is a unique locator for the target element e (the function uniquelyLocate is used to determine this), the algorithm returns it and terminates. Otherwise, by construction we know that x selects more elements than desired, among which the target one is included. Thus, x is inserted into the list xpList, to be specialised in the next iterations 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 actual implementation of ROBULA+ employs a data-structure containing all the useful information (e.g., tag name, attributes, positions) of the target element and its ancestors up to the root node. During the creation of such data-structure, for each element it computes also the power-set of its attributes.

Since ROBULA+ returns the first locator found (i.e., the one that is considered as the most robust), it is very important to consider (1) the order of execution of the transformations; and, (2) the order in which each transformation returns its results (this second aspect will be considered in Section 3.4). At each iteration of the main cycle (line 5), all the transformations could potentially be executed and thus an order of execution must be defined.

We decided to choose as first transformation transfConvertStar, since a locator specifying only the tag name of the target element can be considered the simplest and most essential form of locator (e.g., //td). Then, transfAddID is executed, because the values of the id attributes are usually very robust (e.g., id="result"). Indeed, as already mentioned in Section 2.2, during a previous study [8] where we built several Selenium test suites containing 2735 locators, we found that id locators are the most robust, with less than the 2% of the 459 used id locators broken. Moreover in another work [17] we found that even locators using auto-generated ids are quite robust. The text property (e.g., the text of a link) is also commonly perceived as quite robust and in our previous study [8] we found that only the 12% of the 473 LinkText locators were broken. Their robustness is lower only to the id locators' one. Thus, in ROBULA+, the transformation transfAddText is executed after transfAddID. Then transfAddAttribute is executed since attribute values are usually more robust than position values. Indeed, attribute values usually include meaningful names (e.g., name="username"), while position values are always bound to the web page structure and so they are the most fragile [18]. It should be noticed also that transfAddAttributeSet is executed before transfAddPosition, since relying on a combination of attributes is expected to provide better robustness than relying on positions values. We execute transfAddPosition and then transfAddLevel, 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 is expected to be more robust).

3.3. Algorithm's Analysis

The algorithm is ensured to terminate, since in the worst case 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 Figure 3, transfConvertStar can be used to add the tag name and transfAddPosition 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 complexity of ROBULA+ is exponential in the number of predicates (i.e., attributes, text and position) and levels that can be found between the target element and the root of the DOM. This is due to the fact that, in the worst case, ROBULA+ has to generate virtually all the XPaths of length from 1 to h, with h the length of the absolute XPath (i.e., the distance between the target web element and the root of the DOM).

In detail, given a target element e, let X_e be the set of all the XPaths that can be generated for e by repeatedly applying the transformations shown in Figure 3, and thus by ROBULA+. Each XPath in X_e can be a locator for e or simply include e among the web elements it selects. The cardinality of X_e depends on: (1) h the number of levels between e and the root of the DOM (i.e., the length of the full absolute locator for e); and, (2) the cardinality of each P_i, the set of predicates that can be applied at the i-th level using the information available for the i-th ancestor of e. More specifically:

$$|X_e| = \sum_{i=1}^{h} 2^{(\sum_{k=1}^{i} |P_k|)+i} \tag{1}$$

As an example, let us consider the following DOM of a simplified HTML page:

```
<html>
  <p class='a'>X</p>    <------ Target Web Element e
  <p class='a'>Y</p>
  <div class='a'>X</div>
</html>
```

In this case, for the target element <p> we have a total number of levels $h = 2$ (i.e., <p> and <html>). Counting backward from <p>, these two levels are associated with two sets of predicates: P_1={text()='X', @class='a', position=1} and P_2={position=1}. The XPath expressions that contain only one level start either with //* or //p. They are completed with the addition of a subset of the predicates in P_1 (including the case of the empty set). Since there are 2 possible starting XPaths, completed by any of the $2^{|P_1|}$ subsets of P_1, in total there are $2 \cdot 2^{|P_1|} = 2^{|P_1|+1} = 2^{3+1} = 2^4 = 16$ possible XPaths one level long. The XPath expressions that contain two levels start with //* or //html, possibly completed with the addition of any subset of predicates from P_2, and followed by the next level, starting with /* or /p and completed with any subset of predicates from P_1. Hence, in total, there are $2 \cdot 2^{|P_2|} \cdot 2 \cdot 2^{|P_1|} = 2^{(|P_1|+|P_2|)+2} = 2^{(3+1)+2} = 2^6 = 64$ XPaths two level long. In total, $|X_e| = 16 + 64 = 80$. Although the size of X_e grows exponentially with h and with the number of available predicates, the heuristics introduced in the algorithm (i.e., the order of execution of transformations, as well as prioritisation, black list, discussed below) aim at making the execution time acceptable in practical cases, as confirmed by the results presented in Section 4.

3.4. Attribute Prioritization and Black List

The order in which a transformation provides its results affects the final output of ROBULA+. This is true in the case of a transformation that can return more than one results, such as transfAddAttribute. In fact, if we have more than one attribute (e.g., name="username" and width="210px") for the DOM element matching the beginning of the XPath expression (e.g., //input) and transfAddAttribute is executed, we obtain more than one candidate locator for the target element (e.g., //input[@name="username"] and //input[@width="210px"]). In the original version of ROBULA [9], we simply inserted the attribute values in the XPath expression in the same order as they were found in the HTML source code after the associated tag. We discovered that this is not always effective, because some attributes are more relevant than others if we want to locate a web element robustly for testing purposes. For instance, the name of an input field is probably more robust than its width. For this reason, transfAddAttribute and transfAddAttributeSet implement a (1) *prioritization* (i.e., they prioritize the attributes by expected robustness) and (2) *black-listing* (i.e., they discard the attributes considered too fragile).

Prioritization: we manually defined a prioritized list of HTML attributes, based on our previous experience in test suite construction for web testing [8, 17]. Thus, transfAddAttribute tries, first of all, to use predicates containing the name attribute, which is regarded as the most robust after id. Then, the transformation tries with the following attributes, that usually contain meaningful values: class, used to specify a class name for a web element; title, used to specify extra information about a web element, usually shown as a tooltip text (e.g., for an image) when the mouse moves over the element; alt, used to specify an alternate text for an image; value, used for instance, to define the text shown on submit buttons or the initial (default) value of input fields. Thus, the prioritized list of HTML attributes contains (name, class, title, alt, value). Note that the id attribute has the highest priority since it is inserted first, by the dedicated transformation transfAddID and thus is not included here. Then, transfAddAttribute uses the attributes not included in the prioritized list in the same order as they are found in the DOM. When generating sets of attributes, transfAddAttributeSet inserts first the sets containing the highest priority attributes (in this case including id) as predicates in the candidate XPath expressions, starting from the smaller sets (with cardinality equal to 2) till to the complete set of all the attributes of the DOM element. For each cardinality, the transformation inserts first the sets composed of attributes with higher priority.

Black List: in our previous work [8] we discovered that some attributes hinder the possibility of generating robust locators. For instance, the following XPath locator:
//a[@href="DIR_Path/list.php?cmd=exLock&profile_id=1&offset=0"]
is inherently not robust, since it contains a path in the file system structure for the current release of the application, which may change in a subsequent release. For this reason, we decided to define a *black-list*, i.e., a list of attributes that are not considered during the execution of ROBULA+. In the black list we have inserted: (1) "src" and "href", because they often refer to the directory structure of the web application, which is typically variable across releases; (2) all attributes containing JavaScript code, such as "onclick" and "onload", since the included JavaScript code tends to be quite volatile across releases and hence cannot be used as a locator; (3) "tabindex", since it specifies the tab order of an element (used for navigating among web elements), which is very likely to change if another element is inserted before the one to be located; (4) the attributes that are used for specifying the dimensions of a web element, such as "width", "height", "size", "maxlength"; and finally, (5) "style", used to specify an inline style sheet for an element. Thus, the black list of HTML attributes contains (src, href, onclick, onload, tabindex, width, height, size, maxlength, style).

3.5. Algorithm's Implementations

We developed two implementations of ROBULA+: one as a standalone Java program and another as a Firefox Add-on. For the Java version, we used the JSoup[¶¶] library for generating an equivalent XHTML code for the HTML web pages taken in input, and the JDOM[***] library for manipulating XHTML documents and evaluate XPath expressions on them. This Java implementation has been used for evaluating the effectiveness of ROBULA+ (Section 4), since it can generate (in batch mode) XPath locators for hundreds of web elements, as required for our study.

The equivalent JavaScript version of ROBULA+ has been developed in order to provide such algorithm also as a Firefox add-on (see Figure 5). It can be used by web testers to generate locators during the development of test suites in real industrial contexts. Starting from the source code of the existing Firefox add-on FirePath, we replaced the XPath locator generation engine (absolute and id-based) with our JavaScript implementation of ROBULA+. It allows users to experiment with various configurations of our algorithm, by enabling or disabling the key features of ROBULA+ (e.g., it is possible to compare the produced XPaths by enabling or disabling the blacklisting strategy). Varying the algorithm configuration can be particularly useful when the tester is aware that a certain locator generation strategy might be not appropriate to obtaining robust locators. For instance, suppose that the test has to locate several target web elements having textual values which are dynamic or vary across different versions (e.g., the total of a computation, or dates and timestamps). In these

[¶¶] http://jsoup.org/ [***] http://www.jdom.org/

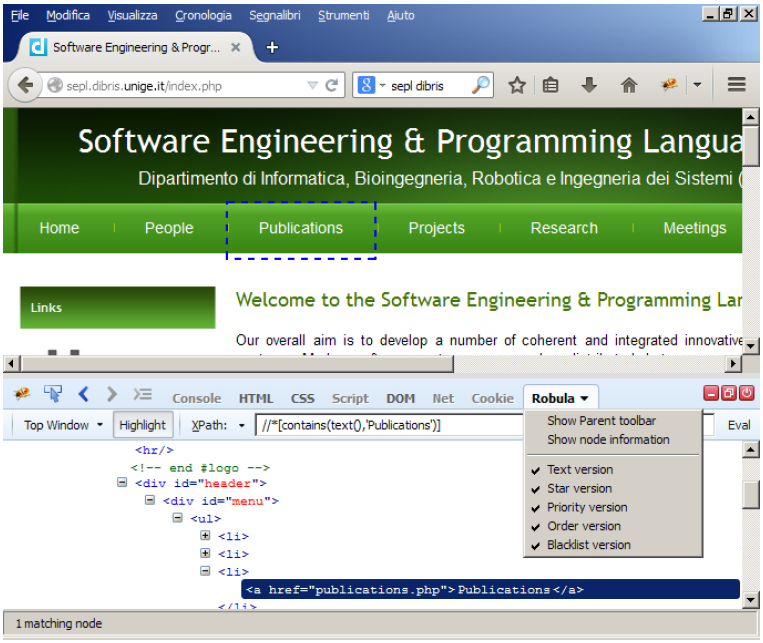


Figure 5. The Firefox Add-on implementing ROBULA+

situations, the tester can disable the capability of creating locators based on such changing texts, letting the algorithm find another kind of locator.

For the interested reader, our implementations of ROBULA+ can be found in our website: http://sepl.dibris.unige.it/ROBULA.php.

4. EXPERIMENTAL RESULTS

This section describes the design, experimental objects, research questions, metrics, procedure, results, discussion and threats to validity of the empirical study conducted to evaluate the robustness of the XPath locators generated by ROBULA+. We follow the guidelines by Wohlin *et al.* [19] on designing and reporting empirical studies in software engineering.

4.1. Study Design

The main *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 *perspective* of (1) testers and project/quality assurance managers, interested in data about the benefits of adopting ROBULA+ as locator generator in an industrial context, and (2) researchers, interested in empirical data about the impact of ROBULA+ on web testware evolution. The *software objects*, used to experiment ROBULA+, are eight open source web applications, some of them already used in a different work [11].

4.2. Web Applications

We conducted our experiments over a sample of eight 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 releases of Apache, PHP and MySQL, technologies we are familiar with (since the XPath locators localize 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

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

	Description	Web Site
MantisBT	bug tracking system	http://sourceforge.net/projects/mantisbt/
PPMA[c]	password manager	http://sourceforge.net/projects/ppma/
Claroline	collaborative learning environment	http://sourceforge.net/projects/claroline/
Address Book	address/phone book, contact manager, organizer	http://sourceforge.net/projects/php-addressbook/
MRBS	system for multi-site booking of meeting rooms	http://sourceforge.net/projects/mrbs/
Collabtive	collaboration software	http://sourceforge.net/projects/collabtive/
TikiWiki	Wiki-CMS-Groupware solution	http://sourceforge.net/projects/tikiwiki/
OrangeHRM	HR management system	http://sourceforge.net/projects/orangehrm/

	1st Release				2nd Release			
	Release	**Date**	**File**[a]	**kLOC**[b]	**Release**	**Date**	**File**[a]	**kLOC**[b]
MantisBT	1.1.8	Jun-09	492	90	1.2.0	Feb-10	733	115
PPMA[c]	0.2	Mar-11	93	4	0.3.5.1	Jan-13	108	5
Claroline	1.10.7	Dec-11	840	277	1.11.5	Feb-13	835	285
Address Book	4.0	Jun-09	46	4	8.2.5	Nov-12	239	30
MRBS	1.2.6.1	Jan-08	63	9	1.4.9	Oct-12	128	27
Collabtive	0.65	Aug-10	148	68	1.0	Mar-13	151	73
TikiWiki	6.0	Dec-10	4841	705	12.0	Dec-13	6322	873
OrangeHRM	2.7	May-12	3019	232	3.1.3	Sep-14	3033	207

[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 - http://www.yiiframework.com/)

releases 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 2014) and different in terms of number of source files (ranging from 46 to 6322) and number of lines of code (ranging from 4 kLOC to 873 kLOC, considering only the lines of code contained in the PHP source files, comments and blank lines excluded).

In the following, we report a short description for each selected application.

- *MantisBT* is a bug tracking system. Over time it has matured and gained a lot of popularity, and now it has become one of the most popular open source bug tracking systems.
- *PHP Password Manager (PPMA)* is a Web based password manager. Each password is (DES-)encrypted with an individual user password.
- *Claroline* is an open source collaborative learning environment allowing teachers or education institutions to create and administer courses through the Web.
- *PHP Address Book* is a simple, Web-based address and phone book, contact manager, and organiser.
- *Meeting Room Booking System (MRBS)* is a system for multi-site booking of meeting rooms. Rooms are grouped by building/area and shown in a side-by-side view. Although MRBS was initially designed as rooms booking system, it can be used to manage any resource.
- *Collabtive* is a collaboration software. It enables the members of geographically scattered teams to collaboratively work using to-do lists, milestones and shared files. Moreover, it tracks the worked time on a task-by-task basis.
- *TikiWiki* is an open source Wiki-CMS-Groupware solution. It has been actively developed since 2002 and more than 250 people have contributed source code to the Tiki project.
- *OrangeHRM Open Source* is a free human resource management system that offers several modules to suit company needs related to its employees.

4.3. Algorithms/Tools for Generating XPath Locators

In this work, we compare the robustness of the XPath locators generated by ROBULA+ with that of the locators generated by five state of the practice tools/research algorithms: FirePath absolute,

FirePath relative, Selenium IDE, Montoto algorithm and ROBULA. The outputs of the algorithms considered in this experimental study are usually different, as shown in Figure 6 where each algorithm is applied to the example presented in Section 2. A description of the considered tools/research algorithms is as follows:

FirePath Absolute: FirePath[†††] is a browser-integrated add-on 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, positioning values are used to select the correct node among the set of siblings. An example of this kind of XPath locator is /html/body/table/tr[3]/td[2], see Figure 6.

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 of unique id values, the XPath locator selects the one closest to the target; it then navigates the remaining path to the target as done for absolute XPaths. An example of this kind of XPath locator is //*[@id="userInfo"]/tr[3]/td[2], see Figure 6.

Selenium IDE[‡‡‡] 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 a locator generation algorithm that generates locators using different strategies (implemented by the so-called locator builders) and that ranks them depending on an internal robustness heuristic[§§§]. In particular, as first step, Selenium IDE tries to locate the target web element by means of its id, linkText or name values (if any). For instance, the locator builders can generate the following locators id=XY, link=XY and name=XY that are equivalent to the following XPath expressions //*[@id='XY'], //a[text()='XY'] and //*[@name='XY']. Then, Selenium IDE tries to build a CSS locator using the corresponding locator builder. For

[†††] https://addons.mozilla.org/firefox/addon/firepath/ [‡‡‡] http://seleniumhq.org/projects/ide/
[§§§] https://code.google.com/p/selenium/source/browse/ide/main/src/content/locatorBuilders.js

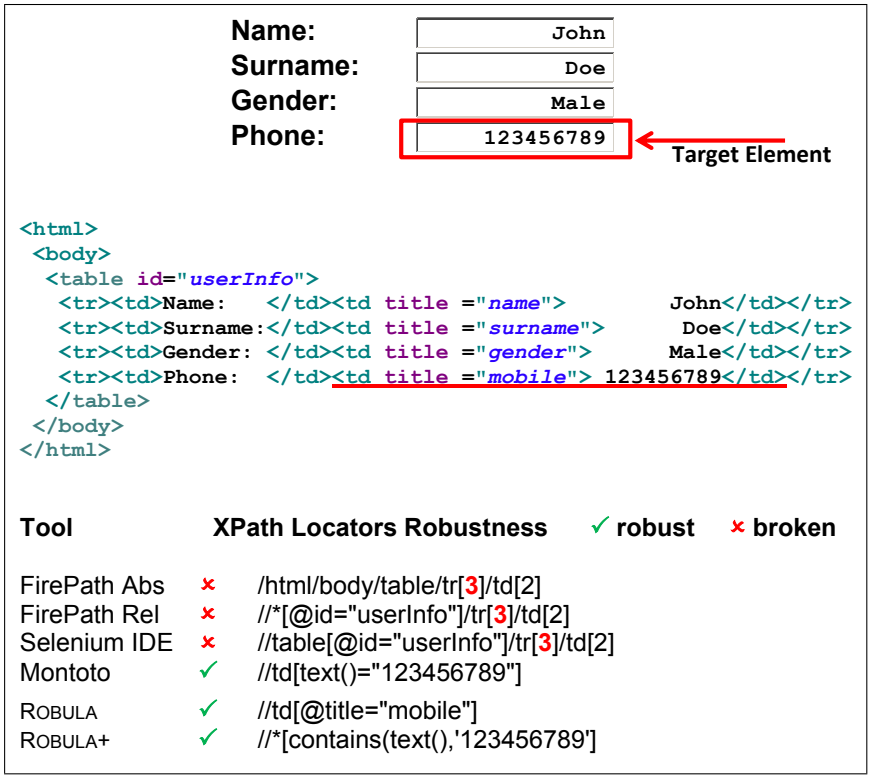


Figure 6. showInfo.php – Ver. 2 – Page, Source, Locators

instance, the expression form[name="searchCourse"] > small > a that is equivalent to the XPath //form[@name='searchCourse']/small/a. Then, if no locator is found, it tries to locate the target web element with the XPath language by applying the following locator builders in this order: (1) if the target element is a hyperlink (i.e., tag <a>), a builder that creates a locator by using the text of the link (if any), (2) if the target element is an image (i.e., tag <img>), a builder that uses the value of the attributes alt, title, and src, when any of them is available, (3) a builder that uses the values of a predefined set of attributes contained in the target node, when any of them is indeed available, in this order id, name, value, type, action, onclick; (4) a builder that creates an id-based XPath locator similar to FirePath, if an id-value is available in the ancestors of the target node; (5) a builder that creates a locator by using the value of the href attribute (if any); and, finally, (6) a builder that creates a locator composed of tag names and position values. Each locator builder is itself a locator generation algorithm, but it might be unable to create a locator — with the exception of the last locator builder (i.e., (6), based on the usage of tag names and position values), which is always able to generate an XPath locator for any target element. The Selenium IDE locator generation approach is very different from ROBULA+. In fact, Selenium IDE tries subsequently different strategies (i.e., the locator builders), and in case a strategy fails, it tries with the next one. On the contrary, ROBULA+ applies at each level of the XPath what it considered to be the best strategy. Moreover, it is also able to combine different strategies at different levels. An example of Selenium IDE locator is //table[@id="userInfo"]/tr[3]/td[2], see Figure 6. It should be noticed that, differently from ROBULA+, Selenium does not generate XPath locators using the wildcard character "*".

Montoto et al. [10] proposed an algorithm for identifying the target elements during the navigation of AJAX websites. The algorithm starts from a simple XPath expression that is progressively augmented with textual and attribute information. More precisely, the algorithm first tries to identify the target element according to its associated text (if the element is a leaf node) and then it conjuncts, one after the other, the predicates based on the attribute values (without prescribing any specific order of insertion). If this is not sufficient for generating a unique locator for the target element, each ancestor (and the value of their attributes) is subjected to the same procedure, until the root of the DOM is reached. Even if both Montoto and ROBULA+ adopt a top-down approach in the construction of the XPaths, remarkable differences between them exist: the XPath expressions generated by ROBULA+ are usually very different and shorter than the ones generated by Montoto. Indeed, when Montoto is not able to create an XPath locator using only attributes and text contained in the target element: (1) it leaves in the current level of the XPath expression all the generated predicates based on the attribute values and text; and, then, (2) it considers the attributes of the ancestors. On the contrary, the strategy implemented in ROBULA+ discards the redundant information contained in the lower levels of the XPath. As an example, to localise the target div element in the web page used as an example in the Montoto et al. [10] paper, the Montoto algorithm generates //td/a[@href="#"]/div[@class="c1" and text()="More Info"] while ROBULA+ generates the following simpler XPath expression //td/*/div. Another example of a Montoto's XPath locator is //td[text()="123456789"], see Figure 6.

ROBULA The previous version [9] of our ROBUst Locator Algorithm. An example of this kind of XPath locator is //td[@title="mobile"], see Figure 6.

ROBULA+ As described in Section 3, this algorithm is an evolution of ROBULA [9], which enhances the previous algorithm with several improvements: *(i)* the adoption of a prioritisation strategy, to rank candidate XPath expressions by heuristically estimated attribute robustness; *(ii)* blacklisting, to exclude attributes that are intrinsically fragile; *(iii)* inclusion of textual information in the predicates composing the XPath locators; *(iv)* conjunction of multiple attributes for the same node in the XPath; *(v)* creation of tag independent locators, thanks to the wildcard "*", which leaves the tag name unspecified. An example of this kind of XPath locator is //*[contains(text(),'123456789'], see Figure 6.

4.4. Research Question and Metrics

Our study aims at answering the following research questions:

RQ1: *Does* ROBULA+ *reduce the number of broken XPath locators w.r.t. the state of the art/practice XPath generation algorithms/tools?*

The goal of the first research question is to compare the robustness of the XPath locators generated by ROBULA+ with the robustness of XPath locators generated by: (1) FirePath (release 0.9.7), a state of the practice XPath generator tool; (2) Selenium IDE (release 2.8.0), a state of the practice functional web testing tool implementing a quite advanced mechanism for creating locators; (3) Montoto [10], a state of the art algorithm for XPath locator generation (see Section 4.3). The metrics used to answer RQ1 is the number of broken XPath locators in the next software release.

RQ2: *Does* ROBULA+ *generate XPath locators more robust than* ROBULA*?*

The second research question aims to compare the robustness of the locators generated by ROBULA+ with its predecessor ROBULA. In this way, we can estimate the effect of the various enhancements introduced in ROBULA+ (see Section 3) on the robustness of the generated XPath locators. In order to answer RQ2 we used the same metrics of RQ1 (i.e., number of broken locators).

RQ3: *Does the complexity of the XPath locators structure influence their robustness?*

The third research question aims to analyse the features/elements composing the XPath locators generated by the considered algorithms, in order to understand how they influence the locators robustness.

RQ4: *Is the execution time of* ROBULA+ *acceptable for a web tester?*

The fourth research question is about the time required for the execution of ROBULA+, as experienced in practical cases. Given that the theoretical computational complexity of ROBULA+ is exponential, we assess the effectiveness of the heuristics introduced in the algorithm for limiting such exponential blow-up. We also compare the time required by ROBULA+ with that required by the other algorithms (ROBULA and Montoto). The metrics used to answer RQ4 is the execution time in seconds.

4.5. Procedure

To answer our RQs we proceeded as follows:

1. We selected eight open-source web applications from *SourceForge.net* as explained in Section 4.2.
2. For each application and for each web page related to core functionalities of the application (e.g., we did not consider the configuration and installation pages), 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); and, (3) 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 detail, we excluded: (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 main page of the web application that can be found in every page and that has in every page exactly the same locator); and, (2) similar web elements from common groups (e.g., for a calendar with a checkBox, for each day we selected only one of the checkBoxes).
3. For the first release of each web application and for each selected web element, we generated: (1) the absolute XPath locator and (2) the id-based relative XPath locator both using FirePath; (3) the Selenium IDE locator[¶¶¶] (when the locator is not an XPath expression, we manually translated it in an equivalent XPath locator as shown in the examples in Section 4.3); (4) the XPath locator generated by the Montoto algorithm; and, finally, (5) the ROBULA and (6) ROBULA+ XPath locators. The result of this activity is, for each web element of the first release of each web application, six XPath locators. While the first three kinds of locator are generated manually using the FirePath and Selenium IDE tools, the last three (i.e., Montoto,

[¶¶¶] Selenium IDE usually proposes a set of locators that can be employed for selecting the target web element (e.g., when more than one locator builder is able to generate a locator, see Section 4.3); the first locator is the predefined choice that is automatically inserted into the test case, but the web tester is free to select another locator. In our empirical evaluation we always chose the first proposed locator, i.e., the one that Selenium IDE would use in the test case.

ROBULA, and ROBULA+) are generated automatically by our Java implementations of the respective algorithms. The absolute XPath locators are used as baseline in terms of robustness.

4. 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 oracles to verify the robustness of the generated XPath locators for the elements of the first release of the applications. It should be noticed that *manual* definition of the oracle abs→abs' is unavoidable, since any hypothetical technique capable of producing such mapping automatically would be also capable of solving the robust locator problem without errors.

To answer RQ1 and RQ2, for each web element we evaluate the robustness of the locators generated by the six considered algorithms on the next release of each web application by automatically verifying whether they are still able to locate the web element of interest, i.e., for each algorithm we automatically verify if the web elements selected by the corresponding XPath locator and by the absolute locator abs' are the same.
To answer RQ3, for each of the six locators created for each web element, we count: (1) the number of levels, (2) the number of position values, (3) the number of attributes, (4) the number of texts that compose the locator itself. We analyse the correlation between the number of broken locators and the number of XPath language constructs employed by the various algorithms. We used the *Pearson's r* coefficient, a measure of the correlation between two variables X and Y.
To answer RQ4, we measure the execution time (in seconds) required by ROBULA+ to generate the XPath locators. We also compared this value with the time required by the other algorithms we implemented in Java (i.e., ROBULA and Montoto). Each algorithm has been executed five times to average over any random fluctuation of the execution time. We executed the algorithms on a machine hosting an Intel Core i5 dual-core processor (2.5 GHz) and 8 GB RAM, with no other computation or communication load, in order to avoid CPU and memory saturation.

4.6. Results

This section reports the quantitative results of the empirical study, while the interpretations and implications of the results are further analysed in Section 4.7.

Table II reports the data used to answer RQ1 and RQ2. For each application and for each kind of locator (i.e., Absolute, id-based relative, Selenium IDE, Montoto, ROBULA 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 eight web applications.

As expected, the performance of absolute XPath locators are not good. In four cases (i.e., Collabtive, MRBS, PPMA, and OrangeHRM) out of eight, the totality of the 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 eight applications, 871 over 1110 absolute locators result broken (i.e., 78%). These results reveal the high fragility of the absolute XPath locators generated by state of the practice tools, as already reported in other works [17, 20] (see Section 2.4).

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 eight applications, 557 out of 1110 id-based relative locators are broken (i.e., 50%). These results are consistent with those reported in our previous work [8], where we found that 58% of the 791 XPath locators were broken from a release to the next one (see Section 2.2). 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.

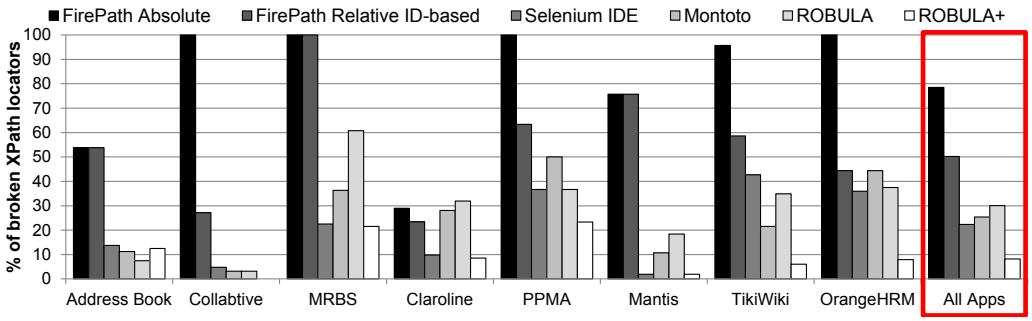
In the following, in order to answer our research questions, we analyse the results of the XPath locators generated by ROBULA+.
RQ1: the XPath locators generated by ROBULA+ are by far more robust than the absolute and id-based relative locators in all the cases (see Table II and Figure 7). Indeed, by adopting the locators generated by ROBULA+, we obtain only 91 broken locators out of 1110 while, as seen before, FirePath produces respectively 871 and 557 broken locators. Thus, by adopting the ROBULA+ locators we have on average 90% ((871–91)/871) fragility reduction with respect to the absolute

Table II. Robustness of the various kinds of XPath locators

	Address		Collabtive		MRBS		Claroline		PPMA		Mantis		TikiWiki		OrangeHRM		All Apps	
N of Target Web Elements	**80**		**125**		**102**		**235**		**30**		**103**		**232**		**203**		**1110**	
Locators	**Broken**	**%**	**Broken**	**%**	**Broken**	**%**	**Broken**	**%**	**Broken**	**%**	**Broken**	**%**	**Broken**	**%**	**Broken**	**%**	**Broken**	**%**
FirePath Absolute	43	54	125	100	102	100	68	29	30	100	78	76	222	96	203	100	**871**	78
FirePath Relative ID-based	43	54	34	27	102	100	55	23	19	63	78	76	136	59	90	44	**557**	50
Selenium IDE	11	14	6	5	23	23	23	10	11	37	2	2	99	43	73	36	**248**	22
Montoto	9	11	4	3	37	36	66	28	15	50	11	11	50	22	90	44	**282**	25
ROBULA	6	8	4	3	62	61	75	32	11	37	19	18	81	35	76	37	**334**	30
ROBULA+	10	13	0	0	22	22	20	9	7	23	2	2	14	6	16	8	**91**	8

% percentage of broken locators over the total number of locators of this kind

Figure 7. Robustness of the various kinds of XPath locators



XPath locators (see Table III) and 84% reduction w.r.t. id-based relative locators. The extreme case is Collabtive, where ROBULA+ is the sole algorithm able to produce robust locators for all the web elements (i.e., 0 broken out of 125), thus the improvement in terms of fragility reduction compared with the other algorithms is always 100% in this case. On the other hand, still in the case of Collabtive all the FirePath absolute locators result broken (i.e., 125 broken out of 125, see Table II). ROBULA+ is able to reduce always the fragility w.r.t. these two kinds of locators (by at least 63%, see Table III). On four applications the reduction exceeds 82% (Collabtive, Mantis, TikiWiki, OrangeHRM).

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 //*[@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 are the resulting locators after such 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].

The XPath locators generated by ROBULA+ are, in general, also more robust than the Selenium IDE and Montoto locators (91 broken vs. 248 and 282 respectively considering all the web applications). Indeed, the adoption of the ROBULA+ locators allows to reduce fragility by 63% and 68% w.r.t. the Selenium IDE and Montoto locators (see Table III). In detail, ROBULA+ always outperforms Selenium IDE and Montoto, with only two exceptions, the case of AddressBook, where Montoto shows a slightly better performance (10 broken for ROBULA+ and 9 for Montoto, see Table II) and the case of Mantis where Selenium IDE and ROBULA+ have the same performance. Note that, AddressBook is the only application where a locator generator algorithm (i.e., Montoto) is able to perform better than ROBULA+ (i.e., only -1 broken locator due to a web element where the title-based

Table III. Fragility Reduction (-) and Increase (+) when adopting ROBULA+ Locators w.r.t the other kinds of locators

		Address Book	Collabtive	MRBS	Claroline	PPMA	Mantis	TikiWiki	Orange HRM	All Apps
ROBULA+ vs.	FirePath Absolute	✓ -77%	✓ -100%	✓ -78%	✓ -71%	✓ -77%	✓ -97%	✓ -94%	✓ -92%	✓ **-90%**
	FirePath Relative ID-based	✓ -77%	✓ -100%	✓ -78%	✓ -64%	✓ -63%	✓ -97%	✓ -90%	✓ -82%	✓ **-84%**
	Selenium IDE	✓ -9%	✓ -100%	✓ -4%	✓ -13%	✓ -36%	0%	✓ -86%	✓ -78%	✓ **-63%**
	Montoto	✗ 11%	✓ -100%	✓ -41%	✓ -70%	✓ -53%	✓ -82%	✓ -72%	✓ -82%	✓ **-68%**
	ROBULA	✗ 67%	✓ -100%	✓ -65%	✓ -73%	✓ -36%	✓ -89%	✓ -83%	✓ -79%	✓ **-73%**

locator generated by ROBULA+ is broken while the src-based locator generated by Montoto is not, a the rare case where the attributes prioritization is not effective). We analyse the reasons for these results in Section 4.7.

To summarize, for what concerns research question RQ1, we can say that, for all the considered applications, the adoption of the XPath locators generated by ROBULA+ results in a significant reduction of the number of broken locators (in the range from -63% to -90%), which is expected to be associated with a corresponding reduction of the maintenance effort required to repair the test cases using such broken locators.

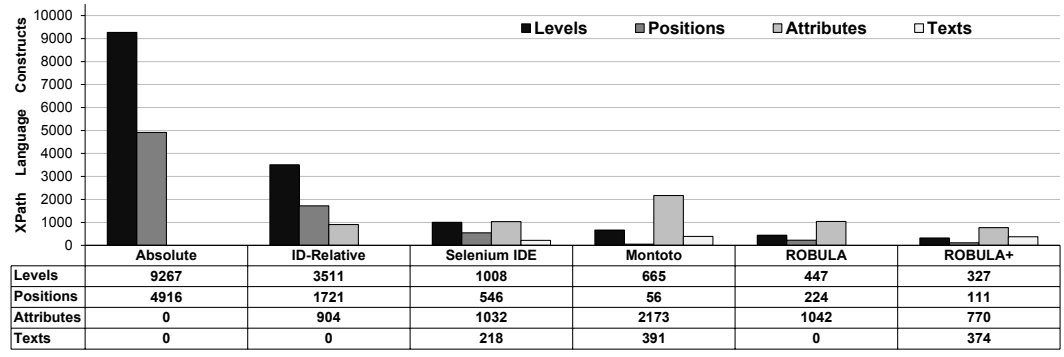
In our experience, repair of broken locators is the major cost factor during web testware evolution [11]. Hence, ROBULA+ can give a substantial contribution to the reduction of such cost.

RQ2: The improvements implemented in ROBULA+ strongly affect the robustness of the generated locators and allow to obtain far better results than ROBULA. Indeed, considering all the eight applications, ROBULA+ scores 91 broken locators against the 334 broken locators of ROBULA, corresponding to 73% reduction of the number of broken locators. In detail, in four cases out of eight (Collabtive, Mantis, TikiWiki, and OrangeHRM) ROBULA+ outperforms ROBULA obtaining drastic fragility reductions (i.e., greater than 79%, see Table III). In three cases (MRBS, Claroline, and PPMA), the fragility reductions are lower but still relevant (i.e., ranging from 73% to 36%). Only in the case of AddressBook ROBULA performs better than ROBULA+ with 6 and 10 broken locators respectively. This case will be analysed in more detail in Section 4.7.

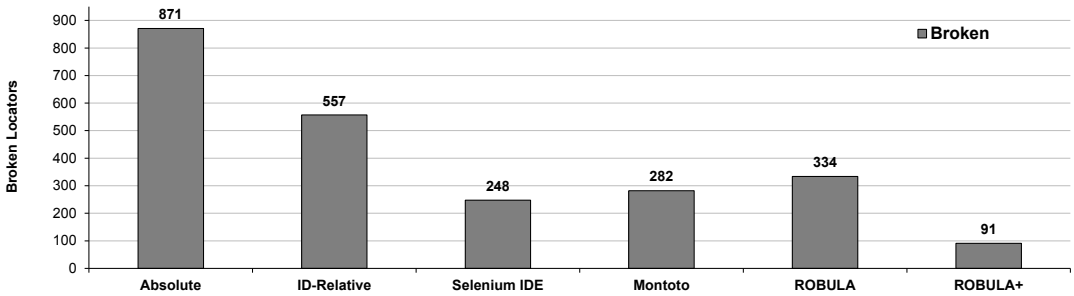
To summarise, with respect to the research question RQ2, we can say that the XPath locators generated by ROBULA+ are by far more robust than the locators generated by ROBULA (-73% of broken XPath locators).

RQ3: Figure 8 contrasts the structure of the various kinds of XPath locators (top) and the corresponding robustness (bottom). In particular, the histogram above reports for each kind of XPath locator the total number of XPath language constructs used to build the locators considered in our study, while the histogram below reports the total number of broken locators for each kind of XPath. For instance, the ROBULA+ locators contain overall 327 additional levels (the initial level is skipped, since each expression must be at least one level long), 111 positions, 770 attributes and 374 texts. As an example, the XPath //tr[2]/td[@name="feature" and @value="one"] has 1 additional level, 1 position and 2 attributes. From Figure 8, it is clear that Absolute and id-based relative XPath locators heavily resort to the usage of additional levels and position constructs. Concerning the id-based locators, in 904 case out of 1110 (the total number of locators analysed in our study), an id value, allowing to create a locator, is found in the target web element or in its ancestors. This helps to reduce the number of additional levels in the XPath locator as well as the number of positions used. The other algorithms rely by far less on the structural constructs of the XPath language (i.e., levels and position values) and employ as much as possible predicates based on the values of various attributes and text. *Pearson's r* correlation between the number of broken locators and the number of additional levels or positions is respectively 0.948 and 0.942. Thus, there is a strong correlation between number of additional levels and positions and fragility of the locators, as can be also noticed

Figure 8. Analysis of the XPath Language Constructs used by the various kinds of XPath Locators



	Absolute	ID-Relative	Selenium IDE	Montoto	ROBULA	ROBULA+
Levels	9267	3511	1008	665	447	327
Positions	4916	1721	546	56	224	111
Attributes	0	904	1032	2173	1042	770
Texts	0	0	218	391	0	374



graphically by looking at Figure 8. On the other hand, the *Pearson's r* coefficient between the number of broken locators and the number of attributes or the use of text is respectively -0.580 and -0.733. Thus, creating XPath locators that rely more on attributes and text helps in obtaining robust locators. Overall, analysing the total number of constructs used in the various XPath locators (i.e., levels, positions, attributes and texts) and their corresponding robustness, for all the six algorithms, we have a value for the *Pearson's r* of 0.953, that reveals a strong correlation between the number of constructs used by the locators and their fragility. This explains the higher robustness achieved by ROBULA+. In fact, this algorithm has been designed so as to keep the XPath locators as short and simple as possible.

To summarise, for what concerns research question RQ3 we can say that reducing as much as possible the number of additional levels and position values used by the XPath locators helps to reduce their fragility; resorting mainly to attribute-value pairs and texts helps also to reduce fragility. In general, maintaining the XPath locators as short and simple as possible helps in achieving lower fragility.

RQ4: We measured the execution time of ROBULA+, ROBULA and Montoto, since we implemented these algorithms in Java. We did not measure the time required by FirePath and Selenium IDE because these tools generate locators interactively. In our experience, they are able to generate the locators almost instantaneously. For generating the 1110 XPath locators, ROBULA+ has required on average 183.40 seconds. This means that, on average, the generation of a ROBULA+ locator took only 0.17 seconds, i.e., a very short time that can be considered instantaneous from the web tester perspective. On the other hand, ROBULA required 84.71 seconds, that corresponds to 0.08 seconds per locator, and Montoto 7.94 seconds, that corresponds to 0.007 seconds per locator. In conclusion, due to its elaborate design, ROBULA+ requires more execution time than its competitors. However, this time is undoubtedly acceptable and negligible for the tester (only 0.17 seconds per locator on average). Moreover, the robustness results highly encourage its adoption in real testing scenarios.

To summarise, with respect to the research question RQ4 we can say that the time required by ROBULA+ for generating the XPath locators is undoubtedly acceptable for a human tester (only 0.17 seconds per locator on average).

4.7. Discussion

Absolute XPath locators exhibited a very high level of fragility, in seven out of eight cases. In the case of Collabtive, MRBS, PPMA, and OrangeHRM, web pages have been modified during the evolution of the web applications at the higher levels of the DOM (i.e., the nodes close to body) and as a consequence all absolute locators are broken. In other cases, the robustness of the absolute locators was negatively affected by the modification of the list of the siblings preceding the target elements. For instance, adding a field in a form breaks all the absolute locators for the subsequent fields, if such locators use positional values, as in the case of FirePath. Only in the case of Claroline, a good amount of absolute locators survive the software evolution (71%), because the majority of the web pages did not significantly change their structure in the subsequent release.

id-based relative XPath locators showed a lower level of fragility (in five out of eight cases) if compared to absolute locators. Indeed, when one of the ancestors of the target element, or the target element itself (e.g., td), contains an id-value (e.g., id="xy"), this can be used for generating an id-based relative locator (e.g., //div[@id="xy"]/.../td) that is not affected by the changes in the higher levels of the DOM (as an absolute locator is). Thus, the availability of (meaningful) ids affects positively the robustness of this kind of locators. In our case study, two applications (i.e., MRBS and Mantis) do not use any id attribute in their first releases. For this reason, the id-based relative locators exhibit exactly the same fragility as the absolute ones. On the other hand, in the case of Claroline and Collabtive, respectively 1/5 and 1/7 of the nodes have an id attribute. Thus, FirePath is often able to generate short locators (one or two levels long), which turn out to be robust XPath locators – for Claroline even better than ROBULA locators, which often use the "unreliable" href attribute, but not better than ROBULA+, which does not use it, thanks to the black list. In the other applications, we have cases where ids are not so used, thus the obtained id-based relative locators may be as long as 5, 6 or 7 levels (hence they tend to be quite fragile) or ids are not meaningful (e.g., auto-generated ids) and as a consequence id-based relative locators are not so robust on the next release.

ROBULA and ROBULA+ XPath locators showed a lower fragility than the one of id-based relative locators (in seven out of eight cases, see Figure 7). The reason is that ROBULA and ROBULA+ locators are less dependent on the web page structure than id-based relative locators. In fact, the former are by far shorter than the latter, with 447/327 vs. 3511 additional levels (see Figure 8). In the case of Claroline (the only application in which ROBULA performs worse than FirePath), several locators use href attributes that proved to be not very robust. Indeed, the href values used in such locators often contain: (1) long filesystem paths changing during the evolution of the application, or (2) parameters passed to the target page that are changed from a release to the next one (e.g., from //a[@href="login.php?s=MTUv"] to a[@href="login.php?s=MDcv"]). The same problem applies to locators using the src attribute. Strangely enough, in the case of Claroline, ROBULA locators are also more fragile than absolute locators, because href and src values have been changed more frequently, among the two releases, than the full DOM paths. On the other hand, **ROBULA+**, thanks to prioritisation and black listing, remains the best option also in the case of Claroline. In the case of Mantis, ROBULA+ is significantly better than ROBULA (respectively 2 vs. 19 broken locators). This is due to the fragile attributes (e.g., tabindex and href) used by ROBULA which are black-listed in ROBULA+. In detail, ROBULA generated 12 locators using the tabindex attribute and 8 of them resulted broken in the new release of Mantis, while in almost all these cases ROBULA+ generated locators using the name attribute, resulting in only one broken locator. In the case of AddressBook ROBULA is slightly better than ROBULA+ (6 vs. 10 broken locators respectively). This is due to the fact that the texts used by ROBULA+ for locating some links changed across the considered releases of the application, while the attribute values used by ROBULA remained more stable.

Comparing the **ROBULA, Selenium IDE,** and **Montoto XPath locators** we can notice that overall they achieve a similar level of robustness (see Figure 7). Some relevant differences can be observed in the case of: (1) Claroline, where as seen before ROBULA produces several fragile XPath locators

relying on the fragile href and src attributes values; for the same reason also Montoto does not perform well. On the contrary, the performances of Selenium IDE are close to the ROBULA+ ones, since it is able to avoid the generation of locators containing the fragile href and src attributes, thanks to the prioritization among the different strategies for generating locators, depending on the internal robustness heuristic strategy that such tool implements (see Section 4.3); (2) Mantis, where, Selenium IDE and Montoto perform better than ROBULA since they avoid the usage of the href attribute and (only in the case of Selenium IDE) of the tabindex attribute. Indeed, higher priority is given to text-based locators and (only in the case of Selenium IDE) to the name attribute; (3) TikiWiki, where Montoto creates more robust locators, thanks to the usage of the text contained in every kind of target element (i.e., not only for links), and, in some cases, thanks to the usage of multiple attributes for the same XPath element, which produces shorter and more robust XPath locators; (4) MRBS, where ROBULA creates fragile XPaths based on the href attribute or on position values, while Montoto, Selenium IDE (and ROBULA+), rely on the more robust text contained in the target elements.

ROBULA+ achieves by far the best performance with respect to all the other considered algorithms/tools. In fact, it incorporates and combines various strengths of the existing solutions in a single algorithm. ROBULA+ makes use of a prioritization strategy like Selenium IDE, even if it is conceived and implemented in a different way. The differences between the two strategies are remarkable. As described in Section 4.3, Selenium IDE tries to create the target locator executing, one after the other (similarly to a pipeline), a set of different locator builders (i.e., XPath generation algorithms). If a locator builder is not able to return a locator, Selenium IDE tries with the next locator builder (locator builders are ordered according to an internal robustness heuristic). On the other hand, ROBULA+ tries all the different strategies for each specialization step (e.g., use id value, text, attribute-value pairs) and it orders them according to a robustness heuristic. For instance, ROBULA+ is able to create an XPath locator that is text-based at level 1 and attribute-based at level 2, while this is not permitted by Selenium IDE, where only one strategy (builder) is selected to produce a locator. This explains why in the case of Claroline ROBULA+ performs even slightly better than Selenium IDE (i.e., avoiding the usage of href and src attributes) and by far better than ROBULA and Montoto. ROBULA+, similarly to Montoto, is able to build XPath locators using the text contained in every kind of DOM element, i.e., in the target and in its ancestors. This has allowed ROBULA+ to produce more robust XPath locators, for instance in the case of TikiWiki. Moreover, ROBULA+ is able to insert predicates based on multiple attribute-value pairs. But differently from Montoto, ROBULA+ uses the smallest set of attribute-value pairs and discards each additional construct not strictly necessary for creating a locator. This is clearly visible by looking at Figure 8, where the locators generated by Montoto for our set of Web applications contains by far more attribute-value pairs than ROBULA+ (respectively 2173 vs. 770).

We also **analysed the effect of the transfAddAttributeSet** in ROBULA+. We created a version of ROBULA+ where such transformation is disabled. On the considered 1110 target elements we discovered that: (1) the fragility increases by only one broken locator (i.e., from 91 to 92 broken locators overall); and, (2) the total time required for generating the 1110 XPath locator decreases from 183.40 to 131.21 seconds. Even if in our case study the robustness level does not change significantly, enabling transfAddAttributeSet in ROBULA+ has allowed to generate XPath locators less coupled to the web page structure. For instance, in the case of TikiWiki, 52 locators contains both attribute-value pairs and positions when transfAddAttributeSet is disabled (e.g., //*[2]/*[@value="comments"]), while they contain only predicates on attribute-value pairs when such transformation is enabled (e.g., //*[@name="permRegistered" and @value="comments"]). In Section 4.6, we have seen that position values are correlated with a higher fragility and for this reason we think that it is always better to include the transfAddAttributeSet transformation, except for the cases in which strong time constraints hold.

Then, we also **analysed the effect of Attribute Prioritization**, **Attribute Black List** (see Section 3.4) and of the **usage of textual values** on the robustness of the locators generated by ROBULA+. We created several versions of our algorithm where such features have been selectively disabled. The experimental results show that by disabling selectively attribute prioritization, attribute black list and usage of textual values, the number of broken locators increases from 91 to respectively 118, 123

and 246. Thus we can say that the usage of textual values has a strong effect on the robustness of ROBULA+ locators, even though attribute prioritization and black list have also a relevant effect.

Then, we also **analysed the effect of specializing the wildcard "*"** in ROBULA+, i.e., the ability of creating locators independent from the DOM element type. By removing this feature from ROBULA+ we obtained a relevant increment, from 91 to 138, of the number of broken locators. Among the considered releases of the eight web applications, some target web elements changed their tag types. For instance, in OrangeHRM, there are some cases of link texts shown in boldface that are implemented in different ways between the two releases. In the first release, the tag b is used, while in the second release the developers opted for a better solution: the tag span and a CSS rule. In such cases, ROBULA+ has generated a locator using "*" (e.g., similar to //*[contains(text(),'Admin')]), while by disabling the star specialization the algorithm has generated a locator using the tag b (e.g., similar to //b[contains(text(),'Admin')]). The first type of locators has survived the application evolution; the second clearly not. The only advantage of disabling the star specialization is in the execution time required for generating the 1110 XPath locator that decreases from 183.40 to 88.34 seconds. The reason is that the search space is at least doubled when adopting the star specialization. In fact, each candidate locator with a tag name is paired with another candidate locator with a star ("*"); the two are combined in all possible hybrid forms of candidate locators, by specializing stars or tags at different levels. One objection to the use of the star in locators is that in some cases, when complex XPath locators are generated, having the tag name instead of "*" might help in locator understanding (e.g., during maintenance of broken locators). Since usually the locators generated by ROBULA+ are very short (typically 1 level long), this is not a relevant problem. In fact, ROBULA+ has produced, on our set of Web applications, only 327 additional levels over 1110 XPath locator (see Figure 8), thus on average a ROBULA+ locator is only 1 or 2 levels long (average length is 1.29).

Finally, we **compared** the robustness of the locators generated by **ROBULA+** and **Selenium IDE** focusing on the subset of web elements that Selenium IDE natively locates using XPath locators. Over our eight subjects web applications, Selenium IDE makes use of an XPath locator for 277 over 1110 web elements (i.e., in 25% of the cases). Such XPath locators are usually very fragile: 146 out of 277 (53%) were broken when evaluated on the second release of the web applications. On the other hand, the locators generated by ROBULA+ are by far more robust, scoring only 35 broken locators out of 277 (i.e., 13% of the cases). By analysing the XPath locators generated by Selenium IDE we discovered that in these cases, very often, the tool relies on id-based relative XPath locators and that their fragility level is consistent with the one we observed for the FirePath Relative id-based XPath locators on the entire set of web elements considered in this study (i.e., respectively 53%, as reported above, and 50%, as reported in Table II, of broken locators). ROBULA+ promotes the generation of shorter XPath locators, less coupled with the page structure w.r.t. id-based relative XPath locators, which were by far more resilient to the web application evolution. Finally, it is interesting to notice that ROBULA+ overcomes Selenium IDE also in the cases in which the latter uses non XPath locators. Indeed, on 833 cases (i.e., 1110-277), ROBULA+ scores only 56 broken locators against 102 of Selenium IDE.

4.8. Threats to Validity of the Study

In this section we describe the main threats to validity that could affect our empirical study [19].

Concerning the generalisation of results, we selected eight real open source web applications belonging to different domains and overall 1110 target web elements subdivided among the chosen applications, which makes the context realistic, even though further studies with other applications and web elements are necessary to confirm the obtained results. We highlight that the specific technologies and languages used to implement the server-side portion of the selected web applications (in our case PHP language and PHP frameworks, see Section 4.2) are not a threat to the validity of this study since we compared XPath locators, which are not dependent on the underlying server-side technology. On the other hand, the adoption of rich internet applications (e.g., based on AngularJS) are expected to affect the results of this study, although we think that the effect should be the same for all kinds of XPath locators. ROBULA+ could, in certain cases, rely on attributes that are browser

specific. In such cases, locators result broken when evaluated in another browser. However this could happen also when adopting the other locators generators.

One threat could be associated with the procedure used to select the target elements. To reduce as much as possible this threat, we adopted the systematic procedure described in Section 4.5, so as to focus on all the web page elements that could be used hypothetically in a functional web test case. We believe that this strategy is better than selecting only the web elements exercised by an existing test suite. First, because it is more general and second because it is more objective (i.e., the selection of the target elements does not depend on the specific choices of the web tester and on the adequacy level aimed for).

The choice of the releases considered in the study (see Section 4.2) may have affected the results. In our study we considered only major releases, because with small differences between releases the majority of the locators and, thus, of the corresponding test cases are expected to work without problems. However, we have no reason to believe that the direction of the results would vary when considering different releases, although the magnitude of our findings might change.

Another threat, concerning particularly RQ1 and RQ2, is the set of locators generator tools/algorithms used in the experiment. We compared the robustness of ROBULA+ against four different tools/algorithms (i.e., FirePath Absolute and id-based relative, Selenium IDE and Montoto) and the initial version of our algorithm (ROBULA) but we have not considered the entire plethora of existing solutions. In principle, different results could be obtained using other state of the practice/art XPath locator generators. However, we want to highlight that:

- The absolute and relative XPath locators generated by the most popular state of the practice XPath locators generation tools (e.g., XPath Helper available from the Chrome Web Store of Google) seem to be, typically, very similar to the ones generated by FirePath. Its generation strategies are also very similar to FirePath's, so the results of our study are not expected to change significantly if other tools, e.g. XPath Helper, are considered;
- To the best of our knowledge, the two state of the art locator generators are Selenium IDE and Montoto. Both are considered in this study;
- Selenium IDE can be considered one of the flagship test automation tools, thus comparing its effectiveness with ROBULA+ can be very interesting both for practitioners and academics;
- Montoto is a quite recent research algorithm that, to the best of our knowledge, is still not commonly used as locator generator. However, it implements an approach that is quite different from both Selenium IDE and ROBULA+, thus its inclusion in the comparison is interesting.

Another threat could be associated to the manual translation of Selenium IDE locators in equivalent XPaths (performed only when Selenium IDE does not produce already an XPath locator). However, this translation was very simple and double checked by the authors. For instance, the Selenium IDE locators id=XY, link=XY and name=XY are trivially equivalent to the following XPath expressions //*[@id='XY'], //a[text()='XY'] and //*[@name='XY']. Moreover, in the empirical evaluation, we chose always the default locator proposed by Selenium IDE among the proposed ones (see details in Section 4.5); in this way we had not introduced any subjectivity and evaluated the robustness of the locators considered the best by the Selenium IDE locator generator.

Concerning RQ3, other more refined statistical strategies and analyses could be used to answer it. However, the trend shown in Figure 8 is clear and so the simple Pearson's correlation seemed sufficient to us to confirm the visual interpretation of the data statistically.

Finally, concerning RQ4, the answer to this research question might depend on the actual usage of the tool. In our experiment, we computed the execution time of ROBULA+ and of Montoto. Even if ROBULA+ requires more time than Montoto, both execution times are largely below one second, hence we consider them negligible for a human web tester perspective (everything happens within the time of a click).

5. RELATED WORK

The problem of creating and maintaining test suites for web applications has been studied under different points of view in the context of testing and test automation [12, 5, 6]. Other works deal with the problem of automatically or semi-automatically extract information from structured sources [10, 21, 22, 20, 23, 24].

5.1. Test Maintenance and Test Automation

The problem of maintaining test scripts is well-known by the practitioners and has been taken into consideration in the last years by the research community, although there is still no consolidated solution. *Choudhary et al.* [12] propose WATER, a tool that suggests changes that can be applied to repair test scripts for web applications. This technique is based on differential testing: by comparing the behaviour of a test case on two successive versions of the web application and analysing the difference between these two executions, WATER suggests repairs that can be applied to update the scripts. This work is complementary to ours since ROBULA+ aims to create robust XPath locators that are less likely to need to be repaired in the future. We will consider automatic repair techniques in our future work.

In the context of GUI testing, *Grechanik et al.* [25] describe an approach called REST for maintaining and evolving test scripts so that they can test new versions of their respective applications. The approach is based on GUI-tree comparison, in order to find altered GUI objects. The test script is then analysed using static analysis to assess the impact of the differences and to provide suggestions for changes that avoid possible failures in the GUI application. Differently from REST, ROBULA+ works in the context of web applications and interacts with the GUI elements via the DOM structure.

In the context of web testing, several papers on robust test automation and change-resilient test script creation have been proposed by *Thummalapenta and colleagues* [26, 27, 5]. The tool ATA, developed at IBM, uses a combination of natural-language processing, backtracking exploration and learning, to improve the tester's productivity in automating manual tests. ATA also produces change-resilient scripts, which automatically adapt themselves in the presence of certain common types of user-interface changes. ROBULA+ also works in a web scenario, but differently from ATA, it aims at strengthening the resilience of DOM locators, rather than considering automatic adaptation to changes. Recently, novel solutions to the test-script fragility problem have been proposed [6, 28]. Both *Yandrapally et al.*'s [6] and *Pirzadeh et al.*'s [28] methods are based on visual landmarks to build locators resilient to the evolution of the applications. These approaches are promising because they take advantage of multiple aspects of the representation of the application and eliminate almost entirely the usage of the web page internals details (i.e., the DOM). On the other hand, using the complete information contained in the DOM allows to generate very robust locators (see Section 4) that cannot otherwise be created.

While the goal of some of the approaches listed above (e.g., ATA) is similar to that to ROBULA+, the technical solution is completely different. In fact, ROBULA+ is focused exclusively on the DOM locators and their robustness, without considering any natural language processing or visual landmark identification for the automatic adaptation of test scripts to changes. While it would be interesting to compare ROBULA+ with such different approaches and to integrate ROBULA+ with them, so as to compensate for each other's weaknesses (indeed, this is part of the plan for our future work), we can anyway notice that operating on the DOM locators alone provides already quite remarkable results and improvements, as reported in Section 4. Moreover, ROBULA+, differently from ATA that requires ad-hoc tools, can be adopted by developers at virtually no additional cost, since it only requires to replace the current locators with those generated by ROBULA+, e.g., by using the Firefox add-on we developed.

5.2. Information Retrieval and Data Mining

Some works in the context of information retrieval and web data mining deal with robust data extraction.

Myllymaki and Jackson [21] describe how robust relative XPath expressions can be manually generated in terms of hops starting from an anchor (e.g., a particularly stable page content). Due to the relative proximity of "interesting" data, many individual elements can be effectively extracted using the same anchor. Unfortunately, these expressions are built manually, leaving the problem of how to automatically produce such XPath expressions open. *Anton* [24] introduces two variants (F-GCP and M-GCP) of a wrapper induction algorithm (a *wrapper* in this context corresponds to what we call a *locator* in web testing) for extracting information from semi-structured documents, such as XML. The algorithm is built on the work of Myllymaki and Jackson [21] and gives preference to the use of string and attribute search patterns over structural search patterns, trying to minimize the wrapper length. The author evaluated the robustness and expressiveness of the two variants, testing both algorithms on manually annotated example pages of two web applications.

Some empirical studies [23, 20] compare the robustness of absolute and relative XPath expressions used in wrappers with manually defined relative XPaths. *Kowalkiewicz et al.* [20] found that, on a dataset composed of different versions of a large number of web pages, absolute and relative XPaths were robust respectively in 47% and 76% of the cases.

Dalvi et al. [22] make use of temporal snapshots of web pages to develop a tree-edit model used to improve the robustness of wrapper construction. They describe an algorithm that learns a probabilistic model from training examples and returns the complete set of *minimal* wrappers, i.e., XPath expressions containing only the necessary information to be locators, among which the most robust candidate is selected by using the probabilistic aforementioned model. Both ROBULA and ROBULA+ generate XPath locators by means of iterative refinements and from this point of view they follow the same principle of the wrapper generation technique proposed by Dalvi et al. [22]. However, the key difference of ROBULA+ w.r.t. Dalvi's approach consists of its web testing-oriented heuristics: prioritization, black listing, textual information, multiple attributes, etc.; see Section 3. In fact, the adoption of the iterative refinement principle alone is not sufficient to generate robust locators, as apparent from the differences in the results of ROBULA+ vs. ROBULA, whose implementation is a direct derivation from Dalvi's refinement principle. Moreover, Dalvi's technique requires to learn a probabilistic model from a corpus. On the contrary, ROBULA+ requires no learning phase, thus it can be adopted quite easily by any web tester. For what concerns the algorithmic details, ROBULA+ implements several improvements over Dalvi's approach, such as specializing only the head of an XPath (without reducing the XPath generation power), using only information available in a node or in its ancestors (to speed up convergence) and returning the first unique locator generated by the algorithm, since it is by construction the best candidate to return.

6. CONCLUSIONS AND FUTURE WORK

We proposed and experimented ROBULA+, a novel algorithm for automatically generating robust web testing-oriented XPath locators. We have compared the robustness of the locators generated by state of the art/practice tools and algorithms (i.e., FirePath absolute and id-based relative locators, Selenium IDE, Montoto and ROBULA) with the ones generated by ROBULA+. Results indicate that the locators generated by ROBULA+ are significantly better in terms of robustness than all the other kinds of locators, with 63% to 90% fragility reduction, which is expected to be associated with a corresponding reduction of the maintenance effort required to repair the test cases. Experimental results support the intuition behind ROBULA+, i.e., that maintaining the XPath locators as short as possible together with a smart combination of predicates help to obtain a lower fragility. The time required by ROBULA+ for generating the XPath locators is acceptable for a human web tester (only 0.17 seconds per locator on average). The Java implementation of ROBULA+ can be freely downloaded from our web site, as well as a preliminary version of a Firefox add-on implementing the algorithm (we plan to make it available on short notice in the official Firefox add-on store).

In our short term future work, we plan to: (1) experiment ROBULA+ with more web applications (considering also rich internet applications) and web elements, (2) develop a transformation tool able to restructure a whole test suite, replacing fragile locators with ROBULA+'s generated locators. For this task, we will use a technique similar to the one we adopted in a previous work [29] for

migrating DOM-based test code to the Visual approach. Moreover, we plan to formulate the problem of generating robust XPath locators as a graph search problem [30], so that we may apply greedy and meta-heuristic solutions (e.g., based on a genetic algorithm). For cross-browser compatibility, we also plan to extend ROBULA+ in order to generate locators that are ensured to work on different browsers. We plan to evaluate the actual readability of ROBULA+ locators (e.g., comparing ROBULA+ with the other algorithms, but also the various variants of ROBULA+, e.g., enabling or not the wildcard "*" specialization) by means of controlled experiments with humans. Finally, after having used a preliminary version of the Robust Locator Algorithm as part of the multi-locator approach [31], we plan to investigate the problem of generating robust XPath expressions able to select at the same time multiple DOM-elements.

REFERENCES

1. Tonella P, Ricca F, Marchetto A. Recent advances in web testing. *Advances in Computers* 2014; **93**:1–51, doi:10.1016/b978-0-12-800162-2.00001-4. URL http://dx.doi.org/10.1016/b978-0-12-800162-2.00001-4.
2. Chapman P, Evans D. Automated black-box detection of side-channel vulnerabilities in web applications. *Proceedings of the 18th Conference on Computer and Communications Security*, CCS 2011, ACM: New York, NY, USA, 2011; 263–274, doi:10.1145/2046707.2046737. URL http://dx.doi.org/10.1145/2046707.2046737.
3. Bruns A, Kornstadt A, Wichmann D. Web application tests with Selenium. *IEEE Software* 2009; **26**(5):88–91, doi:10.1109/ms.2009.144. URL http://dx.doi.org/10.1109/ms.2009.144.
4. Chang TH, Yeh T, Miller RC. GUI testing using computer vision. *Proceedings of the 28th ACM Conference on Human Factors in Computing Systems*, CHI 2010, ACM, 2010; 1535–1544, doi:10.1145/1753326.1753555. URL http://dx.doi.org/10.1145/1753326.1753555.
5. Thummalapenta S, Devaki P, Sinha S, Chandra S, Gnanasundaram S, Nagaraj DD, Sathishkumar S. Efficient and change-resilient test automation: An industrial case study. *Proceedings of the 35th International Conference on Software Engineering*, ICSE 2013, IEEE Press, 2013; 1002–1011, doi:10.1109/icse.2013.6606650. URL http://dx.doi.org/10.1109/icse.2013.6606650.
6. Yandrapally R, Thummalapenta S, Sinha S, Chandra S. Robust test automation using contextual clues. *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ISSTA 2014, ACM, 2014; 304–314, doi:10.1145/2610384.2610390. URL http://dx.doi.org/10.1145/2610384.2610390.
7. Massol V, Husted T. *JUnit in Action*. Manning Publications Co.: Greenwich, CT, USA, 2003.
8. Leotta M, Clerissi D, Ricca F, Tonella P. Capture-Replay vs. Programmable Web Testing: An Empirical Assessment during Test Case Evolution. *Proceedings of 20th Working Conference on Reverse Engineering*, WCRE 2013, IEEE, 2013; 272–281, doi:10.1109/WCRE.2013.6671302. URL http://dx.doi.org/10.1109/WCRE.2013.6671302.
9. Leotta M, Stocco A, Ricca F, Tonella P. Reducing web test cases aging by means of robust XPath locators. *Proceedings of 25th International Symposium on Software Reliability Engineering Workshops*, ISSREW 2014, IEEE, 2014; 449–454, doi:10.1109/ISSREW.2014.17. URL http://dx.doi.org/10.1109/ISSREW.2014.17.
10. Montoto P, Pan A, Raposo J, Bellas F, Lopez J. Automated browsing in Ajax websites. *Data & Knowledge Engineering* 2011; **70**(3):269 – 283, doi:10.1016/j.datak.2010.12.001. URL http://dx.doi.org/10.1016/j.datak.2010.12.001.
11. Leotta M, Clerissi D, Ricca F, Tonella P. Approaches and tools for automated end-to-end web testing. *Advances in Computers* 2016; **101**:193–237, doi:10.1016/bs.adcom.2015.11.007. URL http://dx.doi.org/10.1016/bs.adcom.2015.11.007.
12. Choudhary SR, Zhao D, Versee H, Orso A. WATER: Web application TEst repair. *Proceedings of the 1st International Workshop on End-to-End Test Script Engineering*, ETSE 2011, ACM, 2011; 24–29, doi:10.1145/2002931.2002935. URL http://dx.doi.org/10.1145/2002931.2002935.
13. Mirzaaghaei M. Automatic test suite evolution. *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE 2011, ACM, 2011; 396–399, doi:10.1145/2025113.2025172. URL http://dx.doi.org/10.1145/2025113.2025172.
14. Leotta M, Clerissi D, Ricca F, Spadaro C. Improving Test Suites Maintainability with the Page Object Pattern: An Industrial Case Study. *Proceedings of 6th International Conference on Software Testing, Verification and Validation Workshops*, ICSTW 2013, IEEE, 2013; 108–113, doi:10.1109/ICSTW.2013.19. URL http://dx.doi.org/10.1109/ICSTW.2013.19.
15. Christophe L, Stevens R, De Roover C, De Meuter W. Prevalence and maintenance of automated functional tests for web applications. *Proceedings of the 30th International Conference on Software Maintenance and Evolution*, ICSME 2014, IEEE, 2014; 141–150, doi:10.1109/ICSME.2014.36. URL http://dx.doi.org/10.1109/icsme.2014.36.
16. Collins E, de Lucena V. Software test automation practices in agile development environment: An industry experience report. *Proceedings of the 7th International Workshop on Automation of Software Test*, AST 2012, IEEE, 2012; 57–63, doi:10.1109/iwast.2012.6228991. URL http://dx.doi.org/10.1109/iwast.2012.6228991.
17. Leotta M, Clerissi D, Ricca F, Spadaro C. Comparing the Maintainability of Selenium WebDriver Test Suites Employing Different Locators: A Case Study. *Proceedings of 1st International Workshop on Joining AcadeMiA and Industry Contributions to testing Automation*, JAMAICA 2013, ACM, 2013; 53–58, doi:10.1145/2489280.2489284. URL http://dx.doi.org/10.1145/2489280.2489284.

18. Rao G, Pachunoori A. Optimized identification techniques using XPath. *Technical Report MSU-CSE-00-2*, IBM Developerworks 2013.
19. Wohlin C, Runeson P, Höst M, Ohlsson MC, Regnell B, Wesslén A. *Experimentation in Software Engineering: An Introduction*. Kluwer Academic Publishers: Norwell, MA, USA, 2000.
20. Kowalkiewicz M, Orlowska ME, Kaczmarek T, Abramowicz W. Robust web content extraction. *Proceedings of the 15th International Conference on World Wide Web*, WWW 2006, ACM, 2006; 887–888, doi:10.1145/1135777.1135928. URL http://dx.doi.org/10.1145/1135777.1135928.
21. Myllymaki J, Jackson J. Robust web data extraction with XML path expressions. *IBM Research Report* 2002; .
22. Dalvi N, Bohannon P, Sha F. Robust web extraction: an approach based on a probabilistic tree-edit model. *Proceedings of the 35th ACM SIGMOD International Conference on Management of Data*, SIGMOD 2009, ACM, 2009; 335–348, doi:10.1145/1559845.1559882. URL http://dx.doi.org/10.1145/1559845.1559882.
23. Abe M, Hori M. Robust pointing by XPath language: authoring support and empirical evaluation. *Proceedings of the 3rd Symposium on Applications and the Internet*, SAINT 2003, IEEE, 2003; 156–165, doi:10.1109/saint.2003.1183044. URL http://dx.doi.org/10.1109/saint.2003.1183044.
24. Anton T. XPath-wrapper induction by generating tree traversal patterns. *Lernen, Wissensentdeckung und Adaptivität, GI Workshops*, LWA 2005, DFKI, 2005; 126–133.
25. Grechanik M, Xie Q, Fu C. Maintaining and evolving GUI-directed test scripts. *Proceedings of the 31st International Conference on Software Engineering*, ICSE 2009, IEEE, 2009; 408–418, doi:10.1109/icse.2009.5070540. URL http://dx.doi.org/10.1109/icse.2009.5070540.
26. Thummalapenta S, Singhania N, Devaki P, Sinha S, Chandra S, Das AK, Mangipudi S. Efficiently scripting change-resilient tests. *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE 2012, ACM, 2012; 41:1–41:2, doi:10.1145/2393596.2393643. URL http://dx.doi.org/10.1145/2393596.2393643.
27. Thummalapenta S, Sinha S, Singhania N, Chandra S. Automating test automation. *Proceedings of the 34th International Conference on Software Engineering*, ICSE 2012, IEEE Press, 2012; 881–891, doi:10.1109/icse.2012.6227131. URL http://dx.doi.org/10.1109/icse.2012.6227131.
28. Pirzadeh H, Shanian S. Resilient user interface level tests. *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE 2014, ACM, 2014; 683–688, doi:10.1145/2642937.2642954. URL http://dx.doi.org/10.1145/2642937.2642954.
29. Leotta M, Stocco A, Ricca F, Tonella P. Automated generation of visual web tests from DOM-based web tests. *Proceedings of 30th ACM/SIGAPP Symposium on Applied Computing*, SAC 2015, ACM, 2015; 775–782, doi:10.1145/2695664.2695847. URL http://dx.doi.org/10.1145/2695664.2695847.
30. Leotta M, Stocco A, Ricca F, Tonella P. Meta-heuristic generation of robust XPath locators for web testing. *Proceedings of 8th IEEE/ACM International Workshop on Search-Based Software Testing*, SBST 2015, IEEE, 2015; 36–39, doi:10.1109/SBST.2015.16. URL http://dx.doi.org/10.1109/SBST.2015.16.
31. Leotta M, Stocco A, Ricca F, Tonella P. Using multi-locators to increase the robustness of web test cases. *Proceedings of 8th IEEE International Conference on Software Testing, Verification and Validation*, ICST 2015, IEEE, 2015; 1–10, doi:10.1109/ICST.2015.7102611. URL http://dx.doi.org/10.1109/ICST.2015.7102611.