The Journal of Systems and Software 210 (2024) 111932

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

Investigating the robustness of locators in template-based Web application testing using a GUI change classification model[☆]

Marco De Luca, Anna Rita Fasolino, Porfirio Tramontana [*]

Department of Electrical Engineering and Information Technology (DIETI), University Federico II of Naples, Via Claudio, 21, Naples, 80125, Italy

A R T I C L E I N F O

Keywords:
Capture & Replay
GUI-based testing
Regression testing
Web application testing
E2E testing
Locators robustness
GUI change classification

A B S T R A C T

GUI-based test-cases generated by Capture and Replay tools suffer from the well-known fragility problem: they may break even if small layout changes are operated in a Web application, without modifying the app functionality. An approach based on the automatic injection of HTML tag attributes named hooks in the source code of Web templates has been recently proposed to solve this problem. Such hooks allow the unique identification of GUI items to be located during test case execution. This technique showed its effectiveness in a preliminary validation study, where it allowed to significantly reduce the number of test case locator breakages in regression testing of student-made Web applications. This paper presents a further validation study where we compared the robustness of hook-based test cases against state-of-the-art and state-of-the-practice techniques for locating GUI objects. We proposed a three dimensional model for classifying different types of layout changes and used it to define a benchmark of realistic changes. Thanks to the model, we systematically compared the robustness of test cases generated by different techniques with respect to specific types of changes and studied the relationship between fragility issues and types of changes in different test case generation techniques.

1. Introduction

According to Banerjee et al. (2013), GUI testing is system testing of a software that has a graphical-user interface (GUI) front-end. Because system testing entails that the entire software system, including the user interface, be tested as a whole, during GUI testing test cases are developed and executed on the software by exercising the GUI's widgets (e.g., text boxes and clickable buttons). Capture and Replay (C&R) techniques have been widely used in industry since thirty years to perform GUI testing without requiring advanced testing/programming skills (Hammontree et al., 1992). A tester can exploit a C&R tool to automatically generate GUI-based test scripts starting from real sequences of user interactions on the GUI of the application under test, including mouse clicks, keyboard entries, navigation commands, etc. These sequences are recorded by the tool and automatically translated into executable test scripts. C&R techniques showed their usability in exploratory testing (Kaner, 2008) and the capability to achieve good effectiveness results in mobile application testing (Di Martino et al., 2020). They are very popular especially in regression testing activities, where the captured test cases can be used to re-test a modified application. Capture and Replay (C&R) tools are also commonly used in End-to-End (E2E) testing of Web applications, where the application is tested as a whole from the perspective of the end-user (Ricca et al., 2019) for different types of testing, including robustness, responsiveness, accessibility, compatibility, etc.

However, in spite of their notable advantages, C&R techniques present some known limitations. One of them consists in the *incompleteness* of the generated test cases, because not all the application behaviors can be solicited by GUI based test cases (Rafi et al., 2012). Generated test cases may also be affected by *fragility* issues, since they may fail (and cease to be applicable) even if small changes are operated in the GUI, without modifications of the functionality of the app (Coppola et al., 2019). Such test case failures are usually called "test breakages" (Hammoudi et al., 2016) in order to distinguish them from application failures that occur when tests operate correctly and reveal faults in the system under test. For example, we may consider a layout change consisting in moving an HTML tag in another point of the HTML page. This change may break a test case that located that tag by an XPath expression referencing the previous tag position within the HTML tree. Analogously, any change of a HTML tag value attribute (e.g. a change of the *class* attribute) may break any test case based on a XPath involving that attribute value.

[☆] Editor: Prof. Raffaela Mirandola.
[*] Corresponding author.
E-mail addresses: marco.deluca2@unina.it (M. De Luca), fasolino@unina.it (A.R. Fasolino), ptramont@unina.it (P. Tramontana).

https://doi.org/10.1016/j.jss.2023.111932
Received 15 March 2023; Received in revised form 1 October 2023; Accepted 8 December 2023
Available online 18 December 2023

Test breakages are a cause of inefficiency in regression testing processes, since extra time is needed for repairing the broken tests or substituting them with up-dated ones.

The problem of Web test fragility is well-known in the literature, where several techniques have been proposed either to generate more robust test cases (Yandrapally et al., 2014; Leotta et al., 2014; Pirzadeh and Shanian, 2014; Leotta et al., 2016; Kirinuki et al., 2019) or for repairing broken tests (Choudhary et al., 2011; Hammoudi et al., 2016; Leotta et al., 2015). A third type of approaches described in the literature is intended to improve the testability of the Web applications at the origin (Bajaj et al., 2015b,a) by Web page design or refactoring techniques that enable the definition of more robust locators and test cases.

In our previous work (Fasolino and Tramontana, 2022), we proposed an approach of the third type that targets the category of template-based Web applications and exploits the concept of hook-based locators. Template-based applications use Web templates for implementing the well-known Model-View separation principle for designing GUI based applications (Parr, 2004). Each template statically includes the source code of the target Web pages, i.e. the Views that will be dynamically generated at run-time. The technique that we proposed to improve the robustness of test cases is based on "Hook-based" locators that exploit an HTML tag attribute, the so-called hook, to allow the unique identification of each distinct tag of a Web page. The point of novelty of our approach is that it artificially introduces such hook attributes in the Web page templates that are the origin of each rendered web page. In this way, when a new page is generated, its tags will already include the hook attributes. Hook attributes will allow the definition of a new type of DOM-based locators, which are likely to be more resilient to Web page changes involving layout properties.

In Fasolino and Tramontana (2022) we performed a validation experiment that preliminarily showed the ability of hook-based locators to reduce the number of test case breakages and the effort of the regression testing activities. The study involved three small Web applications which were iteratively developed by university students of an Advanced Software Engineering course. The regression test cases developed using the proposed approach turned out more robust than test cases based on the default locators generated by the state-of-the-practice Katalon Recorder tool.[1] We observed indeed that 16.6% of test cases broke by fragility issues when using Katalon Recorder locators, whereas none of them broke when using Hook-based locators.

These experimental results, though encouraging, were limited to the context of Web applications developed *ad hoc* by students, which may not be representative of the user interface complexity of real template-based Web applications. Another limitation of the study was that the set of layout changes implemented by the students were limited in number, since only a few releases (less than 10) were developed and only some of them included layout changes. Finally, the study only compared hook-based locators against the ones provided by the Katalon Recorder tool, whereas further locators among the ones described in the literature are worth considering.

In this paper we present a new study we performed to further validate our technique. The aims of this study were (1) to evaluate the effectiveness of our approach with respect to GUI layout changes which may occur in real web applications and (2) to compare it against locators already presented in the literature and used in the practice.

To carry out this study we defined a novel classification model of GUI changes that may affect the layout of a Web application. The model distinguishes changes on the basis of the type of GUI item or property involved in the change (i.e. HTML tag, tag attribute, tag attribute value, text included between tags, template tag), the type of change (i.e. GUI item insertion or removal, property modification, position change) and the relative position of the changed GUI item with respect to the one directly involved in the test case. Then, we considered two open-source applications obtained from GitHub repositories and injected a benchmark of 192 exemplar changes of 55 different categories in their code, to modify the layout of these Web applications. The benchmark of changes was defined according to the proposed GUI change model. Eventually, we systematically compared the robustness of existing types of locators with respect to the different implemented changes. This study allowed us to investigate the robustness of existing locators and showed their points of strength and weakness with respect to typical Web layout changes.

While this paper shares with the previous publication (Fasolino and Tramontana, 2022) the proposed hook-based locator technique, it provides the following three additional research contributions. First, it proposes a novel classification model of Web application layout changes that can be used to systematically generate a benchmark of changes for experimental studies. Second, it presents a validation study that systematically compares the robustness of the hook-based test cases proposed in Fasolino and Tramontana (2022) against test cases based on state-of-the-art and state-of-the-practice locators. Third, we make available on a GitHub repository all the experimental materials we developed to carry out this study, for replication aims.

The remainder of the paper is structured as follows. Section 2 provides background information about template-based web applications and different types of locator proposed in the literature and used in practice. Section 3 presents the technique proposed for the generation of hook-based locators in template-based Web Applications. In Section 4 we present the proposed GUI change classification model, while in Section 5 we illustrate the validation study we performed. In Section 6 related work is presented while Section 7 discusses conclusions and future works.

2. Background

In this Section we report background information about Web apps based on templates and about the types of locators implemented in existing C&R tools or described in the literature. Moreover we discuss the problem of Web apps test case fragility.

2.1. Web apps based on web templates

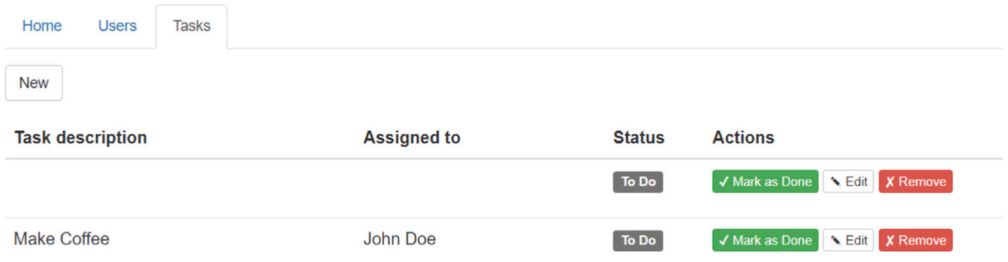
Unlike the technologies for the server-side development of web applications used in the past (such as Java Servlet, or JSP, etc.) (Parr, 2004), most modern solutions are today inspired by the principle of separation between business logic, presentation logic and logic of data, which allows to obtain applications that are more easily developed and maintainable. Currently, many web development frameworks are based on the well-known MVC pattern (or its variants, such as MVVM) that was created precisely to implement this principle. Many modern frameworks also adopt the technology of Web templates and template engines (Parr, 2004) that allow to manage the presentation logic separately from data and business logic.

A template indeed focuses on how to present the data, and other components can focus on what data to present. Template files consist of prewritten markup and template tag blocks where data are inserted. They are created by developers and then processed by the template engines that take in tokenized strings and produce rendered strings with values in place of the tokens as output.

In the following, we report some details about an example Web application developed using templates. The application has been built using the Angular framework,[2] which is an MVC development framework using a JavaScript template engine for generating the Views to be rendered in the client browser. The application is very simple and

[1] Katalon Recorder, https://katalon.com/katalon-recorder-ide/.

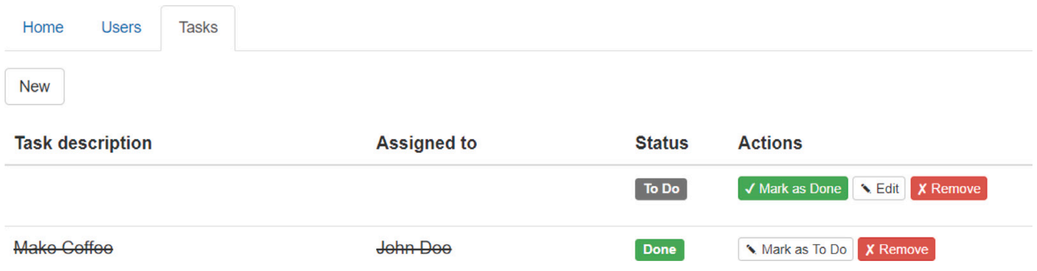
[2] Angular, https://angular.io/.


Home Users Tasks

New

Task description	Assigned to	Status	Actions
		To Do	✓ Mark as Done ✎ Edit ✗ Remove
Make Coffee	John Doe	To Do	✓ Mark as Done ✎ Edit ✗ Remove

Fig. 1. Screenshot of the example Web application showing the 'Make Coffee' task in 'To Do' status.


Home Users Tasks

New

Task description	Assigned to	Status	Actions
		To Do	✓ Mark as Done ✎ Edit ✗ Remove
~~Make Coffee~~	~~John Doe~~	Done	✎ Mark as To Do ✗ Remove

Fig. 2. Screenshot of the example Web application showing the 'Make Coffee' task in 'Done' status.

it allows to manage user tasks by providing features to show the task list, to assign them to different users, and to manage the task status.

Fig. 1 shows the app View reporting the 'Make Coffee' task assigned to 'John Doe' that is in the 'ToDo' status. Fig. 2 represents the View that is obtained after that the user clicks on the 'Mark as Done' action button: in this latter View the status of the 'Make Coffee' task is 'Done' and its text is struck through. Listings 1 and 2 respectively show an excerpt of the HTML source code of the starting page of the Web application and of the HTML template including the task table.

Listing 1: Code excerpt of the Starting HTML Page of the Example Web Application

```
1  <html>
2      ...
3      <body ng-app="myApp">
4          <div class="container">
5              <ui-view></ui-view>
6          </div>
7          ...
8      </body>
9  </html>
```

Listing 2: Code excerpt of a Template of the Example Web Application

```
1  <ui-view>
2   ...
3   <table>
4   ...
5    <tbody>
6     <tr ng-repeat="taskinctrl.tasks">
7     ...
8      <td>
9        <span class="label-default" ng-if="!
             task.done">To Do</span>
10       <span class="label-success" ng-if="!!
             task.done">Done</span>
11     </td>
12     <td>
13      <div class="form-group">
14       <button type="button" ng-if="!!task.
             done" ng-click="task.done=false">
             &#x270E; Mark as To Do</button>
15       <button type="button" ng-if="!task.
             done" ng-click="task.done=true">&#
             x2714; Mark as Done</button>
16       <button type="button" ng-if="!task.
             done" ui-sref=".edit({taskIndex:
             $index})">&#x270E; Edit</button>
17       <button type="button">&#x2718; Remove<
             /button>
18      </div>
19     </td>
20    </tr>
21   </tbody>
22  </table>
23 </ui-view>
```

Observe that line 5 of Listing 1 acts as a reference pointing to the template definition reported from the line 1 of Listing 2. Listing 2 contains the template of the HTML of the table shown in Figs. 1 and 2. This code includes basic HTML tags (such as td, div, button) and Angular keywords (given by tag attributes starting with the *ng* prefix) that are responsible for the different possible appearances of the table. For example, the two action buttons 'Mark as Done' and 'Mark as To Do' can be shown alternatively, on the basis of the values of the boolean *task.done* variable.

2.2. Locators in C&R tools

C&R tools record the user interactions with the GUI of the application under test and translate them into executable test scripts. Afterwards, such interactions can be replayed in order to replicate the recorded execution. In the case of Web applications, such interactions are captured and replayed through a Web browser.

The fundamental problem faced by C&R tools is the generation of *locators* able to uniquely identify the items of the Web page on which the interactions have to be replayed. On the basis of the solution adopted to solve this problem (Ardito et al., 2019), these tools can be classified in "Coordinate-based", which identify GUI items by registering the exact screen coordinates at which the interactions have to be performed, "Visual tools" that identify the GUI items through image matching algorithms like in Sikuli (Yeh et al., 2009), or "Layout-based" ones that implement locators that directly point to specific items of an HTML document.

In Layout-based C&R tools, the most common techniques to identify Web elements in a page are DOM-based (Chapman and Evans, 2011) and locators are usually expressed as XPath expressions (Leotta et al., 2014). XPath expressions allow pointing to different parts of an HTML document, providing a flexible way of navigating through the DOM of the document. They are usually preferred to other technologies for their high expressiveness and for their general applicability (Leotta et al., 2016).

The most popular layout based C&R tools for Web application testing are those offered by the Selenium[3] and Katalon[4] projects. Katalon Recorder is the free C&R tool offered by the Katalon Project, which is built on top of the Selenium open source automation framework. It is able to propose different locators for identifying the involved widgets. An interesting characteristic is that it can also be freely extended by customized locator generation techniques. In this way, the tester can choose the locator he prefers among a range of possible alternatives.

In the following we present the characteristics of different types of locator expressed in XPath: we consider Absolute and Relative locators, locators generated by ROBULA, one of the state of the art techniques (Leotta et al., 2014), and some of the ones generated by Katalon Recorder and Selenium, two of the state-of-the-practice tools. We will illustrate these locators with reference to a pair of common examples. We will show the different types of locators pointing to (1) the *"Mark as Done"* action button of the "*Make Coffee*" task assigned to "*John Doe*" in the Web page reported in Fig. 2, and (2) to the *"Done"* label shown in the row corresponding to the "*Make Coffee*" task reported in Fig. 2.

2.2.1. Absolute locators

An absolute locator can be represented by a XPath expression including references to all the items traversed by a path from the root (i.e., the HTML root tag) to the target element to be located.

When any traversed node presents siblings in the HTML tree, index values can be used to select the correct node among the siblings. The absolute locators for the *"Mark as Done"* button and for the *"Done"* label of the previous example are reported in Listing 3.

Listing 3: Examples of Absolute locators

```
/html/body/div/ui-view/div/ui-view/ui-view/
    table/tbody/tr[2]/td[4]/div/button[1]
/html/body/div/ui-view/div/ui-view/ui-view/
    table/tbody/tr[2]/td[3]/span[1]
```

As it can be seen, absolute locators are very detailed and are able to uniquely point to a specific tag of HTML pages. Of course, any change of the items referenced by the XPath expression may break these locators.

2.2.2. Relative locators

A relative locator specifies a path that does not start from the root node but includes references to the target tag, its parent, or its ancestors and/or attributes or text of these tags that allow to uniquely identify the target element.

Two possible relative locators for the same *"Mark as Done"* button and for the *"Done"* label of the previous Figures are shown in Listing 4. The former locator contains references to the *div* parent tag of the button to be located and to some of its ancestors, whereas the latter one includes only a reference to the exact value of the text contained in the label.

Listing 4: Examples of Relative locators

```
//div[@class='container']/ui-view/div/ui-
    view/ui-view/table/tbody/tr[2]/td[4]/div
    /button[1]
//span[normalize-space()='Done']
```

[3] Selenium, https://www.selenium.dev/.

[4] Katalon Recorder, https://www.katalon.com/katalon-recorder-ide/.

A problem with relative locators is that they may become ambiguous when additional Web elements satisfying the same XPath expression are inserted into the Web page.

2.2.3. ROBULA locators

ROBULA (Leotta et al., 2014) is a technique to build locators by following a top-down approach that starts from the most general locator (i.e., "//*", matching all the elements in the document) and specializes it via transformation steps. The transformation steps start by considering the target tag and adding information about the tag type and attributes into the locator. If a unique locator is obtained, then it is returned by the algorithm otherwise the algorithm repeats the transformations for the parent tag. In the worst case, the algorithm returns an absolute locator. The ROBULA algorithm can be considered as an optimal trade-off between absolute and relative locator.

ROBULA expressions for the *"Mark as Done"* button and for the *"Done"* label are the ones in Listing 5.

Of course, they may break if some of the referenced tag changes and also if the position of the items in the table changes.

Listing 5: Examples of ROBULA locators

```
//tr[2]/td[4]/div/button[1]
//tr[2]/td[3]/span[1]
```

2.2.4. Katalon recorder locators

Katalon Recorder provides several locator generation techniques. The locators provided by Katalon Recorder try to pursue the same objectives as those of ROBULA: they are able to uniquely identify the target element with an expression containing a very limited number of references. In order to obtain this objective they may include references to the target attribute, to its position, to the text it includes and references to the neighbor tags.

Katalon Recorder expressions suggested for the *"Mark as Done"* button and for the *"Done"* label are reported in Listing 6.

Listing 6: Examples of Katalon Recorder locators

```
xpath=(.//*[normalize-space(text()) and
    normalize-space(.)='To Do'])[2]/
    following::button[1]}
xpath=//*/text()[normalize-space(.)='Done']/
    parent::*
```

The first expression locates the button by referring to the preceding item including the *"To Do"* text, whereas the second one individuates the tag including the *"Done"* text avoiding to specify the tag type. The first locator appears fragile to changes of the relative positions of the elements into the page, whereas the second one is not fragile with respect to HTML elements but it can break if the text *"Done"* is translated or if it is added in another part of the page.

2.2.5. Selenium locators

Selenium is one of the most commonly used tools for recording GUI-based test cases via C&R. Similarly to Katalon Recorder, it offers a large set of possible locators for each considered GUI item by using different techniques. In particular, Selenium provides a set of locators consisting in XPath expressions that use tag, attributes, attribute values, included text and other properties to uniquely locate a GUI item. Of course, not all locators are available for each GUI item (e.g. XPath locators including the *href* attribute can be only applied on anchors). In addition, Selenium provides a set of locators based on CSS style properties (i.e. on the values of the *class* attributes).

Selenium expressions suggested for the *"Mark as Done"* button and for the *"Done"* label are reported in Listing 7. The first expression locates the button by referring to the position of the button in the Web page, whereas the second one is based on the existence of the text *"Done"* in a *span* tag.

Listing 7: Examples of Selenium locators

```
xpath=(//button[@type='button'])[2]
xpath=//span[contains(.,'Done')]
```

The first locator appears fragile, for example, to changes in the order of the buttons, whereas the second one is fragile with respect to changes of the container tag (e.g. if *span* is substituted by *div*) or in text changes (i.e. abbreviations or translations).

Examples of the CSS-based locators for the same two GUI items are reported in Listing 8. Of course, the potential robustness of these locators is based on the internal design of the Web application: if each GUI item has its unique style that is never changed, these locators may not break.

Listing 8: Examples of Selenium locators based on CSS

```
css=.btn-success
css=.label
```

3. Hook-based locators and the technique for generating them

A *Hook* can be defined as an artificial attribute inserted into a HTML tag. A *Hook-based locator* is an XPath query including references to hook attributes. Hook-based locators have been proposed in Fasolino and Tramontana (2022) as an alternative category of locators, applicable to Web applications developed using the template technology. These locators require that a hook attribute, characterized by the property that its name is uniquely defined in the context of the Web application, is associated to each tag of each template and each static HTML page of a Web application. Hook-based locators are thus a particular type of XPath queries including references just to the hooks of the widgets to be pointed out and possibly to selected hooks of templates and containers including it. Hook attributes can be injected in the HTML tags of the Web pages by means of an automated technique, named *Hook Injection*, and the Hook-based locators can be generated by a *Hook-based locator generation technique* that will both be presented in the following subsections.

3.1. Hook injection technique

This technique aims at injecting two different types of hook in the tags belonging to each template and each static HTML page of a Web application. The former type of hook is associated with the root tag of each of them. In this case, the hook is defined by a constant string made by the prefix "*x-test-tpl-*" and a variable suffix made by a positive integer number. The suffix is uniquely defined for each root tag of the web application components using a consecutive integer numbering. The latter type is associated with each template HTML tag and is defined by a constant string made by the prefix "*x-test-hook-*" and a variable suffix made by a positive integer number. The suffix part is also uniquely defined for each tag of the web app components using a consecutive integer numbering. The hook attributes do not affect the normal behavior of the Web application and do not conflict with possible HTML *id* or *name* attributes already assigned to the tags.

Listing 9 shows an excerpt of the modified code of the starting page reported in Listing 1 that is obtained after the template hook injection. Analogously, Listing 10 shows the excerpt of the refactored code of the example template reported in Listing 2, where hooks have been injected for each HTML tag and in the template definition tag (at line 1).

Listing 9: Code Excerpt of the Starting HTML Page Generated after Hook Injection

```
<html x-test-tpl-1>
    ...
<body ng-app="myApp" x-test-hook-7>
    <div class="container" x-test-hook-8>
        <ui-view x-test-hook-9></ui-view>
    </div>
    ...
</body>
</html>
```

Listing 10: Code Excerpt of the Refactored Example Template

```
1  <ui-view x-test-tpl-62>
2   ...
3   <table x-test-hook-65>
4   ...
5    <tbody x-test-hook-72>
6     <tr ng-repeat="taskinctrl.tasks" x-test-hook-73>
7     ...
8      <td x-test-hook-76>
9        <span class="label-default" ng-if="!task.done" x-test-hook-77>To Do</span>
10       <span class="label-success" ng-if="!!task.done" x-test-hook-78>Done</span>
11     </td>
12     <td x-test-hook-79>
13      <div class="form-group" x-test-hook-80>
14       <button type="button" ng-if="!!task.done" ng-click="task.done=false" x-test-hook-82>&#x270E; Mark as To Do</button>
15       <button type="button" ng-if="!task.done" ng-click="task.done=true" x-test-hook-81>&#x2714; Mark as Done</button>
16       <button type="button" ng-if="!task.done" ui-sref=".edit({taskIndex:$index})" x-test-hook-83>&#x270E; Edit</button>
17       <button type="button" x-test-hook-84>&#x2718; Remove</button>
18      </div>
19     </td>
20    </tr>
21   </tbody>
22  </table>
23 </ui-view>
```

The injection technique is applicable not only to templates not presenting previously injected hooks, but also to the ones already including hooks and needing to be refactored after a maintenance intervention. In the latter case, the technique is able to preserve existing hooks. The *HookInjection* procedure is described in Algorithm 1. The first loop in the upper part of the procedure searches for the maximum integer suffix number (h_{max}) used by the already existing template and tag hooks. Thus, in the latter lines of the procedure the $newTemplateHook(h_{max})$ function is used to create a new hook for each tag not already associated with its hook. This function will use increasing values in the suffix part of the hooks. Analogously, the $newTagHook(h_{max})$ function is used to create a new hook for each generic tag not already associated with a hook. This function will also use increasing values in the suffix part of the hooks.

3.2. Hook-based locator generation

To locate the widgets of a given Web page, the *HL (hook-based)* locators will consist of XPath expressions including references to the hook of the target widget and possibly to selected hooks of templates and containers including it. The reason for also including template and container hooks in the XPath query is for guaranteeing the unique

ALGORITHM 1: Hook Injection Algorithm

```
Procedure HookInjection(in : webApplication)
    $h_{max} \leftarrow 0$ ;
    foreach $t \in webApplication.templates$ do
        if $exists(t.hook)$ then
            if $t.hook.number > h_{max}$ then
                $h_{max} \leftarrow t.hook.number$ ;
            end
        end
        foreach $tag \in t.tags$ do
            if $exists(tag.hook)$ then
                if $tag.hook.number > h_{max}$ then
                    $h_{max} \leftarrow tag.hook.number$ ;
                end
            end
        end
    end
    foreach $t \in webApplication.templates$ do
        if $!exists(t.hook)$ then
            $h_{max} \leftarrow h_{max} + 1$ ;
            $t.hook \leftarrow newTemplateHook(h_{max})$ ;
        end
        foreach $tag \in t.tags$ do
            if $!exists(tag.hook)$ then
                $h_{max} \leftarrow h_{max} + 1$ ;
                $tag.hook \leftarrow newTagHook(h_{max})$ ;
            end
        end
    end
```

identification of the widget to be located in all those cases in which the template engine dynamically includes multiple instances of the same template in a given page, or multiple instances of the same widget in a given container. For example, the *ng-repeat* attribute shown at line 6 of Listing 10 will produce several instances of the *tr* tag having the same hook in the generated pages.

The technique for generating Hook-based locators traverses the page the widget belongs to, in order to find the hooks that will be included in the XPath query. Heuristic rules are used to select the hooks to be included. Algorithm 2 shows the Hook-based Locator Generation strategy.

ALGORITHM 2: Hook-based Locator Generation Strategy

```
Procedure LocatorGeneration(in: e, out:pathLocator)
    $pathLocator \leftarrow$ "" ;
    while $!e.isRoot()$ do
        if $pathLocator ==$ ""$||e.hasSiblings()||e.isTemplate()||e.includesTemplates()$ then
            if $e.hasSiblings()$ then
                $pathElement \leftarrow indexedElementLocator(e.hook, e.findIndex())$;
            end
            else
                $pathElement \leftarrow elementLocator(e.hook)$ ;
            end
            $pathLocator \leftarrow pathElement + pathLocator$ ;
        end
        $e \leftarrow e.super$ ;
    end
```

The algorithm takes as input the HTML element *e* for which the locator has to be generated and returns as output the *pathLocator* XPath expression. It iteratively builds the *pathLocator* string by backward navigating the hierarchy of the input element *e* from its tag to the root of the page the element *e* belongs to. The string is built from right to left, starting from the hook associated with the input element *e*. During the navigation, the hooks of encountered elements that are considered relevant for the generation of the locator are added to the XPath query. Relevant hooks are defined on the basis of heuristic rules.

At the first iteration, the *pathLocator* is empty and the hook associated with the element to be located will be added to the *pathLocator*. At each successive iteration, the algorithm evaluates if one of the following three conditions is satisfied by the currently analyzed element:

- If the element has some siblings having the same hook (by the *hasSiblings* function), the *findIndex* function evaluates the index number of that element. Therefore, the *indexedElementLocator* function generates a new *pathElement* to be added to the *pathLocator* that both takes into account the hook of the element and its index.
- In the case the analyzed item is a template, a *pathElement* including the hook associated with that template is added to the *pathLocator*. This rule is intended to solve hook homonymy problems which may derive from the integration in the project of homonym hooks from different templates of different projects. The template items are recognized thanks to the *isTemplate* function that evaluates the hook attribute name.
- In case the analyzed item is an HTML element including a template (such as the *ui-view* tag on Line 5 of Listing 9), a query referring the hook of this tag is added to the locator string in order to distinguish between the different template instances. The *includesTemplates* function recognizes these occurrences by observing the hooks of the element and of its sons.

The proposed algorithms for hook injection and locator generation were implemented in the tool-chain presented in Fasolino and Tramontana (2022).

3.3. Example

With respect to the examples reported in Section 2, the obtained XPath expressions for the *"Mark as Done"* button and for the *"Done"* label are shown in Listing 11.

By analyzing the first XPath expression from the right, the first element is *//*[@x-test-hook-81]* that refers to the "*Mark as Done*" button using the hook defined at Line 15 of Listing 10.

The successive element in the expression is the *//*[@x-test-hook-73][2]* one that refers to the *tr* tag on Line 6 of Listing 10. The *ng-repeat* Angular directive on that line may cause the instantiation of multiple rows of the table, all having the same *//*[@x-test-hook-73]* hook. For this reason, the Hook Locator generation algorithm individuates that the row to be located in this case is the second one and adds the corresponding indexed expression *//*[@x-test-hook-73][2]* to the resulting XPath.

Listing 11: Examples of Hook-based locators

```
1  //*[@x-test-tpl-1]//*[@x-test-hook-9]//*[@x-
       test-tpl-62]//*[@x-test-hook-73][2]//*[
       @x-test-hook-81]
2  //*[@x-test-tpl-1]//*[@x-test-hook-9]//*[@x-
       test-tpl-40]//*[@x-test-hook-49]//*[@x-
       test-tpl-62]//*[@x-test-hook-73][2]//*[
       @x-test-hook-78]
```

The third element from the right is the *//*[@x-test-tpl-62]* that refers to the template including the button. The corresponding hook is the one reported at line 1 of Listing 10. The fourth element of the expression is *//*[@x-test-hook-9]* that refers to the *ui-view* tag included at line 5 of the starting page, as reported in Listing 9. This expression refers to the tag causing the inclusion of the *ui-view* template defined in Listing 10. Finally, the leftmost element refers to the *html* tag of the starting page reported on Line 1 of Listing 9. The second XPath expression can be analyzed in the same way. We can observe that it refers to a different template included in the same page (identified as *//*[@x-test-tpl-40]*).

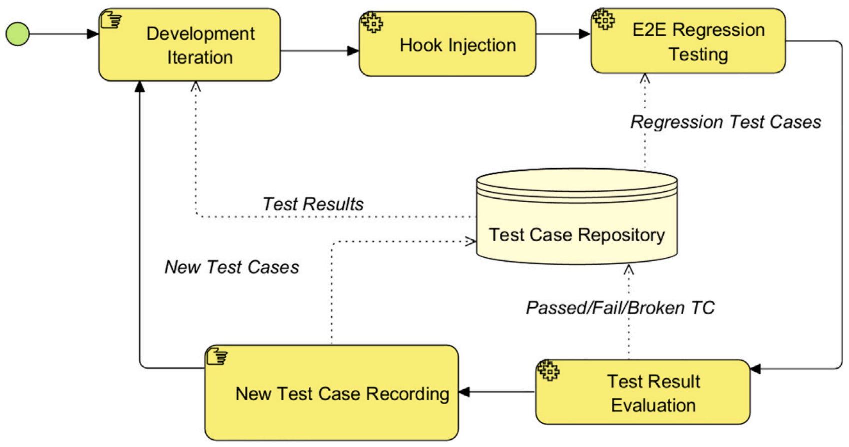


Fig. 3. An example of iterative development process using Hook-based locators.

3.4. Using Hook-based locators in End-to-End regression testing processes

In an End-to-End (E2E) testing process, the application is tested as a whole from the perspective of the end-user. C&R tools allow to record E2E test cases and automatically generate test scripts that can also be re-executed in regression testing. The hook-based technique presented in this Section provides an alternative type of locator with respect to the ones implemented by state-of-the-practice tools, like Selenium and Katalon Recorder.

In our previous work (Fasolino and Tramontana, 2022) we presented a possible implementation of an iterative development process where the E2E Regression testing activity exploits hook-based locators. For clarity, we report in Fig. 3 the description of a generic iteration of this process that includes five either manual or automatic activities (decorated by different icons in Fig. 3: a hand for manual activities and a pair of gear wheels for automatic activities).

The process starts when a new version of the application has been developed (at the end of a generic *Development Iteration*). This version may implement either new Web application features and/or correct faults introduced in the previous iterations. Three sequential automatic activities are therefore executed: *Hook Injection*, *E2E Regression Testing* and *Test Result Evaluation*. The *Hook Injection* automatically injects unique hooks in the HTML tags, when needed, according to Algorithm 1. The *E2E Regression Testing* consists of the execution of all the E2E regression test cases inherited from previous application versions. These test cases are stored in the *Test Case Repository*. In the *Test Result Evaluation* activity, test cases are classified in three different categories according to the obtained results: (1) passed, (2) failed (i.e. test cases successfully executed but with one or more assertions that fail), or (3) broken test cases (i.e. test cases that did not successfully execute, due to a crash of the test code) and are stored in the *Test Case Repository* as well.

The successive activity (*New Test Cases Recording*) consists in recording new E2E test cases, either needed for covering the additional features of the application, or for substituting broken test cases. This activity is supported by a Capture and Replay tool implementing the hook-based locator generation strategy described by Algorithm 2. The recorded test cases will be pushed in the *Test Case repository*. At the end of this activity, a new development iteration will start. This example of process iteration shows how the hook injection step is the only additional activity needed before the execution of a traditional E2E regression testing process.

4. Layout change model for template-based Web applications

Hammoudi et al. (2016) proposed a taxonomy of proximal causes of locator breakages based on the empirical observation of test breakages in real Web applications. In their taxonomy they differentiated between breakages due to modifications in the source code of the applications (e.g. modifications in attributes, tags, texts), breakages due to the application behavior (e.g. JavaScript exceptions), and breakages due to browser behavior (e.g. page reload, user session timeout, pop-up windows). The first type of causes was the predominant one, covering 73.62% of observed test breakages, and in the rest of the paper we will focus on them.

Table 1
Change classification model.

Dimension	Possible values
Types of Object	Tag, Tag Attribute, Tag Attribute Value, Text, Template Tag
Types of Change	Removal, Modification, Insertion, Position Change
Relationships	Parent–Child, Ancestor–Descendant, Sibling, Includes

Since that taxonomy reports generic modifications in attributes, tags, and textual contents that can cause locator breakages, we decided to develop a more detailed classification of changes that may cause such breakages. Our classification considers three dimensions for characterizing a layout change of an element of an HTML file: the first dimension specifies the *Type of Object* involved in the change (that may be a tag, a tag attribute, a tag attribute value, a text, or a template tag). The second dimension specifies the *Type of Change*, that may consist in either a *Removal*, a *Modification* of an existing Object, the *Insertion* of a new Object or the *Position Change* of an existing object to another point of the file. The latter dimension consists of the *Relationship* between the modified Object and the Object that is involved in the test case and needs to be detected. Possible relationships include the "*parent–child*", "*ancestor–descendant*", and "*sibling*" relationship between Objects. A fourth relationship is the "*includes*" one between the template that defines the Object to be detected and the same Object. Of course, changes may also involve the object itself. Table 1 summarizes the three dimensions we use to classify changes that potentially cause breakages and the possible values of each dimension.

It is easy to observe that, while the majority of combinations of these values are representative of feasible changes, a minority of them will be meaningless. For example, if we consider a "*Tag Attribute Value*" Object, it will be possible to define a "*Removal*" or a "*Modification*" type of Change, whereas the "*Insertion*" is not applicable (attributes are not structured data), and the "*Position Change*" is irrelevant for attributes.

Table 2 presents meaningful combinations of types of *Object* and types of *Change* derivable from the Model and some clarifying examples of them. Each row reports the type of Object involved in the change, a *Change Id* and its *Description*, and an example of *Original Code* before implementing that change, and eventually the *Modified Code*. In the Table, the *h* Change Id corresponds to changes that are specific to

Table 2
Example of changes for different *Types of Object* and *Types of Change* included in the classification model.

Object type	Change ID	Change type description	Original Code	Modified Code
	a	Attribute value modification	< *tag attr* = “*value*” > *text* < /*tag* >	< *tag attr* = “*newValue*” > *text* < /*tag* >
Attribute	b	Attribute removal	< *tag attr* = “*value*” > *text* < /*tag* >	< *tag* > *text* < /*tag* >
	c	Attribute identifier modification	< *tag attr* = “*value*” > *text* < /*tag* >	< *tag newAttr* = “*value*” > *text* < /*tag* >
Text	d	Text content modification	< *tag attr* = “*value*” > *text* < /*tag* >	< *tag attr* = “*value*” > *newText* < /*tag* >
	e	Text content removal	< *tag attr* = “*value*” > *text* < /*tag* >	< *tag attr* = “*value*” >< /*tag* >
	f	HTML tag movement (within a container)	< *div* > < *tag* >< /*tag* > < *tag attr* = “*value*” > *text* < /*tag* > < /*div* >	< *div* > < *tag attr* = “*value*” > *text* < /*tag* > < *tag* >< /*tag* > < /*div* >
	g	HTML tag movement (in any point of the HTML tree)	< *div* > < *tag attr* = “*value*” > *text* < /*tag* > < *tag* >< /*tag* > < /*div* >	< *tag attr* = “*value*” > *text* < /*tag* > < *div* > < *tag* >< /*tag* > < /*div* >
Tag	h	HTML tag movement (between two templates)	< *template*1 > < *tag attr* = “*value*” > *text* < /*tag* > < /*template*1 > < *template*2 > < /*template*2 >	< *template*1 > < /*template*1 > < *template*2 > < *tag attr* = “*value*” > *text* < /*tag* > < /*template*2 >
	i	HTML tag removal	< *tag attr* = “*value*” > *text* < /*tag* >	*text*
	j	HTML tag type modification	< *tag attr* = “*value*” > *text* < /*tag* >	< *newTag* *attr* = “*value*” > *text* < /*newTag* >
	k	HTML tag insertion	< *tag attr* = “*value*” > *text* < /*tag* >	< *tag* >< /*tag* > < *tag attr* = “*newValue*” > *text* < /*tag* >

template-based Web applications, whereas the other ones may also be applied to traditional HTML pages.

Regarding the *Relationship* dimension of the classification model, it can be used to define changes that involve objects having different types of relationship with the tag to be pointed to. As an example, the change may involve the item itself, an ancestor, the parent, a sibling of the item, or the Template including it. In the remainder of the paper, we will say that the relationship is of type (α), if the change involves the item to be located itself, it is of type (β) if it involves the parent of the item, of type (γ) if it involves an ancestor, of type (δ) if it regards a sibling, and of type (ϵ) if it regards the template the item belongs to.

Listing 12 reports an excerpt of HTML code to show some examples of relationship values. With respect to this Listing, if we consider the anchor tag < *a onclick* = “*go*()” > at line 4 as the target of a test case, depending on the item that will be changed, we will have a different value in the Relationship dimension. As an example, if the change regards the same anchor tag, the Relationship is of type (α), whereas a change regarding the < *div* > at line 3, will be characterized by a relationship of type (β). The Listing also illustrates examples of the other three types of relationship.

Listing 12: An Excerpt of a HTML Code Showing Tag Classification with respect to the Target Tag (in line 4)

```
1  <template>                          <!-- ε (template)   -->
2      <p>                             <!-- γ (ancestor)   -->
3          <div>                       <!-- β (parent)     -->
4              <a onclick="go()">      <!-- α (target tag) -->
5              <a href="\#top">        <!-- δ (sibling)    -->
6          </div>
7      </p>
8  </template>
```

This model can be used to select a subset of relevant types of change we may be interested to implement in a subject Web application. Consequently, different benchmarks of implemented changes could be defined, depending on the chosen types of model changes.

5. Experiment

In this section we present an experiment we performed to investigate the capability of Hook-based locators to prevent regression test breakages. To this aim, we considered the Layout change classification model proposed in Section 4 to define a benchmark of exemplar changes in Web application layout, implemented these changes in real Web applications and evaluated the fragility of regression tests with respect to the modified applications. We also compared the fragility of test cases based on the proposed Hook-based locators against the ones based on state-of-the-art and state-of-the-practice locators.

5.1. Research questions

Our study investigates two research questions.

RQ1 How does the robustness of test cases using Hook-based locators vary when different types of Layout change are implemented in a Web application?

RQ2 How does the robustness of test cases using Hook-based locators compare to the robustness of test cases using other types of locators, when different types of Layout change are implemented in a Web application?

With the first research questions we aim at evaluating to what extent test cases based on Hook-based locators are resilient to different types of common layout changes of the Web application. The second one, instead, aims at comparing the resilience of these test cases against the one of analogous test cases exploiting diverse types of locators.

5.2. Experimental objects

The experiment involved two open-source template-based Web applications implemented by the Angular framework, named A1 and A2 hereafter. We selected two applications as a sample of real applications to be artificially modified for introducing a systematic set of typical GUI changes which may break locators.

A1 is a small application that allows to manage a contact list. It offers functionalities for adding a new contact, showing the list of contacts, and saving the list in a local database. The application includes 3 templates and a static HTML page. It has been forked from a tutorial example published on GitHub.[5]

A2 is an application that provides an Angular-based clone of the Spotify web client. The source code of the application is available on GitHub[6] where it counts about 300 forks and more than 2K stars. This application, like the real Spotify client, provides multiple functionalities, such as the possibility to search for an artist or to display the most popular songs. It is composed of 9 HTML pages and 47 templates.

5.3. Experimented locators

In the experiment we considered 7 types of locators, namely Absolute, Relative, ROBULA, Katalon, Hook-Based, besides two locators provided by Selenium (respectively, XPath based and CSS based).

The first five locators were implemented by Katalon Recorder that provides the option of adding *"Extension Scripts"* that allows to implement new user-defined locators besides the ones offered by the tool.[7] The implementation of these locators has been made available, together with all the experimental materials in a GitHub repository.[8]

As to the Absolute and Relative locators, we implemented them according to the definitions provided in the Background Section. As to ROBULA locators, we implemented this technique on the basis of the ROBULA algorithm described in Leotta et al. (2014). As to Katalon locators, we decided to always consider the first one among the ones natively proposed by the tool (version 5.9.0[9]) for the following reason. During the test case recording step, the tool provides a number of alternative locators associated with the action being recorded. The number, type and order of offered alternative locators is variable, depending on the specific type of object. In addition, Katalon does not explicitly label the type of the proposed locator. As a consequence, to avoid non-determinisms and assure the repeatibility of our experiment, we decided to always select the first locator proposed by the tool.

As regards Selenium, we have used the version 3.17.2.[10] We chose to consider two types of locator, the former being based on XPath and the latter being based on CSS style attributes. Both types of locator were presented in the Background Section. Since Selenium usually provides more XPath-based locators, we had to select one of them. We preferred the XPath-based locators referring to specific attributes and values (when possible), rather than the ones referring on text and tag hierarchy, since we considered them more promising based on our experience as testers. In the following we will refer to the former Selenium locator as "S" and the latter one as "C".

5.4. Experimental procedure

The experimental procedure we followed included 5 steps. Fig. 4 describes the flow of these steps and the produced artefacts. Automated steps are reported as rounded boxes with red background while manual steps are shown as rounded boxes with blue background, and the produced artefacts are represented as document-shaped boxes. The automated steps were implemented using GitHub Actions in YAML scripts, while the artefacts were stored in a GitHub repository.

Table 3
Test case summary.

	#Actions	#Assertions	#Locators	#Pages
Test case for A1	6	2	8	1
Test case for A2	8	1	9	6

In the first step we automatically injected hooks in the source code of the considered applications using the automated Hook-Injector component presented in Fasolino and Tramontana (2022). This component is able to operate on the source code of template-based Web applications implemented by several different frameworks such as Angular,[11] Freemarker,[12] Twig[13] and Smarty.[14] Of course, the tool can operate also on pure HTML pages. We injected 56 hooks in A1 and 234 in A2, obtaining two updated versions of the original applications. The effort required by the component to execute this step was actually negligible, being of 4 ms and 9 ms, respectively, for A1 and A2.

Therefore, for both applications, we manually recorded a test case consisting of a sequence of actions (such as click, fill-in a text field, etc.) to be performed on different Web page items and assertions to check the test results. In the first application (A1) we recorded a test case that inserts a new contact in the contact list and visualizes the updated list. In the second application (A2), the recorded test case includes the authentication of the user (via login and password), the search for an artist by name, the selection of one of its songs from the returned song list and the navigation to the Web page related to the album including the selected song.

We recorded seven different versions of the same test case, each of them using one of the different types of locator reported in Section 5.3. Test cases were recorded using Katalon Recorder and Selenium. Table 3 reports summary data about each test case including the number of actions, number of assertions, number of locators, and number of Web pages traversed by the test case.

We also had to design and implement a set of modified app versions. To this aim we defined a benchmark of exemplar changes for the web applications under test, according to the change classification model reported in Section 4. Our benchmark had to cover at least once all the types of change reported in Table 2 and possible Relationships defined in the change model.

Each new modified version of the Web application had to include a single change. As to the implementation of the changes, we conveniently chose some GUI item associated to locators included in the recorded test case and applied them the types of change compatible to that specific type of object. As an example, for a "tag attribute" type of object, we could define 3 types of changes (namely a, b, c in Table 2). In another type of change, we modified one of the neighbors of the item to be located by the test case. As an example, with respect to the previous "tag attribute" example, we also applied the three changes to an attribute included in the father, an ancestor, and a sibling of the item to be located, as well as to an attribute of the template including the item to be located.

Tables 4 and 5 show a summary of the changes we applied. Table 4 reports, for each application under test, the ID of the application, the total number of different versions and the number of different versions subdivided with respect to the 11 different types of changes (a .. k). Table 5, instead, shows the number of different versions classified with respect to the 5 different categories of involved items relationships (α .. ϵ). The different count values in the tables are due to the non-applicability of some specific types of changes on the two applications.

[5] https://github.com/bbachi/angular-java-example.
[6] https://github.com/trungk18/angular-spotify.
[7] Katalon Recorder Extensions Scripts, https://docs.katalon.com/docs/plugins-and-add-ons/katalon-recorder-extension/get-your-job-done/extend-katalon-recorder/extension-scripts-aka-user-extensions.js-for-custom-locator-builders-and-actions-in-katalon-recorder.
[8] Experimental Material, https://github.com/reverse-unina/LocatorsFragilityExperimentation.
[9] Katalon Recorder v.5.9.0, https://chrome.google.com/webstore/detail/katalon-recorder-selenium/ljdobmomdgdljniojadhoplhkpialdid.
[10] Selenium IDE v. 3.17.2, https://chrome.google.com/webstore/detail/selenium-ide/mooikfkahbdckldjjndioackbalphokd.
[11] Angular, https://angular.io/.
[12] Apache Freemarker, https://freemarker.apache.org/.
[13] Twig PHP Template Engine, https://twig.symfony.com/.
[14] Smarty PHP Template Engine, https://www.smarty.net/.

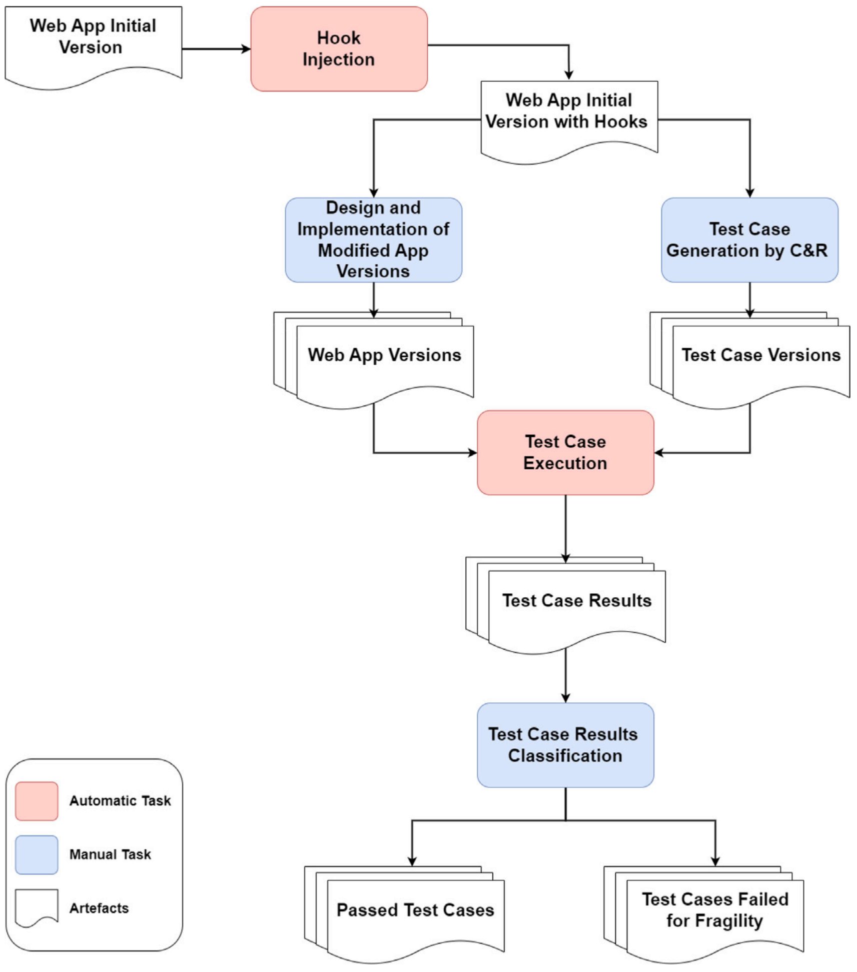

Web App Initial Version
Hook Injection
Web App Initial Version with Hooks
Design and Implementation of Modified App Versions
Test Case Generation by C&R
Web App Versions
Test Case Versions
Test Case Execution
Test Case Results
Test Case Results Classification
Passed Test Cases
Test Cases Failed for Fragility
Automatic Task
Manual Task
Artefacts

Fig. 4. Overview of the experimental procedure. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

Table 4
Implemented changes with respect to change types.

Web	# Versions	Count of versions wrt the change types										
application		a	b	c	d	e	f	g	h	i	j	k
A1	100	12	12	12	6	6	9	8	9	9	9	8
A2	92	10	10	10	4	4	10	10	9	8	9	8

Table 5
Implemented changes with respect to the involved items relationships.

Web	# Versions	Count of versions wrt the involved items relationships				
application		α	β	γ	δ	ϵ
A1	100	25	18	22	22	13
A2	92	18	17	17	22	18

The modified versions of the Web pages/templates were manually implemented and stored in two GitHub repositories that we made available[15][16] and that also report the results of the test executions.

As to the step of Test Case Execution, we exploited the GitHub Actions through the definition of a YAML script that allowed us to setup the same test case execution environment for each test case. The average times needed for executing each test case was 15 s, for the A1 app, and 17 s for the A2 app. As it can be seen, these execution times were sensibly longer than the times required by the hook-injection step that we reported at the beginning of this Section.

Finally, in the Test Case Results Classification step, we manually analyzed the test case results in order to obtain passed test cases and test cases failed for fragility issues.

[15] A1 repository, https://github.com/reverse-unina/A1-ContactList.
[16] A2 repository, https://github.com/reverse-unina/A2-Spotify.

5.5. Variables and metrics

The independent variable of the study is the type of locator *loc* used by test cases. The *loc* considered values are:

A	- Absolute locator
R	- Relative locator
RO	- Locator generated using the ROBULA technique
K	- Locator provided by Katalon
H	- Hook-based locator
S	- XPath-based Locator provided by Selenium
C	- Locator provided by Selenium based on CSS style attributes

The dependent variable we consider is the *number of test cases broken for fragility issues*. In order to evaluate this metric we need to distinguish between test cases broken due (1) to obsolescence or (2) to fragility of the locators. The obsolescence of a test case indicates that the test is no more applicable and has to be abandoned, whereas in the other cases the breakage of the test case was unexpected. For example, if a feature is removed from the Web page (e.g. its enabling button is removed),

Table 6
Test case execution results for the two AUTs.

Test case	A1 : Contact list							A2 : Spotify						
execution	Locator							Locator						
results	A	R	RO	K	S	C	H	A	R	RO	K	S	C	H
Passed	58	86	83	74	77	74	88	47	49	64	59	76	69	79
Failed for Obsolescence	7	7	7	7	7	7	7	8	8	8	8	8	8	8
Failed for Fragility	35	7	10	19	16	19	5	37	35	20	25	8	15	5
Total	100	100	100	100	100	100	100	92	92	92	92	92	92	92

then the test case has to be considered *obsolete* because the test case is no more executable. On the other hand, if other types of layout changes occur (e.g. the enabling button is moved to a different position or its text is changed), then the test case should remain executable. Of course, the obsolescence of a test case does not depend on the locator types and has to be manually assessed.

5.6. Experimental results

Table 6 reports, for each of the applications under test, the results of the execution of the test cases based on the 7 types of locators (*A*, *R*, *RO*, *K*, *S*, *C*, and *H*). Each row of the Table shows, for each type of locator, the cumulative number of passed test cases (*Passed*), those that failed due to obsolescence (*Failed for Obsolescence*), and those that failed due to locator fragility (*Failed for Fragility*).

As the Table shows, for the A1 application the test cases that failed the most for fragility issues were the ones based on Absolute locators (35 failures), whereas the test cases with Hook-based locators proved to be the most robust, failing only 5 time out of 100.

Analogous results were observed with respect to the second application, A2, where the most robust test cases were always the ones based on hook locators, which failed for fragility only 5 times out of 92 executed test cases. The least robust test cases were the ones based on Absolute and Relative locators (with 37 and 35 failures, each), followed by the ones based on Katalon, ROBULA and Selenium locators.

In order to comprehend to what extent the robustness of test cases based on different locators varied with the types of change implemented in the Web applications, we analyzed in detail the observed test breakages. Table 7 reports two Fragility Maps that associate each considered type of change with the type of locators that caused the breakage of a test case for fragility issues at least once, respectively in A1 (in Table 7a) and A2 (in Table 7b). Each table cell corresponds to a specific type of change: the row identifies the type of Layout change (using the same Change IDs presented in Table 2), while the column identifies the Relationship between the modified element and the target tag (using the notation shown in Section 4). Each cell contains one or more IDs, corresponding to any of the 7 considered types of locators (A, R, RO, K, S, C, H) for which at least a failure was observed. A dash (–) in a cell indicates a "Don't Care" case with respect to the fragility. In other terms, we did not detect fragility issues of all locators in that case, either because it was not possible to apply that type of change, or because all the test cases failed due to obsolescence. Eventually, a √ symbol indicates that all the test cases were robust with respect to that type of modification in the context of the executed test cases. A detailed list with all the executed test cases and their execution results is reported in the experimental material.[17]

In order to provide an answer to RQ1, we focused on the results achieved by test cases adopting the Hook-based locators. As Table 6 shows, there were only 5 hook-based test cases in A1 and 5 in A2 that failed by fragility (against overall 167 passed test cases). As the Fragility Maps show, all these breakages were due to changes involving templates. In details, a total of 6 breakages (4 in A1 and 2 in A2) were due to changes of the position of HTML tags between two templates (change type *h*), 2 breakages (one for each application) were due to the removal of a template container (change type *i*) and 2 breakages in A2 were caused by a position change of the template tag (change type *g*). This result was not surprising, because Hook-based locators always depend on template references. We point out that also Absolute locators were fragile to these type of changes, whereas Relative, ROBULA, Katalon Recorder and both the Selenium locators often did not break because they do not always have references to template ancestors.

[17] Experimental Material, https://github.com/reverse-unina/LocatorsFragilityExperimentation.

Table 7
Maps of the test fragility issues for A1 and A2.

(a) Fragility Map for A1

Layout	Relationship with the target tag				
change	α	β	γ	δ	ϵ
a	R,RO,C	C	√	S,C	√
b	R,RO,S,C	C	√	√	√
c	R,RO,C	C	√	√	√
d	R,K	–	√	K	–
e	R,K	–	√	K	K
f	A,RO,K,S,C	A,C	A,S,C	A,K	A,K,S,C
g	A,K,C	A	A,K,S,C	K	–
h	A,K,S,H	A,S	A,H	√	A,S,H
i	√	A,S,C	A,S,H	K	A
j	A,R,RO,K,S	A,S	A	√	A
k	–	A,S,C	A	A,RO,K,S	A

(b) Fragility Map for A2

Layout	Relationship with the target tag				
change	α	β	γ	δ	ϵ
a	RO,K,C	K,S,C	C	K	R,K
b	RO,C	C	C	√	R
c	RO,C	C	C	√	R
d	R,K	–	–	R,K	–
e	R,K	–	–	R,K	–
f	A,R,RO,K	A,R,RO,K	A,R,RO,K	A,R,RO,K	A,R,RO,K,H
g	A,R,RO,K,S,C	A,R,RO,K,S	A,K,H	A	A,R,K
h	–	A,R,RO,K,H	A,H	√	–
i	–	A,R,RO,K	A,R,RO,H	√	A,R,K,C
j	A,R,RO,K,S	A,K	A	√	A,R
k	–	A,R,RO,K	A,R,RO	A,R,RO,K	A,R

Legenda:
Lowercase letters from a to j on first column: layout change types
Uppercase letters from α to ϵ on header row: relationships with the target tag
A, R, RO, K, S, C, H: Absolute, Relative, ROBULA, Katalon, Selenium, Selenium CSS, Hook-based locators
–: inapplicable changes or test cases failed by obsolescence
√: no breakages at all.

On the basis of the observed results, we could answer the RQ1 research question as follows:

> *With respect to the considered layout changes, test cases using Hook-based locators can be considered robust to all layout changes, except the ones consisting in moving a tag between two templates, or template changes.*

In order to answer to RQ2, we analyzed the breakages of test cases based on all the 7 considered locators and compared them. We ranked the different types of locator on the basis of the growing number of observed test breakages. After the Hook-based locators that ranked first, the second most robust ones were the X-Path based ones generated by Selenium. Selenium is probably the best tool in the current state-of-practice, thanks to the multiplicity of supported techniques for generating locators.

Only 24 test cases based on XPath locators proposed by Selenium failed, whereas 153 did not break after the implementation of the layout changes. Selenium supports many effective strategies for generating robust locators: its locators exploit *id* or *href* attributes when available and tends to prefer the inclusion in its XPath expressions of properties of the target tag or of its closer tags. For this reason, locators proposed by Selenium failed especially in those cases in which the modifications involved the value of attributes such as *id* or *href*, or in which such attributes were not present.

We could conclude that:

Selenium XPath locators were robust to most implemented changes, but they resulted fragile when specific attributes are changed (e.g. id or href) or with changes of the type or position of the target tag.

The third most robust type of locator resulted the one generated by ROBULA, one of the state-of-the-art techniques (Leotta et al., 2014).

In our experiments, 30 test cases using ROBULA locators failed for fragility issues, whereas 147 test cases passed. Most of the failures (17 out of 30) concerned changes to the target tag (relationship type α), while the other failures concerned the parent tag (5 out of 30), sibling tags (4 out of 30) or ancestor tags (3 out of 30). Only in a single case the test case using ROBULA locators failed for a template change. This result was not surprising since the ROBULA algorithm behaves similar to the algorithms included in Selenium and produces compact XPath expressions based on properties of the target tag or of its closer tags. Regarding the considered change types, we observed that ROBULA locators failed for each possible change except those involving text changes (i.e. change types *d* and *e*), because text is not considered by ROBULA locators. We could conclude that:

ROBULA locators are generally fragile to any type of change, especially to those involving the target tag or its nearest neighbors, but are robust to changes involving texts included in tags.

The fourth most robust technique in our experiment was the CSS-based one proposed by Selenium. These locators contain references to the style of the target tag and possibly the parent tag or other ancestors. In the experiment, 34 test cases using CSS-based locators broke after layout changes, whereas 143 remained valid. Most locator breakages concerned changes in the style of the target tag (i.e. the *class* attribute) or of one of its neighbors. In further cases where the target tag had no *class* attribute, layout changes involving the target tag type or movements of this tag within the HTML tree also caused test breakages.

We could conclude that:

CSS-based Selenium locators were fragile mostly with respect to changes involving the web page style, whereas they generally resulted robust when style information used in the page were not involved in the changes.

The fifth ranked category of locators was the one provided by Katalon Recorder that caused 44 test breakages against 133 passed test cases. We had some test breakages for each change type except for the ones involving just attributes (change types *a*, *b* and *c*). This result can be explained by observing that the locators suggested by Katalon Recorder usually include predicates related to the text, to the tag type, and to the neighbors of the target tag but not based on attribute values. On the other hand we have observed test breakages for tags having all the considered relationships types α, β, γ, δ, ϵ. On the basis of these data, we could conclude that:

The locator suggested by Katalon Recorder represents the most robust alternative only when the change involves only attributes.

Finally, we analyzed the performance of the two basic techniques for locator generation, i.e. Relative and Absolute, that ranked second-last and last, respectively. As to Relative locators, we observed overall 42 test case breakages, against 135 passed test cases. More in details, Relative locators caused 7 breakages in A1, against 86 passed test cases, and 35 test breakages in A2, against 49 passed test cases. The worse performance of Relative locators in A2 was due to the fact that this latter application has a more complex user interface, with many dynamically generated HTML elements having no distinctive properties. For this reason, the generated XPath expressions contained many references to different tags of the HTML tree, that made them more fragile.

There was at least a test breakage for each type of change and for each type of relationship. With respect to both applications, the majority of the test cases with Relative locators broke after a change in the target tag (relationship α): with respect to 37 changes of type α that did not cause obsolescence, there were 20 broken test cases against 17 passed ones. Of course, each XPath expression included a reference to the target tag, thus the changes on that target caused location breakages in the majority of test cases. On the basis of these data, we could conclude that:

Relative locators may be fragile to any type of change, in particular to the ones involving the target tag and its nearest neighbors.

Absolute locators represent the easiest solution to the problem of identifying the elements of a test case, but proved to be the most fragile in our experiment. We observed the breakage of 72 test cases against 105 passed test cases. More in details, since Absolute locators are built without taking into account attributes and text, they resulted very fragile with all changes regarding tags and templates (change types *f*, *g*, *h*, *i*, *j*, *k*). By narrowing our analysis to these six types of change, we observed 72 test breakages against only 23 passed test cases. Breakages were observed for each type of considered relationship. We can conclude that:

Absolute locators are fragile with respect to almost all the considered types of change, except those regarding only text and attributes.

In conclusion, with respect to the research question RQ2, the experiment showed that none of the considered locator types is robust to any type of change. Hook-based locators were however the most robust, even if they are exposed to fragility issue in case of changes impacting templates. In the latter cases, the other types of locators resulted more robust, with the exception of the Absolute ones.

If we consider the state-of-the-practice C&R tools Katalon and Selenium, Selenium locators showed to be the most robust. The locators suggested by Katalon Recorder, another state-of-the practice tools, performed worse than the Hook-based, ROBULA and Selenium locators, but they showed to be a valid alternative for changes involving only tag attributes.

The locators produced by the ROBULA technique represent another valid alternative, even if we observed fragility issues with respect to any type of change, in particular when the tags directly involved in the test case are modified. Relative locators were generally not very reliable but have the advantage of being generally the most readable ones, by including a few references centered on the element to be identified and on its closest neighbors. Absolute locators proved to be the most fragile solution in all conditions.

5.7. Threats to validity

In this subsection we discuss possible threats to the validity of our study, according to the classification proposed in Wohlin et al. (2012).

5.7.1. Threats to internal validity

In order to execute all the test cases under the same conditions, we run all of them in an execution environment consisting of a container offered by GitHub and configured by a YAML script that defined the use of the same Ubuntu version (22.04.2) and the same headless version of Chrome (110.0.5481.77). A delay of one second was added between each step of each test case in order to reduce the risk of inconsistent test results due to timing issues. Each test case has been executed twice and we have always obtained the same results.

5.7.2. Threats to construct validity

Implementations of Absolute, Relative and ROBULA locator generators were not available in the context of Katalon Recorder so we had to implement and test them before the execution of the experiment. Regarding the Absolute and Relative locators, we implemented generators consistently with their definitions provided in Section 2. Regarding ROBULA, we implemented it by referring to the algorithm reported in Leotta et al. (2014). As for Katalon Recorder, we used the locators provided by the tool. Since Katalon Recorder suggests several types of locator at each request, we decided to always use the first suggested one, in order to avoid non determinism in our experiment. Selenium offers several possible locators for each object (the number and type of different locators vary with the object to be located). As explained in Section 5.4 we have selected the most promising locators based on our experience as testers. As a consequence, we cannot exclude that other testers might choose different locators. All the locator implementations have been made available in the experimental material for replication purposes.[18]

In order to distinguish between test cases broken by fragility or by obsolescence, two authors carefully analyzed the modified versions of the Web applications, in order to evaluate if the functionality involved in the test case could still be successfully executed or if it was obsolete after that layout change.

5.7.3. Threats to conclusion validity

The reported conclusions are based on the subset of considered changes. In order to consider a systematic and realistic set of changes, we have proposed a change model that extends the one previously presented in Hammoudi et al. (2016), which was based on the empirical observation of changes causing test breakages in Web applications. In addition, our extension takes into account specific types of changes that may occur in template-based Web applications.

To make a fair comparison, we tried to define an equal number of changes for each of the considered typologies. However, we implemented fewer changes in some cases because some of them were not always applicable in the considered web applications. In addition, the considered frequency of application changes may not be representative of real change occurrences. In future work, we intend to define a pool of changes with controlled frequencies of change types, on the basis of real change repository mining.

[18] Experimental Material: Implemented Locators, https://github.com/reverse-unina/LocatorsFragilityExperimentation/tree/main/LocatorsImplementation.

5.7.4. Threats to external validity

The first threat to the generalization of our conclusions is due to the small number of considered template-based Web applications. This limitation was essentially due to the effort required to systematically design and implement the benchmark of about 100 GUI changes per application which might break locators. However, even if limited, our sample is representative of real template-based Web applications, since they were selected among popular open-source applications from GitHub.

Another threat depends on the limited number of test cases involved in the study. However, our test cases were defined in order to include several different locators and to involve different pages of the applications under test. In this way, we are confident that the test cases include a representative sample of real interactions with web application widgets which may be involved in realistic web page layout changes.

6. Related work

The problem of locators fragility has been originally studied for Web context extraction (Myllymaki and Jackson, 2002; Kowalkiewicz et al., 2006; Montoto et al., 2011) and Web application wrapping and migration problems (Di Lucca et al., 2007). A broad discussion about the causes of test case fragility and the techniques and strategies proposed in the literature to solve them at different levels is presented by Ricca et al. in 2019 (Ricca et al., 2019). It is possible to summarize that the problem of the fragility of the locators has been faced by three distinct approaches: (1) by generating robust locators, (2) by repairing broken locators and (3) by refactoring the Web pages to enable the generation of more robust locators. In the next subsections we will discuss the main contributions in literature regarding any of these three approaches.

Techniques for the generation of robust locators. The first family of techniques aims at improve the process of generation of locators by using heuristics generation techniques on the basis of the analysis of the DOM of the Web pages observed during the Web application execution.

Thummalapenta et al. (2012, 2013) proposed a tool called ATA (Automating Test Automation) that helped the tester in the development of change-resilient test scripts, and in their maintenance and repairing. In 2014, Yandrapally et al. (2014) proposed a technique to generate robust XPath queries by focusing on attributes that are supposed to be invariant during the application evolution, such as labels or IDs. A similar approach is the one proposed by Pirzadeh and Shanian (2014). They presented a test development framework able to help testers in producing *resilient tests*, i.e. test cases independent on the internal structure of applications, so that a change in the structure will not break them. Their approach is based on analyses of both structure and textual contents of HTML pages. More recently, Kirinuki et al. in 2019 (Kirinuki et al., 2019) presented COLOR (COrrect LOcator Recommender), a technique for the automatic generation of locators which takes into account various properties (i.e., attributes, texts, images, and positions) and their changes between two Web Application releases. In 2021, Nguyen et al. (2021) have proposed another heuristic approach that also considers the neighbors of the Web elements to be located in the DOM hierarchy.

The most relevant contribution to this field in the last years is presented in the series of papers based on the ROBULA tool and on its evolution, starting from 2013. In order to reduce the costs for finding and repairing broken test cases, Leotta et al. proposed in 2014 ROBULA (ROBUst Locator Algorithm) (Leotta et al., 2014), an algorithm and a tool for the generation of robust XPath locators. In 2016 they improved their technique by proposing Robula+ (Leotta et al., 2016), another heuristic technique that overcome some of the limitations of Robula by improving the refinement algorithm prioritizing the use of some attributes that demonstrated their stability on the basis of the previous experiments. The most recent contribution of this series is represented by Sidereal (Leotta et al., 2021), a more complex adaptive approach based on the minimization of a Fragility Coefficient. All these approaches have been tested on real open source Web applications and have brought the number of broken locators between two consecutive releases down to about 10% in the case of Sidereal (Leotta et al., 2021).

Techniques for repairing broken test cases or locators. An alternative approach with respect to the proposal of robust locators is the one of automatically repairs broken locators when a test breakage occurs.

The idea to automatically repair broken test cases of interactive applications was initially proposed in the context of GUI testing of desktop applications by Memon (2008). The first relevant approach in the context of Web testing is the one of Choudhary et al. (2011) that in 2011 proposed the WATER (Web Application TEest Repair) approach to repair locators. Their approach is based on the intuition that a broken locator can be repaired by choosing another locator that is similar to the original one. The similarity between different locators is measured by means of the Levenshtein distance between the corresponding XPath queries. This approach is not more effective when large updates of the Web Application's layout occur. In such a case, the locator may not be correctly repaired even by using WATER. In 2016, Hammoudi et al. (2016) proposed an incremental test repair approach called WATERFALL based on an iterative application of the WATER algorithm.

In 2015 Leotta et al. (2015) extended the ROBULA approach previously described by implementing an algorithm based on multi-locators, i.e. an algorithm that propose a set of different promising XPath queries by applying different heuristics, thus alternative queries can be automatically used if the selected query is not more usable (Leotta et al., 2015). In 2018, Stocco et al. (2018b,a) have proposed an approach and a tool to repair DOM-based locators by means of a visual-based approach that recognizes where is the Web element that should be pointed by the locator and propose new DOM-based locators for the new position of the Web element. More recently, Aldalur and Díaz (2017) and Aldalur et al. (2020) have proposed algorithms to regenerate broken locators on the basis of the analysis of alternative locators from the previous releases of the Web application under test.

Techniques for the preventive improvement of the testability. The third approach consists in carrying out preventive maintenance interventions on the code of the applications under test, in order to subsequently be able to generate robust locators. Few papers in literature proposed this type of solution. The most relevant are the ones based on the tool LED (Live Editor for DOM) of Bajaj et al. (2015b,a). This tool helps a Web developer to find out which are the possible locators (generated with the support of C&R tools such as Selenium) that can be associated with the elements of the Web application GUI while browsing. The Web Developer can add identifiers to the Web application source code in order to limit the use of less robust locators. This approach has the advantage of being applicable to a large variety of technologies for the development of Web applications, but leaves the burden of modifying the source code to the developer. This approach can be more difficult to apply in the case of Rich Internet Applications for which it is more difficult to trace the elements of the DOM to the Javascript code that generates them. With respect to this approach, our proposal presents a complete automation of the activities of modification of the source code of the application under test, applicable on most of the frameworks on which LED is applicable.

At the best of our knowledge, our approach is the first one completely supporting the automatic injection of unique identifiers in the source code of the Web applications that are able to support the generation of robust locators.

7. Conclusions

In this paper we presented an experiment where we systematically compared the fragility of seven different types of locator used by C&R testing techniques. To conduct the experiment, we defined a benchmark of Web page layout changes using a change classification model that we proposed.

In the study we focused on template-based Web applications, to whom the hook-based locator technique applies. We injected 192 different changes on two open source template-based Web applications based on Angular. We studied the robustness of test cases based on the seven considered types of locator, as the Web page layout changed according to our benchmark. The experiment confirmed the validity of our solution based on hook-based locators, which produced less breakages than all the considered techniques. It also allowed us to study how the fragility of different types of locators varied with different types of GUI layout changes.

This experiment also provided interesting data for exploring the cost-effectiveness of our hook-injection technique. With respect to other types of locators, the only additional step required by our technique consists in injecting hooks in the source code of the subject application before recording a test case. The injection needs to be repeated each time a new version of the app is developed. However, as the experiment showed, the impact of this step on an End-to-End regression testing process is irrelevant. Thanks to our automatic Hook-Injector software component, indeed, the injection step requires an execution time negligible with respect to the time required by the overall regression testing process. On the basis of these preliminary experimental data, we are confident about the acceptability and practicability of the approach in E2E regression testing processes. In future work we intend to extend our study and investigate its acceptability also in industrial CI/CD contexts.

A remark to be made is that the hook-based locator technique is only applicable to web applications implemented by a template technology, like Angular, whereas the other ones have a more general applicability.

In future work, we plan to extend our experimentation by considering further applications and other state-of-the-art locators, such as the ones proposed by ROBULA+ or other state-of-the-practice solutions. We also intend to mine real change-sets in open source web application repositories, in order to extend our change model and to support the definition of more realistic benchmarks of changes. In order to carry out further experiments, we also intend to develop a technique for the automatic injection of layout changes on the basis of the proposed change classification model.

Eventually, we plan to develop an optimized technique for locator generation that is able to predict the most promising locators on the basis of the experimental results of our studies.

CRediT authorship contribution statement

Marco De Luca: Software, Validation, Writing – review & editing. **Anna Rita Fasolino:** Conceptualization, Methodology, Investigation, Writing – original draft, Writing – review & editing. **Porfirio Tramontana:** Conceptualization, Methodology, Investigation, Writing – original draft, Writing – review & editing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

Data will be made available on request.

References

Aldalur, I., Díaz, O., 2017. Addressing web locator fragility: a case for browser extensions. In: Campos, J.C., Nunes, N., Campos, P., Calvary, G., Nichols, J., Martinie, C., Silva, J.L. (Eds.), Proceedings of the ACM SIGCHI Symposium on Engineering Interactive Computing Systems, EICS 2017, Lisbon, Portugal, June 26-29, 2017. ACM, pp. 45–50. http://dx.doi.org/10.1145/3102113.3102124.

Aldalur, I., Larrinaga, F., Perez, A., 2020. ABLA: an algorithm for repairing structure-based locators through attribute annotations. In: Huang, Z., Beek, W., Wang, H., Zhou, R., Zhang, Y. (Eds.), Web Information Systems Engineering - WISE 2020 - 21st International Conference, Amsterdam, the Netherlands, October 20-24, 2020, Proceedings, Part II. In: Lecture Notes in Computer Science, 12343, Springer, pp. 101–113. http://dx.doi.org/10.1007/978-3-030-62008-0_7.

Ardito, L., Coppola, R., Morisio, M., Torchiano, M., 2019. Espresso vs. EyeAutomate: An experiment for the comparison of two generations of android GUI testing. In: Proceedings of the Evaluation and Assessment on Software Engineering. EASE '19, Association for Computing Machinery, New York, NY, USA, pp. 13–22. http://dx.doi.org/10.1145/3319008.3319022.

Bajaj, K., Pattabiraman, K., Mesbah, A., 2015a. LED: tool for synthesizing web element locators. In: Cohen, M.B., Grunske, L., Whalen, M. (Eds.), 30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015, Lincoln, NE, USA, November 9-13, 2015. IEEE Computer Society, pp. 848–851. http://dx.doi.org/10.1109/ASE.2015.110.

Bajaj, K., Pattabiraman, K., Mesbah, A., 2015b. Synthesizing web element locators. In: Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering. ASE '15, IEEE Press, pp. 331–341. http://dx.doi.org/10.1109/ASE.2015.23.

Banerjee, I., Nguyen, B.N., Garousi, V., Memon, A.M., 2013. Graphical user interface (GUI) testing: Systematic mapping and repository. Inf. Softw. Technol. 55 (10), 1679–1694. http://dx.doi.org/10.1016/j.infsof.2013.03.004.

Chapman, P., Evans, D., 2011. Automated black-box detection of side-channel vulnerabilities in web applications. In: Proceedings of the 18th ACM Conference on Computer and Communications Security. CCS '11, Association for Computing Machinery, New York, NY, USA, pp. 263–274. http://dx.doi.org/10.1145/2046707.2046737.

Choudhary, S.R., Zhao, D., Versee, H., Orso, A., 2011. WATER: Web application test repair. In: Proceedings of the First International Workshop on End-To-End Test Script Engineering. ETSE '11, Association for Computing Machinery, New York, NY, USA, pp. 24–29. http://dx.doi.org/10.1145/2002931.2002935.

Coppola, R., Morisio, M., Torchiano, M., Ardito, L., 2019. Scripted GUI testing of android open-source apps: evolution of test code and fragility causes. Empir. Softw. Eng. http://dx.doi.org/10.1007/s10664-019-09722-9.

Di Lucca, G.A., Fasolino, A.R., Tramontana, P., 2007. Web pages classification using concept analysis. In: 23rd IEEE International Conference on Software Maintenance (ICSM 2007), October 2-5, 2007, Paris, France. IEEE Computer Society, pp. 385–394. http://dx.doi.org/10.1109/ICSM.2007.4362651.

Di Martino, S., Fasolino, A., Tramontana, P., Starace, L., 2020. Comparing the effectiveness of capture and replay against automatic input generation for android graphical user interface testing. Softw. Test. Verif. Reliab. 31, http://dx.doi.org/10.1002/stvr.1754.

Fasolino, A.R., Tramontana, P., 2022. Towards the generation of robust E2E test cases in template-based web applications. In: 2022 48th Euromicro Conference on Software Engineering and Advanced Applications (SEAA). pp. 104–111. http://dx.doi.org/10.1109/SEAA56994.2022.00024.

Hammontree, M.L., Hendrickson, J.J., Hensley, B.W., 1992. Integrated data capture and analysis tools for research and testing on graphical user interfaces. In: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems. CHI '92, Association for Computing Machinery, New York, NY, USA, pp. 431–432. http://dx.doi.org/10.1145/142750.142886.

Hammoudi, M., Rothermel, G., Stocco, A., 2016. WATERFALL: An incremental approach for repairing record-replay tests of web applications. In: Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering. In: FSE 2016, Association for Computing Machinery, New York, NY, USA, pp. 751–762. http://dx.doi.org/10.1145/2950290.2950294.

Hammoudi, M., Rothermel, G., Tonella, P., 2016. Why do record/replay tests of web applications break? In: 2016 IEEE International Conference on Software Testing, Verification and Validation (ICST). pp. 180–190. http://dx.doi.org/10.1109/ICST.2016.16.

Kaner, C., 2008. A tutorial in exploratory testing. URL https://www.kaner.com/pdfs/QAIExploring.pdf.

Kirinuki, H., Tanno, H., Natsukawa, K., 2019. COLOR: correct locator recommender for broken test scripts using various clues in web application. In: Wang, X., Lo, D., Shihab, E. (Eds.), 26th IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2019, Hangzhou, China, February 24-27, 2019. IEEE, pp. 310–320. http://dx.doi.org/10.1109/SANER.2019.8667976.

Kowalkiewicz, M., Orlowska, M.E., Kaczmarek, T., Abramowicz, W., 2006. Robust web content extraction. In: Carr, L., Roure, D.D., Iyengar, A., Goble, C.A., Dahlin, M. (Eds.), Proceedings of the 15th International Conference on World Wide Web, WWW 2006, Edinburgh, Scotland, UK, May 23-26, 2006. ACM, pp. 887–888. http://dx.doi.org/10.1145/1135777.1135928.

Leotta, M., Ricca, F., Tonella, P., 2021. Sidereal: Statistical adaptive generation of robust locators for web testing. Softw. Test. Verif. Reliab. 31 (3), http://dx.doi.org/10.1002/stvr.1767.

Leotta, M., Stocco, A., Ricca, F., Tonella, P., 2014. Reducing web test cases aging by means of robust xpath locators. In: 25th IEEE International Symposium on Software Reliability Engineering Workshops, ISSRE Workshops, Naples, Italy, November 3-6, 2014. IEEE Computer Society, pp. 449–454. http://dx.doi.org/10.1109/ISSREW.2014.17.

Leotta, M., Stocco, A., Ricca, F., Tonella, P., 2015. Using multi-locators to increase the robustness of web test cases. In: 2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST). pp. 1–10. http://dx.doi.org/10.1109/ICST.2015.7102611.

Leotta, M., Stocco, A., Ricca, F., Tonella, P., 2016. Robula+: An algorithm for generating robust xpath locators for web testing. J. Softw.: Evol. Process 28 (3), 177–204. http://dx.doi.org/10.1002/smr.1771.

Memon, A., 2008. Automatically repairing event sequence-based GUI test suites for regression testing. ACM Trans. Softw. Eng. Methodol. 18 (2), http://dx.doi.org/10.1145/1416563.1416564.

Montoto, P., Pan, A., Raposo, J., Bellas, F., López, J., 2011. Automated browsing in AJAX websites. Data Knowl. Eng. 70 (3), 269–283. http://dx.doi.org/10.1016/j.datak.2010.12.001.

Myllymaki, J., Jackson, J., 2002. Robust web data extraction with XML path expressions. IBM Res. Rep. URL https://dominoweb.draco.res.ibm.com/reports/RJ10245.pdf.

Nguyen, V., To, T., Diep, G., 2021. Generating and selecting resilient and maintainable locators for web automated testing. Softw. Test. Verif. Reliab. 31 (3), http://dx.doi.org/10.1002/stvr.1760.

Parr, T.J., 2004. Enforcing strict model-view separation in template engines. In: Proceedings of the 13th International Conference on World Wide Web. WWW '04, Association for Computing Machinery, New York, NY, USA, pp. 224–233. http://dx.doi.org/10.1145/988672.988703.

Pirzadeh, H., Shanian, S., 2014. Resilient user interface level tests. In: Crnkovic, I., Chechik, M., Grünbacher, P. (Eds.), ACM/IEEE International Conference on Automated Software Engineering, ASE '14, Vasteras, Sweden - September 15 - 19, 2014. ACM, pp. 683–688. http://dx.doi.org/10.1145/2642937.2642954.

Rafi, D.M., Moses, K.R.K., Petersen, K., Mäntylä, M.V., 2012. Benefits and limitations of automated software testing: Systematic literature review and practitioner survey. In: 2012 7th International Workshop on Automation of Software Test (AST). pp. 36–42. http://dx.doi.org/10.1109/IWAST.2012.6228988.

Ricca, F., Leotta, M., Stocco, A., 2019. Chapter three - three open problems in the context of E2E web testing and a vision: NEONATE. Adv. Comput. 113, 89–133. http://dx.doi.org/10.1016/bs.adcom.2018.10.005.

Stocco, A., Yandrapally, R., Mesbah, A., 2018a. Vista: web test repair using computer vision. In: Leavens, G.T., Garcia, A., Pasareanu, C.S. (Eds.), Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018. ACM, pp. 876–879. http://dx.doi.org/10.1145/3236024.3264592.

Stocco, A., Yandrapally, R., Mesbah, A., 2018b. Visual web test repair. In: Leavens, G.T., Garcia, A., Pasareanu, C.S. (Eds.), Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018. ACM, pp. 503–514. http://dx.doi.org/10.1145/3236024.3236063.

Thummalapenta, S., Devaki, P., Sinha, S., Chandra, S., Gnanasundaram, S., Nagaraj, D.D., Kumar, S., Kumar, S., 2013. Efficient and change-resilient test automation: An industrial case study. In: 2013 35th International Conference on Software Engineering (ICSE). pp. 1002–1011. http://dx.doi.org/10.1109/ICSE.2013.6606650.

Thummalapenta, S., Singhania, N., Devaki, P., Sinha, S., Chandra, S., Das, A.K., Mangipudi, S., 2012. Efficiently scripting change-resilient tests. In: Tracz, W., Robillard, M.P., Bultan, T. (Eds.), 20th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-20), SIGSOFT/FSE'12, Cary, NC, USA - November 11 - 16, 2012. ACM, p. 41. http://dx.doi.org/10.1145/2393596.2393643.

Wohlin, C., Runeson, P., Höst, M., Ohlsson, M.C., Regnell, B., 2012. Experimentation in software engineering. Springer, http://dx.doi.org/10.1007/978-3-642-29044-2.

Yandrapally, R., Thummalapenta, S., Sinha, S., Chandra, S., 2014. Robust test automation using contextual clues. In: Proceedings of the 2014 International Symposium on Software Testing and Analysis. In: ISSTA 2014, Association for Computing Machinery, New York, NY, USA, pp. 304–314. http://dx.doi.org/10.1145/2610384.2610390.

Yeh, T., Chang, T.-H., Miller, R.C., 2009. Sikuli: Using GUI screenshots for search and automation. In: Proceedings of the 22nd Annual ACM Symposium on User Interface Software and Technology. UIST '09, Association for Computing Machinery, New York, NY, USA, pp. 183–192. http://dx.doi.org/10.1145/1622176.1622213.

Marco De Luca is currently a Ph.D. Student at the Department of Electrical Engineering and Information Technology of the University of Napoli Federico II, Italy. His main interests regard software processes, software quality, testing techniques.

Anna Rita Fasolino is an Associate Professor at the Department of Electrical Engineering and Information Technology of the University of Napoli Federico II, Italy. She was previously an Assistant Professor at the University of Bari, Italy. She received her M.S. Laurea degree in Electronic Engineering and a Ph.D. in Electronic and Computer Engineering at the University of Naples Federico II.

Her research interests are in the area of software engineering with a focus on software testing, mobile app testing, reverse engineering, Web engineering, and embedded software engineering. In such fields she developed and participated in numerous R&D projects and co-authored more than 100 articles in peer-reviewed international journals, books, and proceedings of conferences and workshops. She has won distinguished paper awards at ICSM 2002 and ICSM 2012 for her work on Web testing and Automated Mobile app GUI testing. Anna Rita serves as a member of the program committee for several conferences in software engineering and is co-organizer of special issues and workshops related to testing of event-based software. She co-chaired the Doctoral Symposium at the 10th IEEE International Conference on Software Testing, Verification and Validation (ICST 2017). Anna Rita is an academic editor of the Journal of Systems and Software, of PeerJ Computer Science open access journal, and of the Computers Journal MDPI.

Porfirio Tramontana is an Associate Professor at the Department of Electrical Engineering and Information Technology of the University of Napoli Federico II, Italy. His main research interests fall mainly in the field of software engineering and include automation of reverse engineering, reuse, reengineering, migration, maintenance models, testing, quality assessment, semantic interoperability, in particular in the contexts of Web applications, Web services and mobile applications. He has served on numerous editorial committees of international conferences and journals, in the roles of reviewer, program committee member, program chair and associate editor.