IET Software

Research Article

IET Journals
The Institution of Engineering and Technology

Automated tests for cross-platform mobile apps in multiple configurations

ISSN 1751-8806
Received on 18th December 2018
Revised 27th June 2019
Accepted on 9th September 2019
E-First on 28th November 2019
doi: 10.1049/iet-sen.2018.5445
www.ietdl.org

Andre Augusto Menegassi[1,2], Andre Takeshi Endo[1] ✉

[1]Department of Computing, Federal University of Technology – Parana (UTFPR), Avenida Alberto Carazzai 1640, Cornelio Procopio, Brazil
[2]Universidade do Oeste Paulista (Unoeste), R. Jose Bongiovani 700, Presidente Prudente, Brazil
✉ *E-mail: andreendo@utfpr.edu.br*

Abstract: Cross-platform apps stand out by their ability to run in various operating systems (OSs), such as Android, iOS, and Windows. Such apps are developed using popular frameworks for cross-platform app development such as Apache Cordova, Xamarin, and React Native. However, the mechanisms to automate their tests are not cross-platform and do not support multiple configurations. Hence, different test scripts have to be coded for each platform, yet there is no guarantee they will work in different configurations varying, e.g. platform, OS version, and hardware available. This study proposes mechanisms to produce automated tests for cross-platform mobile apps. In order to set up the tests to execute in multiple configurations, the authors' approach adopts two reference devices: one running Android and other iOS. As both platforms have their own user interface (UI) XML representation, they also investigated six individual expression types and two combined strategies to locate UI elements. They have developed a prototype tool called cross-platform app test script recorder (x-PATeSCO) to support the proposed approach, as well as the eight locating strategies considered. They evaluated the approach with nine cross-platform mobile apps, comparing the locating strategies in six real devices.

1 Introduction

Currently, mobile devices permeate the daily life of most people and are available in various formats, mostly as smartphones, tablets, and wearables. They are equipped with powerful processors, large storage capacity, and several sensors [1]; modern operating systems (OSs) control the hardware of those devices. In a survey from International Data Corporation (IDC) [2] about the market share of mobile OSs, Android and Apple iOS were the most consumed platforms in the first quarter of 2017, with 85 and 14.7%, respectively. Other surveys of Gartner [3] brought the sales of smartphones in the first quarter of 2017; the Android platform (86.1%) was the market leader, followed by the Apple iOS platform (13.7%). Such OSs serve as a platform for executing a wide variety of software applications called mobile apps. The Statista site [4] offers statistics about the number of apps available for download in the main distribution stores: Android has the largest number of apps available to its users with 2.8 million apps and Apple iOS has 2.2 million apps.

The development of mobile apps can be classified into three groups: native apps, browser-based Web apps, and hybrid apps [5]. Native apps are developed using the mobile OS Software Development Kit (SDK), taking full advantage of the device functions as well as the OS itself. Web apps are developed with technologies to build software for the Web such as HTML5, CSS3, and Javascript [6]. They are stored on a Web server, run under a client browser, and do not have access to advanced features of the mobile OS. Finally, hybrid apps combine Web technologies such as HTML5, CSS3, and Javascript (using a native component known as WebView) with plugins that access the OS native application programming interfaces (APIs). Commercial and open source frameworks support the development of hybrid apps, such as Apache Cordova (https://cordova.apache.org), PhoneGap (http://phonegap.com), Sencha Touch (https://www.sencha.com), IONIC (http://ionicframework.com/), and Intel XDK (https://software.intel.com/xdk).

Unlike native apps, hybrid apps have the advantage of executing across multiple platforms. Cross-platform apps have gained momentum due to their ability to be built for different OSs, reducing the need for specific-platform code. Such an app has a common code base, though platform-specific builds are needed. [When mentioned that a cross-platform app is run, this study means that it is a proper platform-specific build of the app.] Other approaches to cross-platform development are frameworks such as React Native (https://facebook.github.io/react-native) and Xamarin (https://www.xamarin.com). They support the creation of apps with native user interface (UI) elements using programming languages such as Javascript and C#. The final product is a native and cross-platform app [7–9]; in this work, we adopt the term native cross-platform for this type of app.

Cross-platform app testing is challenging due to the variability of device settings and mobile OSs on the market [1, 10, 11]. As testing the app in a single device does not guarantee the correct operation in others [12, 13], each device represents a configuration (platform, hardware, screen size, sensors etc.) that needs to be verified. While automation is essential to cover many configurations, the current test mechanisms are not cross-platform. For instance, a UI test script using a tool such as Appium (http://appium.io/) has to be written twice since the UI's XML representations of Android and iOS are different. Such representations may also be different between versions of the same platform, such as in Android 4 and Android 6 [14]; this implies that two or more scripts might be needed for different versions of the same OS. The maintenance of UI test scripts has been known to be a costly task [15–17], and cross-platform apps aggravate it by requiring two or more versions of the same test script. Existing tools and research on automated testing have focused on specific platforms [18]; however, there is a lack of approaches for automated testing of cross-platform apps.

This study introduces an approach to generate scripts to test cross-platform mobile apps in multiple configurations. The approach relies on a reference device for each OS; currently, we have worked with the Android and iOS platforms. As it focuses on black-box testing at the system level, strategies to locate UI elements are also investigated. In particular, we analysed six individual expression types and two combined strategies. The approach and the locating strategies have been implemented in a prototype tool named cross-platform app test script recorder (**x-PATeSCO**). To evaluate the approach and compare the locating strategies, we conducted an experimental study with nine cross-

platform mobile apps tested on six real devices (multiple configurations).

In summary, this study has the following contributions.

- We describe the difficulties of implementing automated tests for cross-platform apps for different OSs such as Android and iOS (Section 2).
- We introduce an approach that supports the generation of scripts capable of testing cross-platform apps in several configurations. The approach is supported by the investigation of six individual XPath expressions and two combined strategies (Section 3).
- We present **x-PATeSCO**, an interactive tool that automates the use of the proposed approach (Section 4).
- We describe an extensive experimental evaluation of the proposed approach with nine open source and industrial apps. In the experiments, we evaluate the effectiveness and the performance of the proposed approach and associated locating strategies (Section 5).

This paper is structured as follows: Section 2 motivates the research problem. Section 3 introduces an approach to test cross-platform mobile apps. Section 4 describes the prototype tool and the experimental evaluation is shown in Section 5. Section 6 discusses on the obtained results. Section 7 brings the related work. Finally, Section 8 concludes the paper and sketches of future work.

2 Research problem

This section exemplifies the research problem with a hybrid app developed with Apache Cordova. Nevertheless, this problem is shared by all cross-platform apps, even the ones developed with native cross-platform frameworks such as Xamarin and React Native. UIs of a hybrid app are built using HTML elements, which are interpreted and transformed into an XML structure by the mobile platform. This structure differs between the platforms, as shown in Fig. 1.

Notice that HTML element `<a id='search'..>` of the **Fresh Food Finder** app (https://github.com/triceam/Fresh-Food-Finder) has two representations, one presented by Android and other by iOS. Such a structure might differ even within the same platform. Table 1 presents a brief mapping between HTML and XML native UI elements generated by the Android and iOS platforms.

XML nodes are composed of key attributes that contain descriptive information of UI elements; some of them are viewed by the app users. For the Android platform, such key attributes are 'content-desc' and 'text', while for iOS the attributes are 'label' and 'value'. Element identifier attributes are also available, namely 'resource-id' for Android and 'name' for iOS.

Element selectors are used to automate UI test cases of mobile apps. A test case consists of test input values, execution conditions, and expected results, designed to achieve a specific objective [19]. Selectors are 'patterns' or 'models', which provide mechanisms to locate elements (or nodes) in computational structures, such as XML or HTML [20]. After selecting a UI element, the tester can programme an action or verify a property. A well-known mechanism for selecting XML elements are query expressions in XPath, a query language for selecting elements (nodes) in computational structures which represent XML documents [21].

As shown in Fig. 1, the platform manufacturers do not follow a common standard to represent UI elements. Such differences reflect negatively on the test activities for cross-platform apps. The app is developed with features that enable its cross-platform execution, but the mechanisms to test it are not. Therefore, different test scripts are required to automate the testing of the same UI, each one with appropriate selectors. For example, the XPath selectors required to select the highlighted element in Fig. 1 (button 'Search for a market') suitable for Android 6.0.3 is '**//*/android.view.View[@content-desc='Search For a Market']**' and for iOS 9.3 is '**//*/UIAStaticText[@label='Search For a Market']**'. These XPath queries use the element type and key attributes that are

Fig. 1 *Android and iOS UI representations*

Table 1 HTML to XML native UI elements

HTML element type	Android element	iOS element
input button	android.widget.Button	UIAButton
input submit	android.widget.Button	UIAButton
Div	android.widget.View	UIAStaticText
Span	android.widget.View	UIAStaticText
Label	android.widget.View	UIAStaticText
Select	android.widget.Spinner	UIAElement
input text	android.widget.EditText	UIATextField
Textarea	android.widget.EditText	UIATextField
a (anchor)	android.widget.View	UIALink

platform-specific. Only the attribute values remained the same in both platforms.

At least, the tester has to identify and code two selectors to test the same UI element. Yet, there is no guarantee that the automated test will work for other configurations, with different OS version, hardware manufacturer, sensors, and so on. To improve the testing of cross-platform apps, we aim to propose an approach to record and generate a single test script capable of performing in different configurations.

3 Approach overview

Our approach defines a mechanism to automate the testing of cross-platform apps by constructing a test script to be run in multiple configurations. We based on the insight that using a reference device for each platform, a robust automated test can be produced by investigating and combining individual expressions. To do so, we propose an approach divided into three main steps, illustrated in Fig. 2. First, a reference device is chosen for each

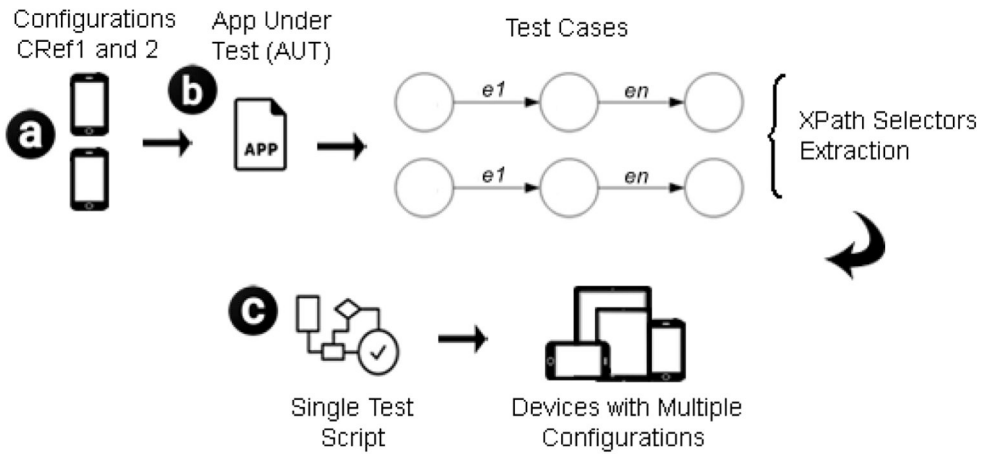

Fig. 2 *Approach overview*

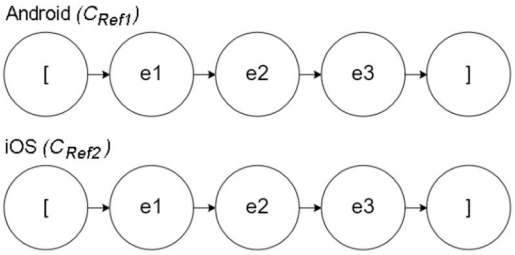

Fig. 3 *ESGs modelling the events under test*

platform; in this work, we have one running Android C_{Ref1}, and other running iOS C_{Ref2} (Fig. 2*a*, Section 3.1). Second, we define an event-driven model to represent the test cases and six individual expressions to locate UI elements were investigated (Fig. 2*b*, Section 3.2). Third, it proposes a single test script generation and defines two strategies that combine individual expressions in a more robust setting (Fig. 2*c*, Section 3.3). The three steps are detailed in the following subsections.

3.1 Device selection

This step consists of selecting a mobile device for each platform, namely one Android and other iOS. Testers might take several aspects into account to decide such as popularity, availability, final users, and so on. Some studies suggest choosing mobile devices for app testing based on their general popularity among end users [22, 23]. The selection of reference devices might also come from an existing demand; for instance, specific apps from a given business organisation run in a controlled set of devices.

While different criteria may be defined and applied to select the devices, any pair of reference devices can be used. The selected devices are then referred to as reference configurations: C_{Ref1} and C_{Ref2}. In addition, a cross-platform mobile app under test (AUT) should be installed on those devices.

3.2 UI element selection and test case definition

This step defines a model to express the test cases. In particular, we want to represent the expected event sequence of the user's interactions with the AUT. We adopted an event sequence graph (ESG) [24] to model the test case as a sequence of UI events (nodes) connected by edges; two ESGs are shown in Fig. 3. For each event, a UI element is selected and some action is executed (e.g. a click or an input text). This step is reproduced in the reference devices and two compatible ESGs are generated. The ESG compatibility is related to both models with the same number of events and data/action provided for each element. Fig. 3 shows a generic example of a test case; there is one ESG for each platform: Android and iOS, respectively. The ESG represents a test case and each node ($e1, \ldots, e3$) is an element with data needed for its test. For the example and all test cases used in the experiment (Section 5), the sequence of UI events was the same in both platforms.

During the UI element selection, the XML structure of app's UI is extracted. Based on this structure, each UI element has its type identified (textbox, button, anchor etc.), as well as its key and identified attributes with its respective values are stored. This data supports the construction of different XPath expressions to locate the element for the event under test. This study investigates six individual expressions: *AbsolutePath* and *IdentifyAttributes* represent the state-of-practice for UI selectors; *CrossPlatform* is a tentative to provider a unique selector for both platforms; *ElementType, AncestorIndex,* and *AncestorAttribute* are based on experience-based guidelines to have more generic and robust XPath expressions. The six expressions are described along with examples for the UI in Fig. 1, as follows.

AbsolutePath: It is a platform-specific expression, based on the absolute path from the root to the given element. In some cases, indexes are required to identify the element position within the XML structure. This expression has been employed in Web application testing to find elements in the document object model (DOM) structure [25]. It is a well-known alternative when the element has no identifier attribute. An example is shown as follows:

Android: `hierarchy/android.widget.FrameLayout /android.widget.LinearLayout/android.widget.FrameLayout /android.webkit.WebView/android.webkit.WebView /android.view.View/android.view.View/android.view.View[2] /android.view.View/android.view.View/android.view.View[3]`

iOS: `UIAApplication/UIAWindow/UIAScrollView /UIAWebView/UIALink[2]`

IdentifyAttributes: It is an expression based on the values of attributes that identify the element, such as *resource-id* for the Android platform and *name* for the iOS platform. Such expression is also well-known for Web applications [25, 26]. Attribute id is one of the primary strategies to locate HTML elements. An example is shown as follows:

Android: `//*[@resource-id='search']`

iOS: `//*[@name='Search For a Market']`

CrossPlatform: We propose it to define a single expression for different platforms. Such an expression is prepared to select a particular element from the app's UI independent of its execution platform. It combines key attributes (Android: *content-desc* or *text* and iOS: *label* or *value*) of elements and their values on both platforms, as discussed in Section 2. Rao and Pachunoori [27] suggest the use of expressions that combine attributes when the identifier attribute is not available. In our study, we combine the attributes of the two platforms in a single expression. An example is shown as follows:

Android and iOS: `//*[@content-desc='Search For a Market' or @label='Search For a Market']`

```
1: Input
   xPathCrossPlatform - Element XPath Cross-platform (CrossPlat)
   xPathsAndroid[] - Element XPaths for Android (AbsPath, IdAttr, ElemType and AncestInd, AncestAttr)
   xPathsiOS[] - Element XPaths for iOS (AbsPath, IdAttr, ElemType, AncestInd and AncestAttr)
   platform - Platform name under test
2: procedure ExecInOrder(xPathCrossPlatform, xPathsAndroid[], xPathsiOS[], platform)

3:     xPathSelectors[] = xPathCrossPlatform;

4:     if platform = "Android" then
5:         xPathSelectors[] += xPathsAndroid;
6:     else if platform = "iOS" then
7:         xPathSelectors[] += xPathsiOS;
8:     end if

9:     for each selector in xPathSelectors[] do
10:        e = FindElementByXPath(selector);
11:        if e != null then
12:            break;
13:        end if
14:    end for

15:    if e == null then
16:        throw new exception("Element not found");
17:    else
18:        e.action();
19:    end if
20: end procedure
```

Fig. 4 *Algorithm 1: ExpressionsInOrder algorithm*

Table 2 Expression types and weights

Expression type	Reliability	Weight
CrossPlatform	high	0.25
ElementType	high	0.25
IdentifyAttributes	medium	0.15
AncestorAttribute	medium	0.25
AbsolutePath	low	0.05
AbsoluteIndex	low	0.05

ElementType: It is an expression to find an element based on the combination of its type and key attributes (Android: *content-desc* or *text* and iOS: *label* or *value*) or platform-specific identifier attributes (Android: *resouce-id* or iOS: *name*). Rao and Pachunoori [27] also suggest combining element type in XPath expressions. We include the element type, prioritising the combination with the key attributes because they are more common than the identifier attributes. An example is shown as follows:

Android: `//*/android.view.View[@content-desc= 'Search For a Market']`

iOS: `//*/UIALink[@label='Search For a Market']`

AncestorIndex: It is a platform-specific expression based on the index of the desired element contained within its ancestor element. The index defines the exact position of the element, in this case, inside the ancestor container. This expression is hybrid since the container is located by a relative query expression (based on key and identifier attributes, when available) and the inner element is located by the index (absolute positioning). This expression might help to find the element when attribute-based expressions fail due to the dynamic changes; each run produces different attributes' valuation. An example is shown as follows:

Android: `//*[@resource-id='defaultView']/*[3]`

iOS: `//*[@label='Fresh-Food-Finder']/*[4]/*[1]/*[1]/*[1]`

AncestorAttribute: It is similar to the previous expression, but the index is replaced by a location based on the values of key attributes. An example is shown as follows:

Android:
`//*[@resource-id='defaultView']/*[@content-desc='Search For a Market']`

iOS: `//*[@label='Fresh-Food-Finder']/*[@label='Search For a Market']`

A type of expression is not always applicable on all platforms and its versions. Each platform can present a different UI XML structure and different attributes; this impacts on selecting elements to test. Besides the CrossPlatform expression, each expression has a platform-specific version. The single test engine, described in the next section, identifies the platform and selects the appropriate expression.

3.3 Single test engine

This step establishes a common mechanism for automated UI testing of cross-platform apps in multiple configurations. The test case represented by an ESG is the basis since each event contains the UI element's data and its query expressions to select it. As the aforementioned individual expressions might be limited in some contexts and only applicable in some cases, we introduce two locating strategies that combine the six expressions. The purpose of this combination is that one expression may compensate for the weaknesses of the other, providing an overall and more robust UI element selection strategy.

ExpressionsInOrder: Expressions are sorted by their type and executed sequentially. If the first expression fails, the next one is executed, and so on. The strategy aims to avoid the incomplete execution of a test case due to an *element not found* error. We compare this strategy with conventional individual expressions in Section 5. The order we defined prioritises relative expressions, starting with the CrossPlatform expression due to its suitability to select UI elements in Android and iOS. Absolute expressions have low priority since some studies indicate fragility in element localisation [15, 16, 27]. The sequence order set up is: CrossPlatform → ElementType → IdentifyAttributes → AncestorAttributes → AncestorIndex → AbsolutePath.

Algorithm 1 (see Fig. 4) presents pseudocode for the proposed strategy. *ExecInOrder* method (line 2) receives, as parameters, a set of six XPath expressions, and locates an element in the XML UI structure of app using the expressions (lines 9–14). In the end, the found element (by one of the expressions) is executed according to the action indicated by the tester (line 18). When the element is not found by any of the expressions, an exception is thrown to notify the test runner that the test case cannot proceed (lines 15 and 16). The general idea of the algorithm was defined in this work.

ExpressionsMultiLocator: In this strategy, all expressions are executed and the element is selected by voting criteria. These criteria are proposed based on reliability strategies to determine weights for each type of expression. This strategy was adapted from Leotta *et al.* [16], which employed it to Web application testing. As this strategy showed promising results for Web applications [16], we adapted it for cross-platform apps and compared with the other strategies in Section 5. Table 2 lists the weights for each expression. As follows, we discuss how the expressions' weights were defined:

- XPath expressions based on attribute values displayed in the UI have a high weight, due to the constancy in maintaining their values and independency of indexes and/or absolute paths. Examples of these attributes are *content-desc*, *text*, *label* and *value*. The expressions in this group are ElementType, CrossPlatform, and AncestorAttributes.

- XPath expressions based on identifier attribute values, such as *resource-id* (Android) and *name* (iOS) are the next ones. In some app development frameworks, the values of identifier attributes change in different executions, thus the selection reliability weakens. An example of this type of expression is IdentifyAttributes.
- XPath expressions based on absolute paths or indexes (positioning) have the smallest weights. They have low confidence due to their fragility during the software evolution [15, 16, 27]. Examples of these expressions are AbsolutePath and AncestorIndex.

Algorithm 2 (see Fig. 5) presents pseudocode for the ExpressionsMultiLocator strategy. As in the previous combined strategy, the *ExecMultiLocator* method (line 2) receives, as parameters, a set of six XPath expressions. For each element found, its weight is extracted according to the current expression type. An element returned by different expressions has the weights accumulated. In the end, the element with the highest voting (weight sum) is used in the test case execution (lines 24 and 25). When the element is not found by any of the expressions, an exception is thrown to notify the test runner that the test case cannot proceed (lines 21 and 22).

4 Tool implementation

The approach has been implemented in a prototype tool called cross-platform app test script recorder (**x-PATeSCO**). The tool is based on Appium, an open source framework to automate tests in native, Web or hybrid apps. In addition, Appium is a cross-platform and makes it possible to automate tests for iOS and Android platforms, using a Selenium WebDriver API.

x-PATeSCO architecture is illustrated in Fig. 6. The tool uses Selenium WebDriver API (http://www.seleniumhq.org/projects/webdriver/) and an Appium server to connect to two devices, one Android and other iOS (C_{Ref1} and C_{Ref2}), and send automation commands to the app's UI. The events specified by the tester are recorded and the tool automatically extracts and parses the UI's XML to build the test script using the appropriate XPath expressions (Section 3). The tool also generates a test project for Microsoft Visual Studio encoded in C# with support for the unit testing framework (https://msdn.microsoft.com/en-us/library/ms243147(vs.80).aspx). The project contains the classes that represent the automated test cases, modularised to help the testers in the test activities.

Fig. 7 shows a screenshot of x-**PATeSCO**; it offers functionalities to check the UI elements, the definition of its actions (click or text input), and test script generation. The first column provides fields to set up a remote connection with the Appium server and mobile devices. The second column shows a visual mechanism for elaborating the test case, providing pieces of information related to the app's UI. Column three brings the expressions that were automatically constructed by the tool based on the available data of the UI XML structure. Finally, column four highlights in an AUT UI screenshot, which is the UI element selected.

Fig. 8 illustrates an excerpt of a test script generated by the tool; in this case, the *ElementType* expression was selected by the tester. Lines 4–26 configure the connection with the Appium server, handling specific parameters for each platform. For each event, a method is created and invoked (lines 30–34); each method uses one of the eight strategies (proposed in Section 3) in accordance with the tester's will. Such methods contain the appropriate XPath expressions to select an element (lines 42–49); in this case, the ElementType strategy was adopted. Then, lines 53–55 execute the expression and fire the recorded action.

We expect that the tool will help developers and testers to implement more robust test scripts for cross-platform apps. The tool can fit in different testing processes. For instance, textual test cases for the cross-platform app may be provided and a tester/developer needs to automate them in scripts. In this scenario, **x-PATeSCO** can support the recording and generation of test scripts. Other case is that the produced scripts are compatible with Appium

1: **Input**
$xPathCrossPlatform$ - Element XPath Cross-platform (CrossPlat)
$xPathsAndroid[]$ - Element XPaths for Android (AbsPath, IdAttr, ElemType and AncestInd, AncestAttr)
$xPathsiOS[]$ - Element XPaths for iOS (AbsPath, IdAttr, ElemType, AncestInd and AncestAttr)
$platform$ - Platform name under test
2: **procedure** EXECMULTILOCATOR($xPathSelectorsCrossPlatform[]$, $xPathSelectorsAndroid[]$, $xPathSelectorsiOS[]$, $platform$)
3: $xPathSelectors[]$ = $xPathSelectorsCrossPlatform$;
4: **if** $platform$ = "Android" **then**
5: $xPathSelectors[]$ += $xPathSelectorsAndroid$;
6: **else if** $platform$ = "iOS" **then**
7: $xPathSelectors[]$ += $xPathSelectorsiOS$;
8: **end if**
9: $elements[]$;
10: $voting[]$;
11: **for each** $selector$ in $xPathSelectors[]$ **do**
12: e = FindElementByXPath($selector$);
13: **if** e != null **then**
14: **if** !elements.Contains(e) **then**
15: elements.Add(e);
16: voting[e] = 0;
17: **end if**
18: $voting[e]$ = $voting[e]$ + ExtractWeight(selector);
19: **end if**
20: **end for**
21: **if** $elements.length$ == null **then**
22: throw new Exception("Element not found");
23: **else**
24: e = MaxElement($voting$);
25: $e.action()$;
26: **end if**
27: **end procedure**

Fig. 5 *Algorithm 2: ExpressionsMultiLocator algorithm*

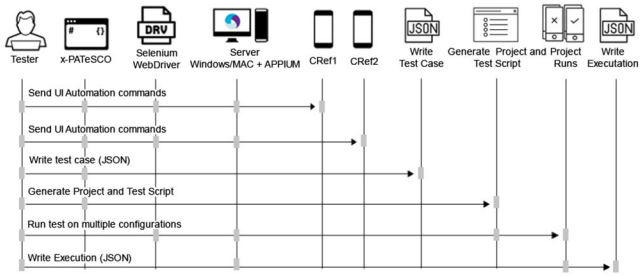

Fig. 6 *Tool architecture*

and might be used in cloud test environments, such as Amazon Device Farm (https://aws.amazon.com/en/device-farm), Bitbar (http://www.bitbar.com/testing), and TestObject (https://testobject.com). These services offer a large number of real devices that can be connected and used to test cross-platform mobile apps. Finally, the tool may be used by testers with low experience on cross-platform app testing. Tests are all recorded by means of its UI (see Fig. 7), so **x-PATeSCO** has a mild learning curve.

The **x-PATeSCO** tool is available as an open source project in [28]. The overall application of the tool and its locating strategies are analysed in the next section.

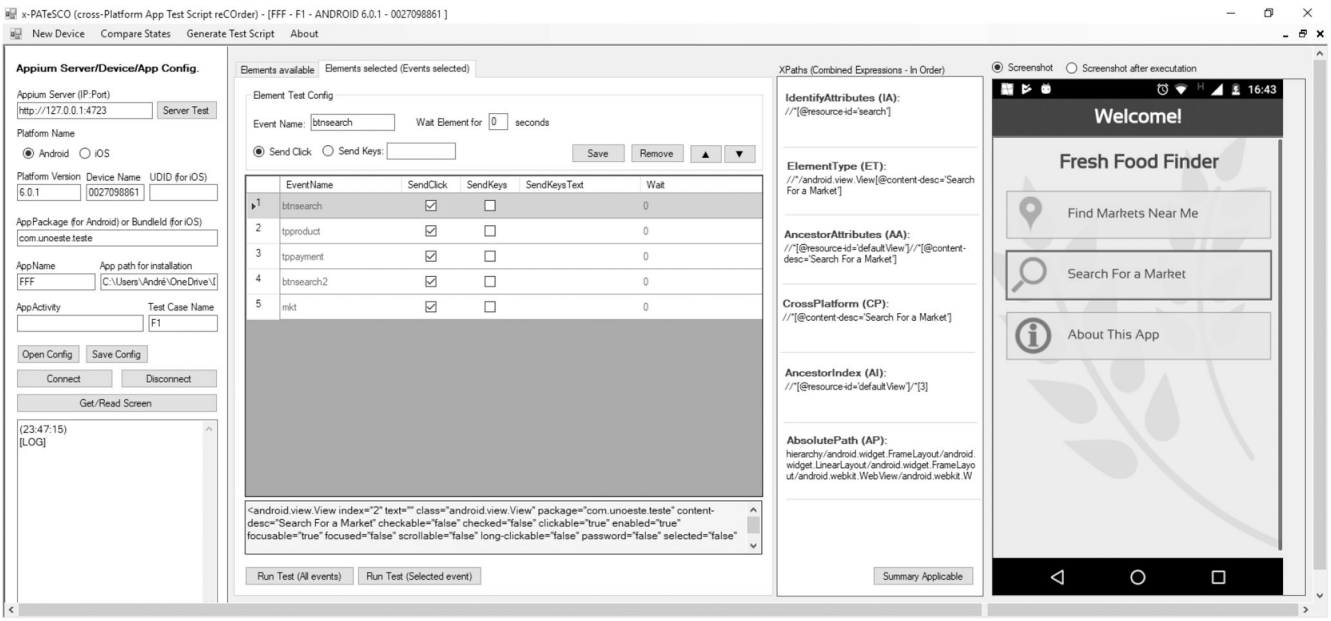

Fig. 7 *X-PATeSCO tool*

```
1   [TestMethod]
2   public void TestMethodMain()
3   {
4           /*APPIUM config*/
5           _capabilities.SetCapability("platformName",
                                        ProjectConfig.PlataformName);
6           _capabilities.SetCapability("platformVersion",
                                       ProjectConfig.PlatformVersion);
7           _capabilities.SetCapability("deviceName", ProjectConfig.DeviceName);
8           _capabilities.SetCapability("appPackage", ProjectConfig.AppPackage);
9           _capabilities.SetCapability("newCommandTimeout", "3000");
10          _capabilities.SetCapability("sessionOverride", "true");
11          _capabilities.SetCapability("app", ProjectConfig.AppPath);
12
13          Uri defaultUri = new Uri(ProjectConfig.AppiumServer);
14
15          if (ProjectConfig.PlataformName == "Android")
16          {
17               _capabilities.SetCapability("appActivity",
                                          ProjectConfig.AppActivity);
18              _driver = new AndroidDriver<IWebElement>(defaultUri, _capabilities,
                                             TimeSpan.FromSeconds(3000));
19          }
20          else if (ProjectConfig.PlataformName == "iOS")
21          {
22              _capabilities.SetCapability("automationName", "XCUITest");
23              _capabilities.SetCapability("bundleId", ProjectConfig.AppPackage);
24              _capabilities.SetCapability("udid", ProjectConfig.Uuid);
25              _driver = new IOSDriver<IWebElement>(defaultUri, _capabilities,
                                           TimeSpan.FromSeconds(3000));
26          }
27
28          /*initialing test*/
29
30          btnsearchSendClick_Test(); //btnsearch
31          tpproductSendClick_Test(); //tpproduct
32          tppaymentSendClick_Test(); //tppayment
33          btnsearch2SendClick_Test(); //btnsearch2
34          mktSendClick_Test(); //mkt
35  }
36
37  public void btnsearchSendClick_Test()
38  {
39        ForceUpdateScreen();
40        string[] selectors = new string[0];
41
42        if (ProjectConfig.PlataformName == "Android")
43        {
44            selectors =
                new string[]
                 {@"//*/android.view.View[@content-desc='Search For a Market']"};
45        }
46       else if (ProjectConfig.PlataformName == "iOS")
47        {
48            selectors = new string[]
                 {@"//*/UIAStaticText[@label='Search For a Market']"};
49        }
50
51        string[] selectorsType = new string[] {@"ElementType"};
52
53        IWebElement e = _locator.FindElementByXPath(selectors[0],
                                                    selectorsType[0]);
54
55        e.Click();
56
57        /*Insert your assert here*/
58  }
59  ...
```

Fig. 8 *Test script generated by* `x-PATeSCO`

5 Evaluation

To evaluate the proposed approach, it is necessary to understand the performance of the locating strategies, to analyse their behaviour with different apps, and to verify the results obtained from different configurations. Therefore, we conducted an experimental evaluation to compare the eight locating strategies: six individual expressions (Section 3.2) and two combined strategies (Section 3.3). The following research questions (RQs) have been investigated:

- *RQ1:* How effective are the locating strategies to test cross-platform mobile apps in multiple configurations?
- *RQ2:* How do the locating strategies perform with respect to execution time?

RQ1 aims to compare the effectiveness of the locating strategies through the analysis of how applicable are such expressions and if tests based on them can be executed successfully in different configurations. First, we measure their applicability to observe in how many events each strategy can be used to select UI elements. The executability was taken into account to see how successful (to select an element at runtime) a strategy might be. Then, we analyse executability at event and test case levels. As for *RQ2*, we aim to analyse how the locating strategies might influence the execution time of test cases. Then, we measure the CPU time in seconds of events successfully executed using a given locating strategy.

5.1 Experimental objects and procedure

To answer the RQs we proceeded as follows. We selected a set of cross-platform apps, two industrial apps, and seven samples apps to run on Android and iOS platforms. Partner IT companies provided the industrial apps projects containing the necessary assets to build them in Android and iOS. The sample apps were from books on developing hybrid and native cross-platform apps, as well as GitHub repositories.

The apps used in the evaluation are characterised in Table 3, showing the type of app, number of lines of code (LOC), main programming language, cross-platform app type, and development framework. Notice that we adopted a varied set of open source and industrial cross-platform apps, with different sizes (LOC), implemented with a diverse set of programming languages and frameworks.

The projects of these apps were compiled for the Android and iOS platforms and all the tests were run in the devices listed in Table 4. The iOS projects pose restrictions to perform automated tests, namely, the app needs to be signed with a developer account provided by Apple and registered mobile devices are required. This fact limited our experiments only for apps that we had the source code available. Table 4 shows pieces of information about the six actual devices (three with Android and three with iOS) in which

Table 3 Apps under test

App name	Type	LOC	Programming language	Cross-platform app type	Development framework
Fresh Food Finder	open source	13,824	Javascript	hybrid	Cordova
Order App	industrial	71,565	Javascript	hybrid	Cordova
MemesPlay	open source	5484	Javascript	hybrid	Cordova
Agenda	open source	1038	Javascript	hybrid	Cordova
TodoListCordova	open source	9304	Javascript	hybrid	Cordova
MovieApp	open source	2088	Javascript	native cross-platform	ReactNative
TodoList	open source	405	Javascript	native cross-platform	ReactNative
Tasky	open source	654	C#	native cross-platform	Xamarin
Bargains	industrial	178,266	C#	native cross-platform	Xamarin

Table 4 Configurations evaluated

Device	OS	CPU	RAM
Samsung Tab E 7″	Android 4.4.4	4-Core 1.3 GHz	1 GB
Motorola G4 5.5″	Android 6.0.1	8-Core 1.4 GHz	2 GB
Motorola G1 5″	Android 5.1	4-Core 1.2 GHz	1 GB
iPhone 4 3.5″	iOS 7.1.2	2-Core 1 GHz	512 MB
iPad 2 9.7″	iOS 9.3	2-Core 1 GHz	512 MB
iPad 4 9.7″	iOS 10.2	2-Core 1.4 GHz	1 GB

Table 5 Test data

App name	#TCs	#Ev.	TP LOC	Avg LOC/TC
Fresh Food Finder	3	20	5600	233
Order App	3	16	5134	214
MemesPlay	3	12	3849	160
Agenda	3	18	5432	226
TodoListCordova	3	10	3649	152
MovieApp	3	10	3435	143
Todolist	3	13	4136	172
Tasky	3	9	3153	131
Bargains	2	10	3248	203

the apps were tested. The data on the devices' processing power provides a means to interpret the execution time analysed in the results. In addition, this data can also be used to compare with future replications of this study. The devices with Android 6.0.1 and iOS 9.3 were selected as the reference configurations. We based this decision on the popularity of the platform versions [29, 30].

We employed the **x-PATeSCO** tool to support the experimental evaluation. It connects to Appium which, in turn, connects to the mobile devices in order to execute the test scripts and extract data. Specific versions of Appium (according to the mobile OS) have been installed and configured in two test servers for network access by local computers. The resources used in the experiments are the following:

- XCode 7.3 (for iOS 7);
- XCode 8.2 (for iOS 9.3 and 10.2);
- Android SDK r24.4.1;
- VisualStudio 2017;
- Appium 1.4.3 (Windows OS) and 1.5/1.6.3 (MAC OS);
- Selenium WebDriver for C# 2.53.0;
- Appium WebDriver 1.5.0.1;
- Windows 10 OS;
- MAC OS X Sierra.

The nine apps were divided among four independent participants with knowledge in software testing and mobile computing. We asked them to explore the apps and design two or three test cases for each one. Each test case, recorded in **x-PATeSCO**, has an event sequence representing common user interactions. The test projects (generated by the tool) implement the eight strategies and collect the metrics used in this study.

5.2 Analysis of results and findings

To answer the RQs proposed, this section presents and discusses the results obtained. Table 5 summarises pieces of information about the size of the tests generated by our tool and executed for each application; it shows the number of test cases (#TCs), how many events were tested (#Ev.), LOC of the test projects generated by the **x-PATeSCO** tool (TP LOC), and average LOC per test case (Avg LOC/TC). The test projects were then run in each of the six configurations listed in Table 4, and the results were collected.

RQ1 – effectiveness: First, we observe the applicability of the locating strategies. We define applicability ratio as the number of events in which a given strategy can be used to select a UI element divided by the total number of events. As discussed in Section 3, some strategies might not be used in some contexts, e.g. the UI element has not some key attribute. As this information is useful for the tester when the test case is being coded, we collected the applicability ratio for each locating strategy.

Fig. 9 illustrates the applicability ratio when recording the test cases in C_{Refs}. Individual expressions based on absolute paths (`AbsolutePath` and `AncestorIndex`) obtained 100% of applicability in all apps. This was expected once absolute expressions map an exact path in the XML structure and do not depend on often-optional parts (e.g. attributes). For other individual strategies based on the attributes (namely, `CrossPlatform`, `AncestorAttributes`, and `IdentifyAttributes`), their applicability ratio was inferior. Among them, the `IdentifyAttributes` strategy presented the lowest applicability ratio, being useful in 60.1% of the events. As the combined expressions use the six individual expressions, `ExpressionInOrder` and `ExpressionMultiLocator` have an applicability ratio of 100%.

While the applicability shows when a given strategy can be used, it is important to analyse how such a strategy is successful to select a UI element at runtime. We define the executability ratio as the strategy capability to select successfully a UI element when an event is tested. After running the scripts in the six configurations, the executability ratio of the strategies was calculated.

First, we look into the executability ratio for each event performed during the tests. Table 6 summarises the event executability ratio showing the results for each configuration (first row). For each configuration, it shows the number of successfully executed events (SEE), the event executability ratio with respect to the number of applicable events (%Ex.), and the event executability ratio concerning the total number of events (%To.). The last column (ALL) summarises all configurations, while the last row shows, per configuration, the average event executability ratio.

For each configuration, 118 events were executed, except for configuration IOS7-1-2 with 85 events. The number of applicable events was smaller since three apps (MovieApp, Bargains, and TodoList) were not compatible with this iOS version. Configurations with older versions than C_{Refs} (namely, Android4-4 and IOS7-1-2) present the lowest ratio. This is particularly evinced by absolute path strategies (`AbsolutePath` and `AncestorIndex`). We noticed that the UI's XML structures in these configurations are very different from the references.

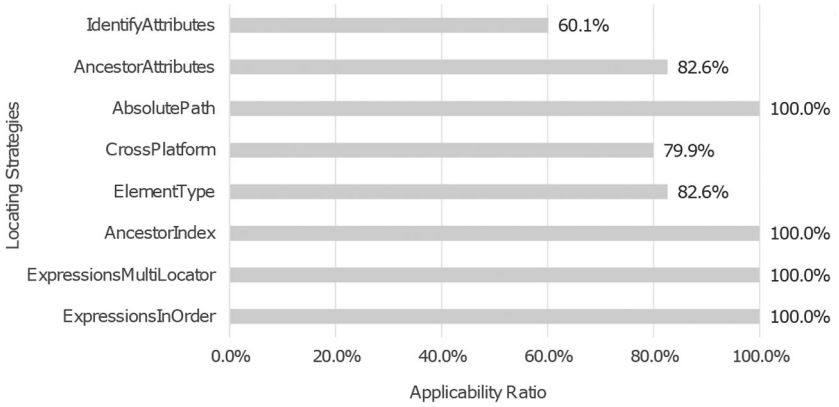

Fig. 9 *Applicability ratio in C_{Refs}*

Table 6 Event executability ratio

Locating strategy		And4-4	And5-1	And6-0-1^{Ref}	IOS7-1-2	IOS9-3^{Ref}	IOS10-2	ALL
`AbsolutePath`	SEE[a]	0	76	117	14	118	30	59.2
	%Ex.[b]	0.0	64.4	99.2	16.5	100.0	25.4	52.6
	%To.[c]	0.0	64.4	99.2	16.5	100.0	25.4	52.6
`AncestorIndex`	SEE	31	108	117	6	106	35	67.2
	%Ex.	26.3	91.5	99.2	7.1	89.8	29.7	59.7
	%To.	26.3	91.5	99.2	7.1	89.8	29.7	59.7
`ElementType`	SEE	29	89	95	14	93	64	64.0
	%Ex.	29.3	89.9	96.0	19.2	92.1	63.4	67.1
	%To.	24.6	75.4	80.5	16.5	78.8	54.2	56.9
`CrossPlatform`	SEE	16	64	80	36	95	73	60.7
	%Ex.	18.0	71.9	89.9	42.4	97.9	75.3	66.7
	%To.	13.6	54.2	67.8	42.4	80.5	61.9	53.9
`AncestorAttributes`	SEE	17	81	86	18	97	53	58.7
	%Ex.	17.2	81.8	86.9	24.7	96.0	52.5	61.5
	%To.	14.4	68.6	72.9	21.2	82.2	44.9	52.1
`IdentifyAttributes`	SEE	12	47	47	29	80	45	42.8
	%Ex.	19.0	74.6	74.6	46.0	93.0	52.3	61.3
	%To.	10.2	39.8	39.8	34.1	67.8	38.1	38.5
`ExpressionsInOrder`	SEE	29	99	107	39	118	82	79.0
	%Ex.	24.6	83.9	90.7	45.9	100.0	69.5	70.2
	%To.	24.6	83.9	90.7	45.9	100.0	69.5	70.2
`ExpressionsMultiLocator`	SEE	29	99	107	39	118	82	79.0
	%Ex.	24.6	83.9	90.7	45.9	100.0	69.5	70.2
	%To.	24.6	83.9	90.7	45.9	100.0	69.5	70.2
`%Ex. per configuration`		19.9	80.2	90.9	30.9	96.1	54.7	—

[a] SEE stands for a number of successful events executed.

[b] %Ex. stands for the event executability ratio with respect to the number of applicable events.

[c] %To. stands for the event executability ratio with respect to the total number of events.

Ref stands for the reference configurations.

The specific characteristics of the two apps affected the results. A ratio of 99.2% for `AbsolutePath` in Android6-0-1, and not 100% as in IOS9-3, was observed. As Android6-0-1 is one of C_{Refs} and `AbsolutePath` is always applicable, this result was unexpected. The issue happened in the MovieApp app: the content of a certain screen is dynamic, changing the reference XML structure from which the test case was recorded. This broke the expression and no UI element was selected. Such a case also affected the executability of the combined strategies. Also, in Android6-0-1, the ratio of 90.7% (below 100%) was observed for the combined strategies. Besides the issue with MovieApp, the ToDoList app presented a unique behaviour. One of its native elements (a menu bar) has different structures for each platform. In Android, the values of the menu's attributes collided with the values of other elements within the same UI. This affects some expressions as well as the combined strategies. In iOS, this collision did not occur and the combined strategies have a ratio of 100%.

As for individual expressions in all configurations, `ElementType` had the best results with 67.1% of executability ratio (for applicable events – %Ex.), followed by `CrossPlatform` (66.7%), `AncestorAttributes` (61.5%), `IdentifyAttributes` (61.3%), and `AncestorIndex` (59.7%). `AbsolutePath` had the worst event executability with only 52.6%. By comparing with the overall number of events (%To.), `AncestorIndex` was the best (59.7%) since it is always applicable and `IdentifyAttributes` was the worst (38.5%) due to its lowest applicability ratio. Concerning the combined strategies, `ExpressionsInOrder` and `ExpressionsMultiLocator` were better than individual expressions (both with 70.2%). Their improvement is from 3.1% (w.r.t. `ElementType`) to 17.6% (w.r.t. `AbsolutePath`). By analysing the comparison with the overall number of events (%To.), their executability is even better with improvements from

10.5% (w.r.t. AncestorIndex) to 31.7% (w.r.t. IdentifyAttributes).

We now look into the executability ratio from the test case perspective. This perspective is taken into account because practitioners may want to know whether the test case has been run completely or not using a specific strategy. A test case was considered applicable when all its events can be executed using a given strategy. Table 7 summarises the test case executability ratio; it shows the results for each configuration (first row). It is pretty similar to Table 6, the only modified row is the first for each locating strategy, namely successful test cases (STC). It measures how many test cases were executed with success until the last event. For each configuration, 26 test cases are available, except for configuration IOS7-1-2 with 18 test cases.

As for individual expressions in all configurations, IdentifyAttributes had the best results with 75% of executability ratio (for applicable test cases – %Ex.), followed by ElementType (58.6%), CrossPlatform (58.1%), AncestorIndex (54.1%), and AncestorAttributes (51%). AbsolutePath had the worst test case executability with only 49.3%. By comparing with the overall number of test cases (%To.), AncestorIndex was the best (54.1%) since it is always applicable and IdentifyAttributes was the worst (14.2%) due to its lowest applicability ratio. Concerning the combined strategies, ExpressionsInOrder and ExpressionsMultiLocator (both with 65.5%) were behind IdentifyAttributes (with 75%). However, they outperformed other individual expressions from 6.9% (w.r.t. ElementType) to 16.2% (w.r.t. AbsolutePath). By the comparison with the total number of test cases (%To.), their test case executability ratio is the best with improvements from 11.4% (w.r.t. AncestorIndex) to 51.3% (w.r.t. IdentifyAttributes).

> In general, the executability ratio of attribute-based expressions is better than the absolute path-based. Nevertheless, absolute path-based strategies were always applicable. As for event executability, the combined strategies were better than the individual expressions. From the test case perspective, ExpressionsInOrder and ExpressionsMultiLocator were only behind IdentifyAttributes.

RQ2 – execution time: Besides the effectiveness, we shed light on the cost of each locating strategy by analysing the CPU execution time. To do so, we did not take into account events in which the strategy failed to locate the UI element. Fig. 10 illustrates the average CPU time (in seconds) for SEE.

Concerning individual expressions, IdentifyAttributes was the fastest (13.2 s), followed by CrossPlatform (14.9 s), ElementType (16.4 s), and AncestorAttributes (16.4 s). Absolute path-based strategies (AncestorIndex and AbsolutePath) were the slowest (24.4 and 25.1 s, respectively). As for combined strategies, ExpressionsInOrder and ExpressionsMultiLocator were slower than individual expressions with 26.9 and 166.7 s, respectively. While ExpressionsInOrder is close to the slowest individual expressions (around 24–25 s), the execution time of ExpressionsMultiLocator is approximately six times slower. This was expected since its implementation requires the execution of the six individual expressions plus the voting calculation.

> Attribute-based expressions are the fastest, followed by the absolute path-based. The combined strategies have the worst execution times: ExpressionsInOrder is close to the slowest individual expressions and around six times faster than ExpressionsMultiLocator.

Table 7 Test case executability ratio

Locating strategy		And4-4	And5-1	And6-0-1[Ref]	IOS7-1-2	IOS9-3[Ref]	IOS10-2	ALL
AbsolutePath	STC[a]	0	15	25	3	26	4	12.2
	%Ex.[b]	0.0	57.7	96.2	16.7	100.0	15.4	49.3
	%To.[c]	0.0	57.7	96.2	16.7	100.0	15.4	49.3
AncestorIndex	STC	7	23	25	0	20	5	13.3
	%Ex.	26.9	88.5	96.2	0.0	76.9	19.2	54.1
	%To.	26.9	88.5	96.2	0.0	76.9	19.2	54.1
ElementType	STC	7	13	13	3	13	9	9.7
	%Ex.	41.2	76.5	76.5	16.7	86.7	60.0	58.6
	%To.	26.9	50.0	50.0	16.7	50.0	34.6	39.2
CrossPlatform	STC	3	7	7	5	12	9	7.2
	%Ex.	30.0	70.0	70.0	41.7	75.0	56.3	58.1
	%To.	11.5	26.9	26.9	27.8	46.2	34.6	29.1
AncestorAttributes	STC	4	11	11	4	14	6	8.3
	%Ex.	23.5	64.7	64.7	30.8	82.4	35.3	51.0
	%To.	15.4	42.3	42.3	22.2	53.8	23.1	33.8
IdentifyAttributes	STC	3	4	4	3	4	3	3.5
	%Ex.	75.0	100.0	100.0	75.0	66.7	50.0	75.0
	%To.	11.5	15.4	15.4	16.7	15.4	11.5	14.2
ExpressionsInOrder	STC	7	21	22	6	26	15	16.2
	%Ex.	26.9	80.8	84.6	33.3	100.0	57.7	65.5
	%To.	26.9	80.8	84.6	33.3	100.0	57.7	65.5
ExpressionsMultiLocator	STC	7	21	22	6	26	15	16.2
	%Ex.	26.9	80.8	84.6	33.3	100.0	57.7	65.5
	%To.	26.9	80.8	84.6	33.3	100.0	57.7	65.5
#TC per configuration		26	26	26	18	26	26	—

[a] STC stands for a number of successful test cases executed.

[b] %Ex. stands for the test cases executability ratio with respect to the number of applicable test cases.

[c] %To. stands for the test case executability ratio with respect to the total number of test cases.

[Ref] stands for the reference configurations.

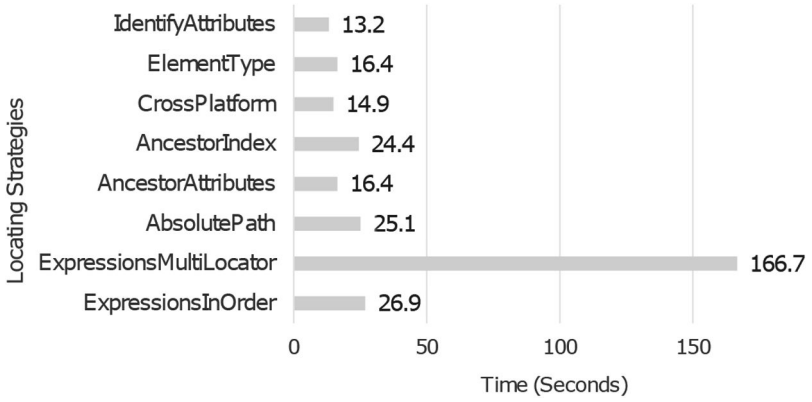

Fig. 10 *Average execution time of successful events*

6 Discussion

This section discusses threats, limitations, and observations, as well as raises directions for future research.

Threats to validity: We herein analyse the threats to validity. First, we adopted a limited number of configurations (devices). There exist a huge amount of devices in the market and each device may have several settings (e.g. OS version, language, and connected accessories). As the cost of a comprehensive investigation would be prohibitive, we opted by varying the OS versions, mixing smartphones and tablets with diverse configurations. Yet, we recognise that only a small portion of possible configurations has been covered, reducing the generalisation of obtained results.

Another threat is the selection of the reference configurations. As the locating strategies are designed based on them, other reference devices might produce different results. We believe that such selection would be usually built on top of popularity; thus, we based our decision on known statistics on users' usage of platform versions.

As for the apps, we needed to access the source code as a restriction of the iOS platform; this limited our options. Moreover, app developers may employ one of the several cross-platform development frameworks available in the market. So, it is difficult to characterise a meaningful sample of cross-platform apps. To reduce this threat, we selected not only open source projects but also two industrial projects. Besides, we assured that apps developed with main development frameworks (namely, React Native, Xamarin, and Apache Cordova) were selected.

To reduce the bias on the tests' types and structure, four independent participants with experience in mobile computing designed the test cases. While the use of independent participants helps to avoid a biased selection of test cases, one might argue on their quality and representativeness. Using our previous experience in software testing and analysing open source mobile UI tests [31], these test cases were inspected. We conclude they have well-defined test goals and represent common UI tests for mobile apps.

Most parts of the approach and data collection steps were executed automatically. As a consequence, some data could be misleading since our implementation is subjected to faults. To mitigate this threat, we manually tested the locating strategies in **x-PATeSCO**.

Together with the diversity of apps' domain, size, and development framework, we aimed to cover the main characteristics that influence the automated testing of cross-platform apps. Furthermore, we adopted a well-defined set of auditable quantitative metrics, described the sample selection, as well as other methodological steps. We then have done our best to employ a sound method of evaluation, though most of the listed threats might be overcome in the future with replications. Thus, we made the tool and the experiment objects as an open source experimental package available in [28].

Limitations and observations: **x-PATeSCO** only implements a subset of all events (e.g. click and fill input field) that can be performed in mobile apps, so it is not possible to fully evaluate its expressiveness. Nevertheless, the core of our approach is how to select the UI element the tester wants to verify or interact it with. Then, any of the existing events may be applied to the selected UI element. As the tool is built on top of a well-known UI testing framework, the comprehensive set of events supported by Appium (http://appium.io/docs/en/about-appium/intro/) may be incorporated in future versions.

In the evaluation, we observed that the test cases were equal (w.r.t. a sequence of UI events) in Android and iOS. While not surprising since cross-platform apps have a unique code base, there would be cases where UI components are different and event sequences may be shorter/longer in a certain platform. A future direction could be the identification of such cases in practice and assessing how our approach would fare.

Other perceived limitation is that the test scripts generated by **x-PATeSCO** are not sound if we consider the ample universe of available devices. In other words, the script will fail to select UI elements in some specific devices. For these cases, the tester will need to repair the expression. Our goal was to minimise the chances of a broken test script, as the approach improved the executability ratio. We anticipate that with a well-defined and restrict set of devices, the approach and its tool could be tuned to produce better results. In informal tests with some partner IT companies, we have received positive feedbacks and future versions of the tool could be used in practice. However, we recognise that further formal experiments in real-world settings are needed.

The OS versions of Android and iOS used in this research, as well as other versions currently available, share similar XML structures for the apps' UIs. So, we believe that the approach and its supporting tool are applicable in most of the current cross-platform apps; this is corroborated by the results in Section 5. However, there is no guarantee that the mobile OS vendor will maintain compatibility between its versions. During the experiments, Android 4 showed meaningful differences with respect to the reference version (Android 6). In addition, the latest version of iOS uses a new XML structure for organising UI elements. We anticipate that general models of the XML structure could be defined in future, as well as mappings between the versions and OSs. Such artefacts could be integrated into our approach to making it more general.

Since the main purpose of this study is to produce robust test scripts for cross-platform apps, we focused our analysis on this aspect by measuring the effectiveness (with respect to executability ratio) and performance (execution time). Therefore, fault detection is out of the scope of this study. In general, cross-platform apps pose the same challenges for bug detection as native apps [18], being more error-prone to compatibility bugs. A future direction is to investigate if the test scripts produced by **x-PATeSCO** can be amplified to detect compatibility bugs.

While this study investigates six different types of individual XPath expressions, there exist several others in the literature, mainly for Web applications [16, 27, 32]. Practitioners and researchers can even come up with new kinds of expression and further improvements may be achieved in future work. As for combined strategies, we explored two approaches `ExpressionsInOrder` and `ExpressionsMultiLocator`. We

envision that two well-known techniques are promising in this context: evolutionary algorithms and machine learning. On the one hand, a search-based approach may support the construction of more robust expressions by optimising the executability ratio of test scripts. On the other hand, the design of robust and general expressions may be 'learned' from a well-constructed set of examples.

There are several UI testing frameworks for mobile apps; notable examples are Espresso and Robotium for native Android apps, XCTest for iOS, and Appium for cross-platform apps. With such frameworks, practitioners code and maintain scripts that automate the execution of UI tests; this scenario seems to be the current state-of-the-practice in automated UI testing of mobile apps [31, 33]. We surmise that our approach would require a similar effort as coding native UI tests. `x-PATeSCO` automates most of the script generation, so the tester will spend less time coding. On the other hand, native tests are less-prone to have issues when selecting UI elements since they have access to internal structures of the native project. For cross-platform apps, the current state-of-the-practice would consist of one Appium script for each platform, as well as the adoption of well-known expressions such as `IdentifyAttributes` and `AbsolutePath`. As the proposed approach employs only one script, the effort spent to produce it would be at most half of the current state-of-the-practice. Moreover, the obtained results gave evidence that `ExpressionsInOrder` is more effective than individual expressions (e.g. `IdentifyAttributes` and `AbsolutePath`), imposing little overhead. This means that scripts generated by **`x-PATeSCO`** would demand less maintenance effort when testing in different devices. Yet, **`x-PATeSCO`** has previously discussed limitations we intend to overcome before adopting it in practice. We plan to have more quantitative and formal comparisons by conducting controlled experiments. Another direction is to perform case studies with partner IT companies in a real-world context of cross-platform app development.

Testing of cross-platform apps is essential if developers hope to cover as many devices as the existing in the market. This motivates cloud services, such as Amazon Device Farm and BitBar, to offer test features over hundreds of devices. The proposed approach aims to shed some light on how to automate such tests. Unfortunately, each mobile platform has its own UI representation, which also varies between different versions of the same OS. Consequently, we also investigated and compared strategies to locate UI elements. The overall analysis suggests that `ExpressionsInOrder` should be used along with the approach, though we recognise that there is still room for improvements. While the generalisation is not possible, we experimented our approach in apps developed using different frameworks and the results gave evidence that test scripts can be generated for other cross-platform apps. We hope that the approach, **`x-PATeSCO`**, and the results obtained help to pave the way for future advances.

7 Related work

This section describes the most relevant research in the area. Lee *et al.* [34] have developed a framework named HybriDroid for static code analysis of hybrid apps in Android. HybriDroid analyses the intercommunication between Java and Javascript (native and Web environment), exploring Java classes extracted from the apps. The authors evaluated 88 hybrid real-world apps from play store, detecting 24 faults in 14 apps. Different from our approach, Lee *et al.* focused on static analysis and not automated testing. Moreover, only hybrid apps in Android were taken into account.

Wei *et al.* [35] conducted an empirical study to understand and characterise fragmentation issues in Android apps. The authors investigated 191 compatibility issues collected from five open source Android apps to understand their root causes, symptoms, and fixing strategies. The study led to a technique called FicFinder to detect automatically compatibility issues. While compatibility bugs might be even a bigger problem for cross-platform apps, we concentrate on test execution. To detect such bugs, one might argue that insights on automated tests in several configurations are mandatory.

Boushehrinejadmoradi *et al.* [1] analysed the cross-platform app development framework Xamarin. They developed a tool called X-Checker to uncover framework inconsistencies by comparing the performance of different builds (one for Android and other for iOS) through differential testing. They generated 22,645 test cases, which invoke 4758 methods implemented in 354 classes from 24 Xamarin DLLs. Overall, they found 47 inconsistencies in the Xamarin code. The authors aimed to verify the cross-platform framework, while we deal with the testing of the apps developed by such frameworks.

Li *et al.* [17] propose an approach and tool, called ATOM, to automatically maintain app UI test scripts for regression testing. ATOM uses event sequence models (ESMs) to describe the app's behaviour, as well as a delta ESM to capture changes introduced by a new version. The delta helps to understand the impact of changes and to make adjustments in the test scripts. ATOM was applied to maintain test scripts of 11 Android apps, using 22 different versions (two versions of each app). Maintenance of test scripts is important for apps with constant updates. We plan to investigate how the scripts generated by our tool perform in such a scenario.

Fazzini *et al.* [14] have implemented a technique for (i) recording the user interaction with the app, as well as the definition of assertion-based oracles; (ii) generating test cases based on the recorded interactions; and (iii) executing test cases on multiple devices, summarising the test results. The generated UI test cases are based on the Espresso framework (https://google.github.io/android-testing-support-library). The authors implemented a tool called Barista, which was evaluated with 15 human subjects, 15 real-world Android apps, and seven (real) devices. Overall, the average compatibility rate across all apps and devices was 99.2%. Barista focuses on Android apps and test scripts coded in a platform-specific framework, while this study aims to automate the testing of cross-platform apps running on different platforms (Android and iOS) and configurations.

Joorabchi *et al.* [13] assume that, ideally, a given cross-platform app must provide the same functionality and behaviour in the various platforms. They introduced a tool, called CHECKCAMP, that detects inconsistencies between versions of the same native app programmed for different platforms, namely Android and iOS. During the execution, a parser dynamically intercepts calls to methods implemented and captures data from the UI after the return of these methods. The tool then generates a template for both platforms and maps their nodes. In the end, these models are compared to uncover inconsistencies. CHECKCAMP was evaluated with 14 pairs of apps (apps contained in the two platforms), detecting 32 valid inconsistencies. Their work and ours are similar in the way both aim to test cross-platform apps. However, our approach focused on black-box test execution of cross-platform apps (developed from the same code base) in multiple configurations, while CHECKCAMP requires code instrumentation (white-box) and aims to detect inconsistencies, a common problem with two or more code bases.

Automated test input generation (ATIG) tools have shown promising results on fault detection and code coverage for Android apps. Such tools automatically walk around the UIs of mobile apps using different exploration strategies. ATIG is out of the scope of this study, though the results herein presented can foster the development of similar tools for cross-platform apps. Currently, the tools are mostly for Android and do not aim to replicate the same test in different configurations.

8 Conclusion and future work

This study has introduced an approach to generate scripts to test cross-platform mobile apps in multiple configurations. To do so, we have also investigated XPath-based strategies to locate UI elements. Six individual expressions have been researched and two combined strategies (ExpressionsInOrder and ExpressionsMultiLocator) were derived from them. We have developed a tool, called **`x-PATeSCO`**, to implement the approach as well as the locating strategies. To evaluate the approach and compare the locating strategies, we conducted an experimental

study with nine cross-platform mobile apps tested on six different configurations.

The results have shown that four strategies (AbsolutePath, AncestorIndex, ExpressionsInOrder, and ExpressionsMultiLocator) are applicable to all events under test. Strategies based on key attributes (AncestorAttributes, ElementType, CrossPlatform, and IdentifyAttributes) are less applicable. When compared with known expressions employed in practice, they outperformed IdentifyAttributes and AbsolutePath for event executability by 8.9% and 17.6%, respectively. For test case executability, IdentifyAttributes reached 75% though being less applicable for all test cases and configurations are taken into account (14.2%). As for execution time, combined strategies were slower than individual expressions; ExpressionsInOrder was around six times faster than ExpressionsMultiLocator.

The results evinced that ExpressionsInOrder is the locating strategy recommended, being highly applicable and with reasonable executability ratio and time. Nevertheless, future advances might be achieved to improve the results herein provided. We plan to research different individual expressions and new combined strategies to enhance the applicability and executability. Another question to look into is how the tool performs with an extensive set of mobile devices accessible through cloud services such as Amazon Device Farm, BitBar, and others. Finally, the maintenance of UI test scripts is also a topic worth investigating in evolving cross-platform apps.

9 Acknowledgments

The authors thank the partner IT companies that provided the industrial apps used in this study. Special thanks to the independent participants who helped to elaborate on the test cases adopted in the experimental evaluation. The authors are also grateful to the anonymous reviewers for their useful comments and suggestions. A.T.E. is partially financially supported by CNPq/Brazil (grant no. 420363/2018-1).

10 References

[1] Boushehrinejadmoradi, N., Ganapathy, V., Nagarakatte, S., *et al.*: 'Testing cross-platform mobile app development frameworks'. 2015 30th IEEE/ACM Int. Conf. on Automated Software Engineering (ASE), Lincoln, NE, USA, November 2015, pp. 441–451

[2] IDC.: 'Smartphone OS market share, 2017 Q1'. 2017. Available at https://www.idc.com/promo/smartphone-market-share/os, accessed 17 April 2019

[3] Gartner.: 'Gartner says worldwide sales of smartphones grew 9 percent in first quarter of 2017'. 2017. Available at https://www.gartner.com/newsroom/id/3725117, accessed 17 April 2019

[4] Statista.: 'Number of apps available in leading app stores as of march 2017'. Available at http://www.statista.com/statistics/276623/number-of-apps-available-in-leading-app-stores/, accessed 17 April 2019

[5] IBM.: 'Native, web or hybrid mobile-app development'. Somers, NY. Available at ftp://public.dhe.ibm.com/software/pdf/mobile-enterprise/WSW14182USEN.pdf, accessed 17 April 2019

[6] W3C.: 'Web design and applications'. Available at https://www.w3.org/standards/webdesign/, accessed 17 April 2019

[7] Xanthopoulos, S., Xinogalos, S.: 'A comparative analysis of cross-platform development approaches for mobile applications'. ACM Proc. 6th Balkan Conf. in Informatics, BCI'13, New York, NY, USA, 2013, pp. 213–220. Available at http://doi.acm.org/10.1145/2490257.2490292

[8] Willocx, M., Vossaert, J., Naessens, V.: 'Comparing performance parameters of mobile app development strategies'. 3rd Int. Conf. on Mobile Software Engineering and Systems (MOBILESoft), 2016, pp. 38–47. Available at http://dl.acm.org/citation.cfm?doid=2897073.2897092

[9] Willocx, M., Vossaert, J., Naessens, V.: 'A quantitative assessment of performance in mobile app development tools'. 2015 IEEE Int. Conf. on Mobile Services (MS), New York, NY, USA, , 27 June – 2 July 2015, pp. 454–461

[10] Gronli, T.M., Ghinea, G.: 'Meeting quality standards for mobile application development in businesses: a framework for cross-platform testing'. IEEE 49th Hawaii Int. Conf. on System Sciences (HICSS 2016), Koloa, HI, USA, January 2016, pp. 5711–5720

[11] Joorabchi, M.E., Mesbah, A., Kruchten, P.: 'Real challenges in Mobile app development'. IEEE Int. Symp. on Empirical Software Engineering and Measurement (ESEM 2013), Baltimore, MD, USA, October 2013, pp. 15–24

[12] Nagappan, M., Shihab, E.: 'Future trends in software engineering research for Mobile apps'. 2016 IEEE 23rd Int. Conf. on Software Analysis, Evolution, and Reengineering (SANER), Suita, Japan, March 2016, pp. 21–32

[13] Joorabchi, M.E., Ali, M., Mesbah, A.: 'Detecting inconsistencies in multi-platform mobile apps'. IEEE 26th Int. Symp. on Software Reliability Engineering (ISSRE 2015), Washington DC, USA, November 2015, pp. 450–460

[14] Fazzini, M., Freitas, E.N.D.A., Choudhary, S.R., *et al.*: 'Barista: A technique for recording, encoding, and running platform independent android tests'. 2017 IEEE Int. Conf. on Software Testing, Verification and Validation (ICST), Tokyo, Japan, March 2017, pp. 149–160

[15] Leotta, M., Stocco, A., Ricca, F., *et al.*: 'Reducing web test cases aging by means of robust XPath locators'. Proc. – IEEE 25th Int. Symp. on Software Reliability Engineering Workshops, ISSREW 2014, Naples, Italy, November 2014, pp. 449–454

[16] Leotta, M., Stocco, A., Ricca, F., *et al.*: 'Using multi-locators to increase the robustness of web test cases'. 2015 IEEE 8th Int. Conf. on Software Testing, Verification and Validation (ICST), Graz, Austria, April 2015, pp. 1–10

[17] Li, X., Chang, N., Wang, Y., *et al.*: 'ATOM: automatic maintenance of GUI test scripts for evolving Mobile applications'. Proc. 10th IEEE Int. Conf. on Software Testing, Verification and Validation, (ICST 2017), Tokyo, Japan, March 2017, pp. 161–171

[18] Zein, S., Salleh, N., Grundy, J.: 'A systematic mapping study of mobile application testing techniques', *J. Syst. Softw.*, 2016, **117**, (C), pp. 334–356. Available at https://doi.org/10.1016/j.jss.2016.03.065

[19] IEEE: '*IEEE standard glossary of software engineering terminology*', vol. **12**, (IEEE, 1990). Available at http://ieeexplore.ieee.org/xpls/absall.jsp?arnumber=159342

[20] W3C.: 'Selectors level 3'. Available at https://www.w3.org/TR/2011/REC-css3-selectors-20110929/, accessed 17 April 2019

[21] W3C.: 'XML path language (XPath) version 1.0'. Available at https://www.w3.org/TR/xpath/, accessed 17 April 2019

[22] Vilkomir, S., Marszalkowski, K., Perry, C., *et al.*: 'Effectiveness of multi-device testing Mobile applications'. 2nd Int. Conf. on Mobile Software Engineering and Systems (MOBILESoft), Florence, Italy, May 2015, pp. 44–47

[23] Vilkomir, S., Amstutz, B.: 'Using combinatorial approaches for testing mobile applications'. 2014 IEEE Seventh Int. Conf. on Software Testing, Verification and Validation Workshops, Cleveland, OH, USA, 31 March – 4 April 2014, pp. 78–83. Available at http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6825641

[24] Belli, F., Budnik, C.J., White, L.: 'Event-based modelling, analysis and testing of user interactions: approach and case study', *Softw. Test Verif. Reliab.*, 2006, **16**, (1), pp. 3–32. Available at http://dx.doi.org/10.1002/stvr.v16:1

[25] Leotta, M., Clerissi, D., Ricca, F., *et al.*: 'Comparing the maintainability of selenium webdriver test suites employing different locators: a case study'. Proc. 2013 Int. Workshop on Joining AcadeMiA and Industry Contributions to Testing Automation, Jamaica, Lugano, Switzerland, July 2013, pp. 53–58

[26] Leotta, M., Clerissi, D., Ricca, F., *et al.*: 'Repairing selenium test cases: an industrial case study about web page element localization'. 2013 IEEE Sixth Int. Conf. on Software Testing, Verification and Validation, Luxembourg, March 2013, pp. 487–488

[27] Rao, G., Pachunoori, A.: 'Optimized identification techniques using XPath'. MSU-CSE-00-2, IBM Developerworks, 2013

[28] Menegassi, A.A., Endo, A.T.: 'x-PATeSCO and experimental package'. Available at https://github.com/andremenegassi/x-PATeSCO, accessed 17 April 2019

[29] Statista.: 'Android version market share distribution among smartphone owners as of September 2017'. Available at https://www.statista.com/statistics/271774/share-of-android-platforms-on-mobile-devices-with-android-os/, accessed 17 April 2019

[30] Statista.: 'Share of Apple devices by iOS version worldwide from 2016 to 2017'. Available at https://www.statista.com/statistics/565270/apple-devices-ios-version-share-worldwide/, accessed 17 April 2019

[31] Silva, D.B., Eler, M.M., Durelli, V.H.S., *et al.*: 'Characterizing mobile apps from a source and test code viewpoint', *Inf. Softw. Technol.*, 2018, **101**, pp. 32–50. Available at http://www.sciencedirect.com/science/article/pii/S0950584918300788

[32] Leotta, M., Stocco, A., Ricca, F., *et al.*: 'Robula+: an algorithm for generating robust XPath locators for web testing', *J. Softw.: Evol. Process*, 2016, **28**, (3), pp. 177–204. Available at https://onlinelibrary.wiley.com/doi/abs/10.1002/smr.1771

[33] Cruz, L., Abreu, R., Lo, D.: 'To the attention of mobile software developers: guess what, test your app!', *Empir. Softw. Eng.*, 2019, **24**, pp. 2438–2468. Available at https://doi.org/10.1007/s10664-019-09701-0

[34] Lee, S., Dolby, J., Ryu, S.: 'Hybridroid: analysis framework for android hybrid applications'. 31st IEEE/ACM Int. Conf. on Automated Software Engineering (ASE 2016), Singapore, September 2016, pp. 250–261

[35] Wei, L., Liu, Y., Cheung, S.C.: 'Taming android fragmentation: characterizing and detecting compatibility issues for android apps'. 31st IEEE/ACM Int. Conf. on Automated Software Engineering ASE '16, Singapore, September 2016, pp. 226–237