An analysis of widget layout attributes to support Android GUI-based testing

An analysis of widget layout attributes to support Android GUI-based testing

Tommaso Fulcini, Riccardo Coppola, Marco Torchiano and Luca Ardito
Department of Control and Computer Engineering
Politecnico di Torino
Turin, Italy
first.last@polito.it

***Abstract*—In the context of GUI testing, identifying robust locators (i.e., attributes to unambiguously identify on-screen widgets to be used in test sequences) is still considered an unsolved challenge by the researchers' community. The frequent variation of attributes between different releases of the System Under Test (SUT) leads in fact to testing fragility, i.e., test case failing because of invalidated locators. Recent studies have highlighted the benefits of adopting multi-locator approach, i.e., the combination of multiple locators to enhance the robustness of widget identification.**

The objective of this work is to provide insights into the composition of Android applications, assessing the characteristics of different layout-based properties and their suitability to be used as locators for widgets in the context of GUI-based testing.

We investigated the state of the practice by analysing the distribution of widget values within 30 real apps selected from the Google Play Store. For those apps, we selected two different versions to examine how they evolved over time from both visual and structural perspectives.

The results of our analysis showed that missing values, variability, and instability of attributes make it hardly possible to identify a single attribute or technique (either coordinate-, property-, or visual-based) capable of providing robust GUI testing over multiple releases of mobile SUTs.

***Index Terms*—GUI testing, GUI testing repair, Software Testing, Software Engineering, GUI testing fragility**

I. Introduction

The mobile application industry is growing; its revenues are expected to reach 201 billion dollars by 2023, according to recent projections [1]. From 2016 to 2021, the yearly number of global app downloads increased by 63.5% with an average year-over-year growth rate of 10.56% [2].

These impressive figures attracted the interest of several companies that started offering support for mobile applications in most of their services, especially during 2020, when due to the Covid-19 pandemic, most of the work has been done remotely. In fact, business app downloads increased by 102% in 2020, with a massive 225% spike solely in march of 2020 [2].

As a consequence, this growing interest in mobile application development brought the necessity of having the shortest possible time to market for companies launching their own apps. In this scenario, one of the most frequent concerns is the need for a thorough testing phase, which is still predominantly performed superficially or neglected, being a costly and time-consuming activity.

GUI testing plays a key role in this context, as it allows direct user interaction to be simulated. GUI testing, especially when automated, makes it possible to quickly and reliably demonstrate the functional correctness of the Application Under Test (AUT). Having a suite of automated tests also makes it possible to quickly perform the same tests on demand on the target AUT, guaranteeing the non-regression of the application if the tests are successful.

When an automated test suite is built, its purpose is to ensure that all the components of an app are correctly working: when a test case fails, it should reflect misbehavior in the application. Although this is mostly always true when testing the application with a low-level focus, such as unit testing and integration testing, when testing apps through their GUI, tests may also fail due to changes in their locators. Locators are the means to identify an element on-screen to target, allowing tests to access widgets' functionalities.

During the evolution of an app, it is common that the visual appearance of its GUI, the inner description, or its properties change over time, either with small attribute differences in certain widgets or with a radically different visual appearance on specific screens. In both cases, tests might fail when the locators of the widgets change between different releases, with the necessity of repairing tests with a new, correct locator. These test failures are not caused by a malfunction of the app itself but by a failure in the location process of a widget. Therefore, additional effort is required at each test failure to (1) understand whether the failure reflects an actual defect; and (2) fix the locators that are no longer valid (namely, *broken* locators). This widespread issue for GUI testing is known as *fragility* by literature in the field.

To address the issue of fragile tests, we aim to define a more robust widget localization strategy than the state-of-the-art localization strategies that are based on single locators (i.e., they use a single locator value to identify an attribute on the screen of the app). Our objective is to study the use of a combination of multiple locator values to guarantee higher location robustness. This paper reports the results of a preliminary study aimed at studying the distributions of values of locators in popular Android applications to understand which attributes are less prone to changes and can guarantee more stability to test suites when used as locators. The study replicates the approach used in a study by Nass et al., [3],

that performed the same analysis for web-based application testing.

In our study, we analyze how 30 applications from Google Play Store[1] are assembled, breaking down all the attributes of their widgets. This analysis will provide the following insights regarding Android apps: a set of guidelines and recommendations for developers, a starting point to develop a new test localization algorithm, and directives on how to select and identify corresponding widgets.

II. Background and Related Works

Android apps are software applications designed to run on Android, an operating system with a layered architecture, consisting of several layers including the Linux kernel, native libraries, the Android runtime (with a Virtual Machine able to be run on several devices), the application framework, and the user-facing apps.

In the present section, we only focus on the top layer and its testing-related issues and challenges, leaving out what lies underneath.

The topmost layer is the user-facing apps, which are the apps that the user interacts with. They are handled by the Application framework, an Android component that takes care of the lifecycle of the app, providing the app with the needed services and resources such as layouts and strings. Among them, built-in apps such as the home screen, the phone dialer, and the browser are included.

When it comes to the architecture of Android applications, there are several popular architecture patterns, such as Model-View-Controller (MVC), Model-View-ViewModel (MVVM), and Model-View-Presenter (MVP) which are used to structure the code. These patterns help separate the concerns of the app's data, user interface, and control flow. In all three architectures, the concept of separation of concern is the main focus, intending to separate functional duties from the layout and aesthetic. The first implementation of Android Components decoupling is based on the declaration of UI layouts in XML files (containing the definition of the static visual aspect of a component) and their inflation at runtime. The said components can also be updated programmatically during their lifecycle, with changes in the visual aspects or in their internal state.

Newer recommendations (March 2022) from Google developers suggest relying on Jetpack Compose toolkit (a collection of Kotlin APIs) for UI development [4]. It appears to be an intuitive declarative way of defining flexible stateless components that can be reused, customized, and tested easily.

Testing Android apps through their Graphical User Interface (GUI) can be cumbersome and time-consuming due to their complexity and the gestures-driven nature of the interactions to be reproduced. Evidence in the literature shows that automated testing approaches are far from being adopted on a large scale [5] [6]. GUI test suites can be automatically generated, with a low cost associated with the generation process but a higher maintenance cost, or manually scripted, with an inherent building complexity.

Automated GUI testing of Android applications can be classified based on different criteria. The first general classification proper of scripted GUI testing, valid for Android apps and web applications, is based on the widget localization strategy. There exist three main methods to find a widget in an Android app during its execution [7]:

1) Coordinate-based locators: elements are identified by means of the coordinates relative to the position on the screen of an element;
2) Layout-based locators: widgets are accessed via the app layout, using their properties value;
3) Image-based locators: starting from a screenshot some algorithms for image recognition are used to find the corresponding Android element.

The first technique (also called *first-generation* locator) is nowadays considered outdated due to the high device fragmentation in Android-based smartphones, which reflects a wide variety in screen sizes. First-generation based locators are reportedly considered to be fragile due to their strong dependence on the device on which the app runs [8]. The second location strategy is one of the most used and widespread to date [9]. It uses a specific value of a layout property to retrieve the corresponding Android widget. It is considered a more refined approach with respect to the previous one, in fact, it is also called a *second-generation* technique. Lastly, the *third-generation* and more recent manner are based on image recognition algorithms, able to detect, from an existing screenshot, the most similar screen element.

Although progress in improving both the second and the third approaches has been made over the years, some frailties remain inherent in the nature of the approaches. Namely, test breakage can be caused by a failing locating property for second-generation ones and by a radical appearance change for those of third-generation. For example, a GUI test checking for a specific button on the screen may break if the button's location or text changes. Similarly, a test that checks for the presence of a specific element on the screen may fail if that element is removed or its layout is modified. This phenomenon is known as test fragility.

In the existing literature, several ways to mitigate test fragility in Android exist. One approach is to use the Screen Object Design Pattern [10], a mobile adaption of the Page Object Design Pattern [11], which separates the test code from the implementation details of the app, making the test more loosely coupled to the app logic and more robust to changes. Another approach is to use more robust selectors to identify application layout elements. Robust locators are selectors that are less likely to break when changes are made to the code or layout of the application. They are generally based on stable and reliable widget properties; nevertheless, it should be noted that they are not immune to failure if the identifying properties change or if the elements they refer to are removed.

According to the literature review carried out by Nass et al., this topic is one of the main issues that the community should

[1] https://play.google.com/store/apps

address, and that could reflect improvements in other existing challenges [12].

III. Research Methodology

The final goal of our study is to analyze the composition of popular Android applications in terms of the attributes used and the different values that these attributes can assume. We also want to analyze how these attributes can vary over time when the applications evolve.

To review the composition of different applications, we decided to assess real apps from the market. In the following section, we will explain the research method, and the selection process followed to identify the pool of subjects to analyze, and then we will discuss the widget analysis process we adopted.

A. Research Questions

To drive our research, we first established our research questions (RQ) to frame the remainder of the methodology:

- **RQ1**: What is the variability of the attributes in Android applications, and how frequently are they valued with meaningful values?
- **RQ2**: How often do widgets change visually and how do their properties change between two releases?

The first RQ is meant to review how attributes are used, assessing the presence and usage of their values: to answer this question we will divide this RQ into separate analyses.

The first aspect we consider is the population of attributes. Some of the attributes are in fact related to functional aspects of the widget (e.g. *clickable*, *focused*, *scrollable*, ...). These attributes are typically boolean and always valued (default to false). Another category of widgets represents the textual content of the widget, if any, or provides a description to identify the widget (e.g., *id*, *content-desc*, ...). For these widgets categories, we seek to understand how frequent the possibility to have a value, and how diverse is the set of values that are assumed over the applications. The rationale behind this analysis is that an attribute is more suitable to be used as a locator if it is frequently valued (i.e., not empty) and if it is sufficiently diverse over the set of applications. Attributes that have low variability, in fact, will not be able to distinguish efficiently between different widgets on the same screen or application.

For the second RQ, we assess the way the attribute values evolve over two different releases. We manually define a set of corresponding widgets, i.e. widgets that are present in two versions of the applications with the same functionality, and we compare the attribute values in the two versions in order to observe if and how their attributes change over time. Computing these statistics allows us to estimate the stability of the attributes, indicating which are the best attributes to be used as a locator (the less stable, the more is likely to break). The second step of this RQ is meant to shed light on the visual differences that can be appreciated by a visual inspection of the GUI: we classify the changes according to the visual mutation characterization provided by Alégroth et al. [13] in order to associate to each corresponding widget its alteration if any. For our labeling process, we adopted a subset of the full characterization (reported in Table II), since some mutant operators did not suit our case. In particular, removal and insertion are not applicable as, by construct, we searched corresponding widgets present in two app versions. Other types of mutations related to window resizing cannot be replicated for Android applications, as resizing was not enabled in the considered virtual device. We considered mutant operator pairs M11-M13 and M15-M16 as mutually exclusive. The first is because both mutants involve a modification of the original coordinates, respectively by assuming a generic different value, and by overlapping other widgets. The pair M15-M16 inherently is referred an alteration of the visual aspect of the widget, either due to a change in the type of a widget or with a general variation. During the labeling process, we noted that each modification in the type of a widget reflected its appearance: to achieve a more specific characterization we decided to mark the said condition as the M16 mutant operator. Finally, to express cases where nothing changed we introduced the mutant operator M0.

B. Dataset selection

To analyze the variability and presence of valued attributes, we resorted to selecting a set of popular mobile applications. To find a set of suitable applications, we used two web resources: AppBrain[2], a website that provides statistics about the popularity of Android applications released on the Google Play Store; and APKMirror[3], a website providing the history of packaged Android apps (APK files), as opposed to the Google Play Store where only the last release of each application can be found.

To guarantee external validity to our results we selected popular applications for all the categories that are listed on the AppBrain website. We thus browsed the website by category, sorting the results according to the most downloaded apps.

Following the popularity sorting of the application, we searched the apps on APKMirror, where several past versions of the same app can be found. We selected only applications for which multiple versions were available. For each app, we downloaded the latest version and applied the Inclusion Criteria that are described below. Each time the first four criteria were met, we iteratively tested each version available on APKMirrorfrom the newest to the oldest, until we found a version that met condition IC5.

To determine whether an app was eligible to be included in the study, we defined a set of Inclusion Criteria (IC), each app had to fully meet all the IC. We do not list explicit Exclusion Criteria (EC) since we considered them as the opposite of the ICs. The inclusion criteria have been defined as follows:

- **IC0**: The app is ranked in AppBrainand at least two different versions are available for download in APKMirror.

[2]https://www.appbrain.com
[3]https://www.apkmirror.com

TABLE I: List of selected apps and versions

Category	Name of the app	Old Version	Old Version Date	New Version	New Version Date
Art	Sketchbook	4.1.5	1/22/2019	5.3.1	11/27/2022
Auto	CarMax	2.47.1	8/30/2018	3.7	8/13/2019
Beauty	Mirror Plus	2.9.1	9/24/2016	4.2.1	9/25/2022
Books	YouVersion Bible app	6.4.2	12/21/2016	9.16.2	11/2/2022
Business	UPS mobile	4.5.0.1	8/7/2016	9.7.72	11/16/2022
Comics	Cdisplayex	1.1.107	11/27/2018	1.3.36	11/3/2022
Communication	Firefox Fast & Private Browser	65.0.1	2/13/2019	107.1.0	11/14/2022
Education	Google Classroom	4.5.212	6/12/2018	8.0.421	11/2/2022
Entertainment	Tubi	3.7.0	12/1/2020	4.21.1	3/10/2022
Events	Gametime	11.2.15	3/28/2019	2022.17.2	11/2/2022
Finance	Wise	7.29.1	10/13/2021	7.83.1	11/28/2022
Food	Burger King	6.2.0	5/14/2019	6.25.8	11/8/2022
Health	Calm	5.1	8/10/2020	6.12.1	11/17/2022
House	Angi	21.0.18	11/26/2021	22.46.0	11/21/2022
Libraries	Allinone toolbox	6.4.3	8/8/2016	8.2.8.1	7/18/2022
Lifestyle	Pinterest	8.39.0	10/24/2020	10.42.0	11/17/2022
Maps	Transit	4.3.1	11/16/2017	5.11.3	4/13/2022
Media	Video downloader for Instagram	1.1.98	9/21/2020	1.1.60	7/27/2018
Medical	Nevada COVID trace	1.4.7	9/14/2022	1.2.11	9/14/2022
Music	Soundcloud	2017.12.14	12/14/2017	2022.16.11	11/16/2022
News	NewsBreak	4.6.4	11/27/2019	22.47.0	11/25/2022
Personalization	Backgrounds hd	4.8.25	1/17/2017	5.0.052	1/19/2022
Photography	Lightroom	4.4	8/13/2019	8.0.1	11/18/2022
Productivity	HP smart	4.7.104	5/1/2018	9.6.2.3732	11/10/2022
Shopping	Walmart	22.1.1	1/15/2022	22.43.4	11/19/2022
Social	Reddit	2020.30.0	8/13/2020	2022.44.0	11/30/2022
Sports	FOX Sports	5.0.0	7/20/2020	5.60.0	11/22/2022
Tools	Google translate	5.12.0	9/8/2017	6.49.0	11/9/2022
Travel	Booking.com	24.0	11/9/2020	34.5	11/18/2022
Weather	AccuWeather	4.8.2	7/6/2017	8.7.1	11/17/2022

TABLE II: List of GUI mutants.

#	Mutant Operator
M0	No changes affected the widget
M11	Modify the location of a widget to a proper location
M13	Modify the location of a widget to overlap with another
M14	Modify the size of a widget
M15	Modify the appearance of a widget
M16	Modify the type of widgets (e.g. Button changed to TextView)

- **IC1**: Both the selected versions of the application had to run without crashes, and without requiring any system updates on the Android Virtual Devices.
- **IC2**: The two versions selected must have at least one perceptible visual difference.
- **IC3**: It must be possible to identify at least one corresponding widget, not necessarily with the same aspect.
- **IC4**: The application must not require an explicit registration or login process to directly reach the home screen. This criterion does not include the usage of a login based on external APIs (such as login via Google, Facebook, or Twitter), whose case is considered valid.
- **IC5**: The older version must be the most recent version working on the Google Nexus 5X Android Virtual Device (AVD).

Of the 32 categories found in AppBrain, 30 apps from different categories were selected: for two categories (namely, Dating and Parenting) all the apps on the retrieved list did not fulfill the Inclusion Criteria. For this reason, these two categories were excluded.

The complete list of used apps can be seen in Table I: for each application, we report its category, the package name, the release name, and the publication date of the two versions considered.

C. Analysis Process

For each selected app, once the two selected versions were downloaded and installed in the Google Nexus 5X AVD, a setup process was necessary to prepare the app to get to the home screen, i.e. performing login with external APIs when necessary, and closing all the one-time screens or wizards appearing the first time an app is opened on a device. Then, we started analyzing its XML layout, which was obtained by means of the command UIAutomator dump.

Once obtained the dump file, we created a script to filter out all the Android Views that are used only as containers and that are never directly visualized on screen. We then parsed all the remaining widgets to create a CSV file containing the names of all the possible attributes and, for each widget, the values assumed. Reviewing the obtained files, we noted that no unique identifier, corresponding to the XPath for the web, was present in Android apps. Thus, we decided to introduce

in the generated CSV file a progressive number allowing us to distinguish between all the widgets unambiguously. The CSV file containing the attribute values was obtained, for each application, for both the old and the new version.

We took a screenshot of the home screen for each version, in order to easily get the actual visual representation if needed during the analysis process.

To analyze how the attributes varied for the same widget in two different versions of the same application, we had to identify all the pairs of corresponding widgets, i.e., widgets that have the same role in the old and new versions of the application. To get this set of corresponding widget pairs, we manually inspected all the pairs of versions of the 30 selected apps and identified widgets that had the same conceptual meaning and were used to perform the same operations in the application. By applying this manual identification of widget correspondence, we came up with a set of 201 corresponding widget pairs. Then we retrieved – through a lookup in the CSV files – the variations in the attributes that were linked to the same conceptual widget in the application. Using this information it is possible to assess the suitability of an attribute as a locator from a perspective of attribute evolution. The key criterion is that an attribute which is more likely to change in future releases of the application can be considered less dependable as a locator.

IV. Results

To answer the identified Research Questions, we collected all the related data and plotted the corresponding distributions, in order to compare the different properties of the widgets.

A. RQ1 - Attribute Emptiness and Variability

The first analysis is based on the number of valued attributes. At first glance, we noticed that all attributes of the boolean type are endowed with values: this is due to the fact that they are set by default by UI Automator to their corresponding state at the time of the extraction of the dump file.

For this reason, we separated the two value analyses in order to avoid mixing information of different natures. In Figure 1 we report the distribution of the values that are empty (i.e., an empty string is present in the UIAutomator dump in correspondence of such attribute) or are valued (i.e., any string is present as a value in the dump in correspondence of such attribute). From the graph, it is possible to note that only three attributes (*resource-id*, *text*, *content-desc*) are not always characterized by the presence of values. The attributes that have 100% valued instances are those which are generated by Android Studio, or by UIAutomator itself. In particular, the *class* attribute always corresponds to the Java class of the widget, the *bounds* attribute refers to the upper-left and lower-right corners of the widgets on the screen. Instead, the *index* attribute reports the pointer to the widget in the hierarchical tree structure of the app. Finally, the *package* attribute is automatically set by Android Studio when a new View is generated in the context of the app. Although *NAF* is actually

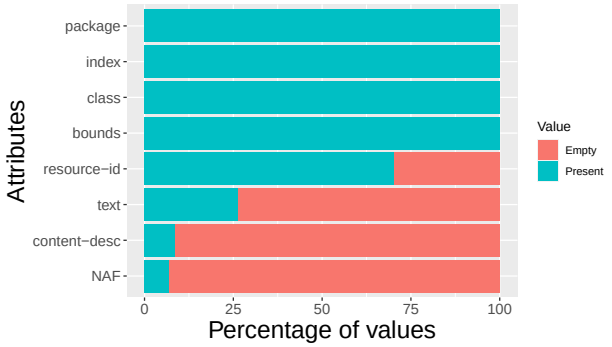

Fig. 1: Distribution of empty and valued attributes in the selected apps

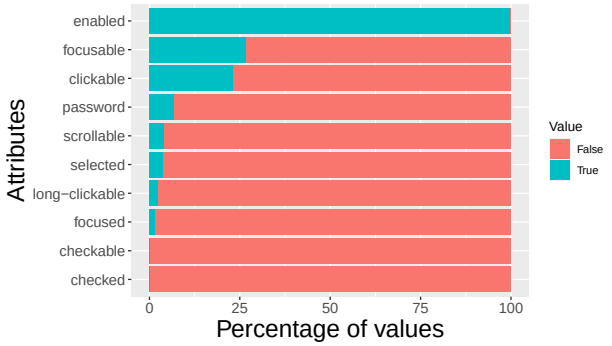

Fig. 2: Distribution of values for boolean attributes in the selected apps

a boolean attribute, in most cases it is not provided (in the form of a NULL Value).

In figure 2 we report the distribution of values for the boolean attributes extracted by the UIAutomator dump. We can see the distribution of values in the different attributes: most of the widgets are enabled, while only approximately 25% of them are clickable, i.e. allowing interaction. Although boolean values are by no means usable for the purpose of locating widgets per se, it is worth noting that, since their presence is always guaranteed by UI Automator, the use of a weighted combination of such values can be useful to corroborate others, yet stronger locators. This aspect will be discussed as an implication in the Future Work section.

It is worth noting that, even if an attribute is given a value frequently, such an attribute will be of no use as a locator if the value is always the same for all different widgets. In that case, in fact, it would not be possible to discriminate between different widgets based on such value of the locator. This leads to the need of deepening the analysis by mixing the information on the availability of values for an attribute, with the different values assumed by such specific attribute.

The second step for a deeper analysis regarding the nature of attribute usage is to consider the diversity of values that are assumed by an attribute. Figure 3 plot the measure of how different values are distributed for each property: in particular, the value refers to the ratio between the number of different values assumed over the number of valued widget properties.

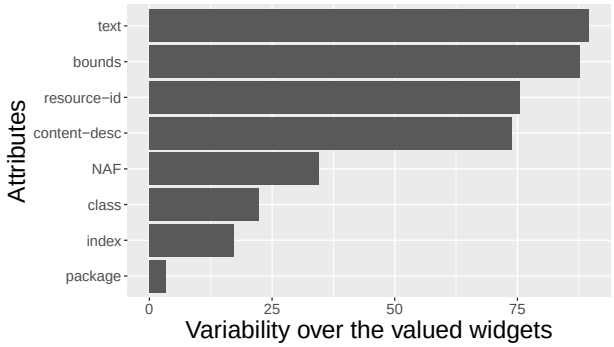

Fig. 3: Distribution of different values for each attribute over the number of valued attributes

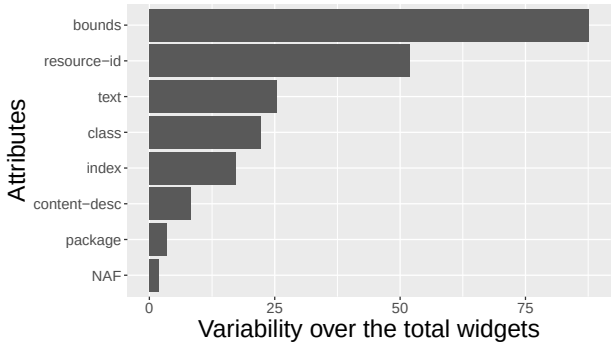

Fig. 4: Distribution of different values for each attribute over the total number of attributes

Although this indicator shows that the greatest variability is concentrated in the *text*, *NAF*, and *content description* attributes, from the previous finding we infer that these are more prone to be left unassigned. Thus, to combine the information regarding the variability with that coming from the previous RQ, we considered the ratio between the number of different values of a property and the total number of widgets. This particular distribution highlights that the *bounds* attribute is the most variable, with several different values, followed by *resource id* and *text* attributes.

Although the *bounds* attribute has the highest variability, the usage of this attribute as a location strategy can be traced back to outdated first-generation tools, characterized by well-known drawbacks. Nevertheless, corroborating position-related properties with other locators could harden existing location strategies.

Notwithstanding the result, it must be taken into account that, the value comparison was performed in a boolean way, which implies that for the *bound* attribute, the widgets must have coincident positions and dimensions to obtain two corresponding values. This constraint appears to be rare, leading us to wonder whether position-related metrics comparisons should be based on more sophisticated treatments such as the percentage of screen overlapping.

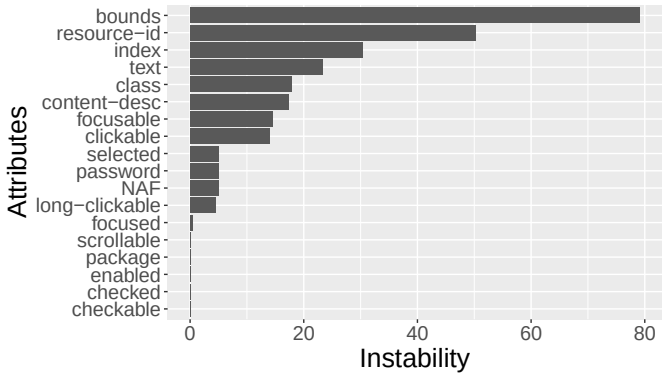

Fig. 5: Attribute instability for the selected corresponding widgets

B. RQ2 - Corresponding Widgets Analysis

The second research question concerns the detection of corresponding widgets, i.e. widgets that are present in different versions of an app with the same functionality. Defining a corresponding widget is important in software testing maintenance since it allows the identification of an artifact in the screen that, regardless of its possible graphical evolution, is associated with a precise function. RQ2 has been formulated to find a robust and reliable way to use attributes to identify corresponding widgets in an app.

To drive our analysis of how attribute values can be used to locate corresponding widgets when the application evolves, we manually selected and classified a set of corresponding widget pairs. For each of the selected corresponding widgets, we analyzed the value of all the attributes in both the selected versions to understand which attributes are the most stable during the evolution of the SUT.

Figure 5 ranks corresponding widgets attributes based on their variability over the two versions, i.e. their instability. The attribute resulting as the most unstable is unsurprisingly the *bounds* one (with an approximate value of 80% of instability), in accordance with the drawbacks of coordinate-based locators pointed out by the existing literature. For this reason, even if the said attribute has the highest presence and diversity in assumed values, its instability makes it unsuitable to be used as a locator. A change in *bounds* attribute is often reflected in the alteration in the value of *index* attribute since different coordinates may imply a change of position not only in the visual arrangement but also in the tree structure.

Resource-id is ranked second for instability with a variability of 50%: even though Android Developers recommend the usage of this attribute to identify a resource [14] uniquely, half of the widgets changes are actually affecting this identifier. This statistic partly invalidates the usage of the *resource-id* as a locator for testing since, differently from an internal usage where developers can refactor the identifier in the whole project, the access to widgets is based on the knowledge of the ids and the structure. Also, textual and content description attributes are quite unstable, with a variability of 23% and

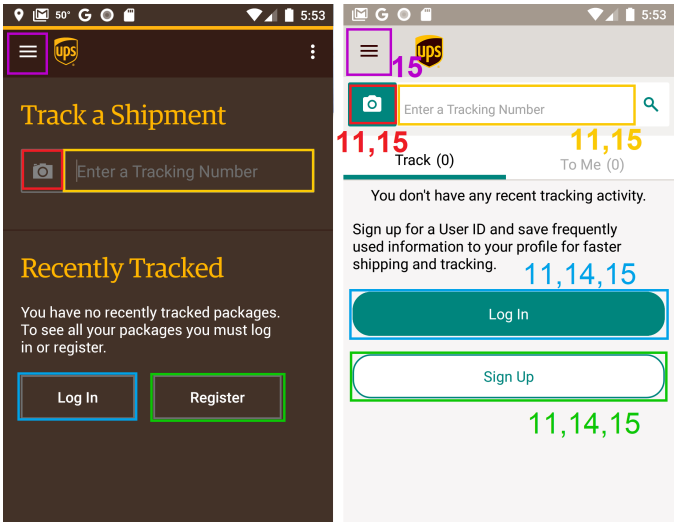

Fig. 6: An Example of visual changes affecting corresponding widgets between different releases

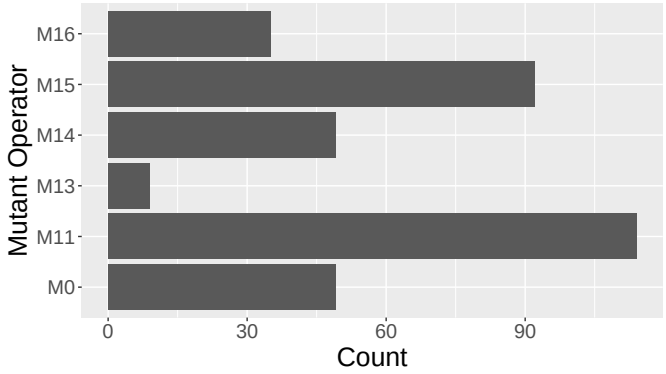

Fig. 7: Occurrence of visual changes for the selected corresponding widgets

17%, respectively, which, when added to the frequent absence of values, makes these attributes unreliable.

Out of the total of 201 corresponding widgets, 174 are those that have undergone mutation of at least one attribute. The total number of attribute value changes was 536, with an average of three attributes alteration.

Regarding visual mutations that corresponding widgets underwent, results are shown in Figure 7. Unsurprisingly Location Change is the one with the highest frequency (M11, 57% corresponding widget pairs), in accordance with the high instability of the *bounds* attribute. The second visual mutation by frequency is the change in the graphical appearance of a widget (M15, occurring in 46% corresponding widget pairs). In most cases, a visual alteration of the application was accompanied by at least a partial reorganization of the widgets (e.g. search bar and photo icon in Figure 6), while only a few cases presented a restructuring of the composition of the GUI while maintaining the same general appearance (what happened to the hamburger menu in Figure 6).

The rescaling mutation (M14, occurring in 24% corresponding widget pairs) ranked third, closely followed by the absence of mutation (M0, occurring in 24% corresponding widget pairs). It is partly due to the readjustment required following a change in the layout, as happens for ”Log in” and ”Sign Up” buttons in Figure 6, in other cases a rescale can be a symptom of a variation of the importance of the function associated with the widget.

Next to last, by frequency, was the mutation related to changes in the widget type (M16, occurring in 17% corresponding widget pairs): this type of mutation mainly concerns widgets that, from a given release, acquire new functions that cannot be associated with the previous type. A recurring example is a transition from ImageButton to ImageView or vice versa.

The less frequent mutation among those considered is a change in the position to overlap a different widget (M13, occurring in 4% corresponding widget pairs), meaning that maintaining a clean and uncluttered look is important for the graphical evolution of an app. An exception is the inclusion of a Floating Action Button that allows a particular functionality to stand out from the rest of the GUI, being clearly visible.

V. Discussion

The results that we gathered to answer our research question allowed us to collect information about three properties of Android widget attributes, when used as locators: *presence*, *variability* and *instability*.

The presence of an attribute is an index of how often such an attribute is provided with a meaningful value. We consider the presence of values as a fundamental prerequisite for an attribute to be a valid locator for a widget.

We defined variability as the possibility for an attribute to assume different values for different widgets. Therefore, a high variability for an attribute over a widget population can encourage the use of such an attribute as a locator, since it is more likely that a specific value of the attribute can identify a single widget on a given SUT.

Finally, we defined instability as the possibility that the same attribute, on the same widget, varies when the application evolves. We assume that a high instability of an attribute makes it unsustainable as a locator over different releases of an application. That would, in fact, require multiple test case repairs to fix the locator values that are changed.

As a last analysis, we performed an inspection of the visual mutations that were applied to the widgets during the evolution of the SUTs, to analyze if the changes in the widgets were only related to their layout attribute values or were also reflected by their on-screen appearance.

Based on our findings, we can argue the following about locators for Android application testing:

- Regarding *presence*, some attributes are always present by construction (package, index, class, bounds). Of them, only the bounds assume values that are specific to different elements on a screen;

- Regarding *variability*, text, resource id, and content description have a very high variability over different attributes so they may have an use as locators. However, such attributes are not always valued (e.g., content-desc is present in less than 10% of widgets) so they cannot be used as locators alone;
- Regarding *instability*, we measure a very high change rate for bounds (as expected) but also for resource-ids, undermining the suitability of such attribute as a locator for tests to be used over multiple releases of the same app;
- Regarding the *graphical mutations*, we have measured the frequency of occurrence of different types of graphical changes on the widgets in a frequency range between 4% and 57% of oracles. Therefore, visual locators cannot be depended if used alone to locate widgets.

In summary, the high frequency of changes in both widget position (bounds), layout attributes and visual aspect denotes how coordinate-based, layout-based, and image-based localization strategies may be unsuitable for robust GUI testing of mobile applications on the run of multiple releases, when used alone and without the combination of multiple attributes

This general picture of Android apps highlights the necessity of shifting from a single-attribute (or single-technique) localization process to a multi-locator and multi-technique approach, mixing different attributes and methodologies together, as proposed for GUI testing in web applications [3].

A. Threats to Validity

As an *External Validity* threat to our study, we identify the possibility that the selected sample of SUTs may not be representative of the entire population of Android apps. To mitigate this threat, we have based our app selection on an available categorization and we have selected a single item for each category.

As *Construct Validity* threat to our study, we identify the possibility that the procedure applied for the collection of the version pairs of the SUTs (section III.B) may privilege the selection of older versions, therefore raising the probability of having attribute and visual changes between corresponding widget pairs. To analyze the effect of release selection on attribute instability, we plan to replicate the study by performing longitudinal analyses over the entire release histories of the considered SUTs.

VI. Conclusion and Future Work

In this paper, we assessed the state of the practice of Android apps by selecting 30 apps from the most downloaded ranking in order to perform an analysis of widgets composition. We provided statistics concerning attribute values. In particular, it appears that textual attributes are those with the highest percentage of empty values: more than a quarter of the widgets have no *resource-id* defined, only one in four widgets has a non-empty *text* attribute. Although their presence is not guaranteed, the textual properties, in addition to the *bounds* attribute, are those whose values exhibit the greatest variability.

When considering changes affecting widgets, starting from a set of 201 corresponding widgets, we noted that the most variable attributes are *bounds* and *resource-id*.

The present work is to be considered as a preliminary study for a broader research topic aiming at defining a multi-locator and multi-technique testing strategy for the mobile domain.

We also plan to apply machine learning-based techniques on attribute value distributions, to automatically infer the most suitable combination of attributes for a proper widget localization, by considering widget identification in GUI testing as a classification problem.

References

[1] Statista Research Dept., "Revenue of mobile apps worldwide 2017-2025, by segment," 2022, accessed: 2022-12-15. [Online]. Available: https://www.statista.com/forecasts/1262892/mobile-app-revenue-worldwide-by-segment

[2] J. Flynn, "40 fascinating mobile app industry statistics [2022]: The success of mobile apps in the u.s." 2022, accessed: 2022-12-15. [Online]. Available: https://www.zippia.com/advice/mobile-app-industry-statistics/

[3] M. Nass, E. Alégroth, R. Feldt, M. Leotta, and F. Ricca, "Similarity-based web element localization for robust test automation," 2022. [Online]. Available: https://arxiv.org/abs/2208.00677

[4] A. Developers, "Why adopt compose," 2022, accessed: 2022-01-12. [Online]. Available: https://developer.android.com/jetpack/compose/why-adopt#less-code

[5] R. Coppola, M. Morisio, and M. Torchiano, "Scripted gui testing of android apps: A study on diffusion, evolution and fragility," in *Proceedings of the 13th International Conference on Predictive Models and Data Analytics in Software Engineering*, ser. PROMISE. New York, NY, USA: Association for Computing Machinery, 2017, p. 22–32. [Online]. Available: https://doi.org/10.1145/3127005.3127008

[6] P. S. Kochhar, F. Thung, N. Nagappan, T. Zimmermann, and D. Lo, "Understanding the test automation culture of app developers," in *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*, 2015, pp. 1–10.

[7] M. Leotta, D. Clerissi, F. Ricca, and P. Tonella, "Chapter five - approaches and tools for automated end-to-end web testing," ser. Advances in Computers, A. Memon, Ed. Elsevier, 2016, vol. 101, pp. 193–237. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0065245815000686

[8] L. Ardito, R. Coppola, M. Morisio, and M. Torchiano, "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*, ser. EASE '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 13–22. [Online]. Available: https://doi.org/10.1145/3319008.3319022

[9] M. Linares-Vásquez, K. Moran, and D. Poshyvanyk, "Continuous, evolutionary and large-scale: A new perspective for automated mobile app testing," in *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2017, pp. 399–410.

[10] D. Zelenchuk, *The Screen Object Design Pattern in Android UI Tests*. Berkeley, CA: Apress, 2019, pp. 231–244.

[11] M. Leotta, D. Clerissi, F. Ricca, and C. Spadaro, "Improving test suites maintainability with the page object pattern: An industrial case study," in *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops*, 2013, pp. 108–113.

[12] M. Nass, E. Alégroth, and R. Feldt, "Why many challenges with gui test automation (will) remain," *Information and Software Technology*, vol. 138, p. 106625, 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0950584921000963

[13] E. Alegroth, Z. Gao, R. Oliveira, and A. Memon, "Conceptualization and evaluation of component-based testing unified with visual gui testing: An empirical study," in *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*, 2015, pp. 1–10.

[14] Android Open Source Project, "App resources overview," 2022, accessed: 2022-01-25. [Online]. Available: https://developer.android.com/guide/topics/resources/providing-resources