A Novel Tool for Automatic GUI Layout Testing

Kristian Fjeld Hasselknippe
Department of Computer Science
Norwegian University of Science and Technology
Trondheim, Norway
kristian_hasselknippe@outlook.com

Jingyue Li
Department of Computer Science
Norwegian University of Science and Technology
Trondheim, Norway
jingyue.li@ntnu.no

*Abstract***—As mobile apps are expected to run in many different screen sizes, the need to validate the correct positioning and rendering of graphical user interface (GUI) elements in such screens is increasing. In this paper, we first focus on identifying typical layout errors in modern GUIs and categorising them. Then, we implement a tool called Layout Bug Hunter (LBH) to automatically identify if a GUI layout is rendered correctly in different screen sizes. The tool is evaluated on mobile apps and is compared to state-of-the-art layout-testing tools. Results show that LBH can identify all the typical layout errors we categorise, and LBH is more accurate than a layout-testing tool based on image diffing algorithms. In addition, LBH does not require writing layout test scripts manually. LBH is currently implemented on only one mobile app development platform, but it is designed to be portable and extensible. With only limited effort, LBH can be extended to other mobile and web development platforms for layout testing.**

Keywords—GUI; software testing; software tools

I. Introduction

A GUI usually consists of many elements, which visually represent either the data of the application or the actions which can be performed by the user. GUI layout engines lay out these elements to make them fit well on the screen and make them fully visible. To access the Internet, we now have many types of devices, such as laptops, tablets, netbooks, and smartphones. The devices have different screen sizes and screen resolutions [1]. Developers and GUI designers should make sure that their websites or mobile apps are compatible with all the possible screen sizes and resolutions. To verify whether a GUI is compatible with different screen sizes and resolutions, some tools such as Screenfly [2] and ResizeMyBrowser [2] let developers manually resize their browser window or app window to many preset screen sizes to run visual inspections. If there are *m* web pages or app windows and *n* preset screen sizes, developers need to perform *m* * *n* manual inspections. Some other tools, for example, Galen Framework [3] and ITArray's Automotion framework [4], automate the layout verification on different screen sizes and resolutions. However, developers need to write scripts to describe how various elements of a GUI relate to each other and what kind of layout parameters they will consider valid. Then, the scripts will generate test cases and regression test cases to verify GUI layouts. If there are *m* web pages or app windows and *n* preset screen sizes, developers need to manually write and maintain *m* scripts. Based on image diffing algorithms, tools such as AppliTools [5] and PhantomCSS [6] can potentially be used for GUI layout verification. AppliTools can detect whether a screenshot is significantly different from another one. However, the tools based on image diffing algorithms usually do not work well if the images under comparisons are identical but are presented in different screen sizes and resolutions.

To improve cost-effectiveness and precision of GUI layout testing, we developed a tool called Layout Bug Hunter (LBH). We started with identifying and categorising several typical layout errors, such as overlap, overflow, and lost alignment. For each web page or window of a mobile app, LBH first automatically collects the layout information. Then, LBH performs rule-based and statistical analysis of the layout information to check whether the typical layout errors will happen if the web page or the app window is presented in different screen sizes and resolutions. If any layout errors happen, LBH will generate reports to show error details.

To evaluate LBH, we first developed a small mobile app. Then, we manually inserted several typical GUI layout errors in the app and ran LBH to see if it could find out all the inserted errors. Results show that LBH can precisely find all the inserted GUI layout errors. To compare LBH with AppliTools, we first made one GUI of a mobile app and then resized the GUI into different screen sizes and resolutions. Then, we used LBH and AppliTools to identify if there were GUI layout errors after resizing. Results show that AppliTools reports much more false positives than LBH. We did not compare LBH with Galen Framework or ITArray's Automotion framework using case studies. The difference between LBH and Galen Framework or ITArray's Automotion framework is that LBH does not require developers' effort on writing and maintaining scripts for each web page or app window.

The rest of this paper is organised as follows. Section 2 introduces the state of the art of GUI testing and GUI layout testing. Section 3 presents our classification and categorisation of typical GUI layout errors. Section 4 describes the design and implementation of LBH. Section 5 presents our experimental evaluation. Section 6 discusses evaluation results, and section 7 concludes.

II. State of the Art

GUI testing can generally be classified into two categories, namely, function tests and layout tests.

A. GUI function tests

The goal of GUI function tests is to make sure that all functionalities of an application are in place and that they do their jobs correctly. Traditionally, GUI function tests were performed by dedicated GUI testers, who would run all functions of the application and verify their correctness. Today, several methods and tools, for example, capture and replay (e.g., [7]) and model-based GUI testing (e.g., [8]), have been developed to help application developers automate GUI function tests.

B. GUI layout tests

GUI layout tests are concerned with validating the visual aspects of the GUI. The focus of GUI layout tests is to verify that each GUI element and all possible data the GUI is supposed to present is laid out and rendered correctly on all target platforms and devices.

When we start making GUIs, we need to describe the layout of its various elements, meaning their sizes and places on the screen. The most basic solution is to just use absolute positioning, which means that the developer specifies a 'X' and a 'Y' position as well as a 'width' and a 'height' for each visual element. This is a viable solution, if we know the aspect ratio and size of the window or screen that we are targeting and if we know exactly which elements we need to place. However, such absolute positioning will not work with dynamic data because we do not know how many items we need to place or how large they are required to be before the app runs.

There are two main dynamic GUI layout approaches used by modern layout engines, that is, hierarchical layouts and constraint-based layouts.

1) Hierarchical layouts: Generally, the hierarchical layout approach (e.g., WPF [9], HTML [10], Android UI [11], and Fuse [12]) uses a hierarchy of layout containers that subdivide the available space of a screen. The containers are in charge of placing all their children within a proper space according to defined rules. A markup language is typically used to describe the hierarchies. In this paper, we use Fuse [12] as an example to explain hierarchical layouts. Fuse is a cross-platform mobile app development tool that allows developers to create custom layouts and animations that look exactly the same on iOS, Android, and Windows. We have been working extensively with Fuse, both on its inner workings and as a development tool. Thus, we think Fuse is a proper platform to be used as a base for implementing the ideas proposed in this paper. Layouts in Fuse are hierarchies of panels. A panel is an invisible container that has multiple child panels, child elements, and an associated layout type. Fuse defines a set of panel types, such as StackPanel, DockPanel, and Grid. All panel types can contain other panels and elements which represent visual controls, such as buttons, sliders, and shapes. Fuse uses a markup language called UX, which is based on the XML family of markup languages. Figure 1 shows an example UX code snippet that creates a GUI using Grid. Grid contains four elements: a red circle, some text, a button, and a slider control. The four elements are placed in a two-row, two-column grid.

2) Constraint-based layouts: In constraint-based layout solutions, for example, iOS' Auto Layout [13], layouts are described using a set of linear constraints. A hierarchical structure is also used here, but it is used more for organisational purposes than for dictating the position and size of elements. Constraint-based layouts give the designer a lot of flexibility but can be quite cumbersome to write by hand. Visual tooling is often needed to apply constraint-based layout techniques. For example, a tool (called 'Interface Builder') that comes with XCode allows the designer to visually edit layouts instead of having to write constraints using a markup language. The textual output of these visual tools is also less human friendly than their hierarchy-based counterparts.

```
<App>
    <Grid RowCount="2" ColumnCount="2">
        <Circle Color="Red"/>
        <Text>Some text</Text>
        <Button Text="Button text"/>
        <Slider />
    </Grid>
</App>
```

Fig. 1. A GUI UX code snippet using Grid

GUI layout-testing tools can generally be classified into three categories.

1) Visual inspection based: One category of tools, for instance, Screenfly [2] and ResizeMyBrowser [2], facilitate visual inspections of GUI errors. Users of these tools need to manually load a web page or mobile app GUI into different screen sizes or resolutions and visually inspect the validity of the GUI.

2) Layout analysis based: Another category of tools, for example, Galen Framework [3] and ITArray's Automotion framework [4], provides testing libraries for testing GUI layouts. These tools let users create GUI layout descriptions that describe relationship and layout validity of elements of a GUI. Executable test cases will be automatically generated from these descriptions to verify validity of the placement and size of GUI elements.

3) Image diffing based: The third category of tools, for example, AppliTools [5], PhantomCSS [6], Webdriver [14], and text-colour change detector [15], check GUI validity by comparing screenshots pixel by pixel. A screenshot of a valid GUI needs to be taken first. By detecting the difference between two screenshots, these tools can find out if presentations of two web pages or two app windows are significantly different. If the differences are significant, the tool will report the differences as errors.

III. Our Classification of Typical GUI Layout Errors

When creating GUI layouts for mobile devices, the layouts need to be valid for a wide variety of different screen sizes and resolutions. It can be difficult to validate that a set of layout elements do not overlap with each other when the screen becomes smaller than originally expected (this happens often because the GUI was never designed with all targeted screen sizes in mind) or when a user's input is larger than originally assumed. There are many kinds of layout errors. Based on our experience of creating app GUIs, we have classified typical GUI layout errors into three main categories, namely, overflow, overlap, and lost alignment. Although our current classification may not cover all kinds of GUI layout errors, within such a classification framework, we can continue to collect more typical GUI layout errors and extend our classification.

A. *Overflow*

Overflow happens when the GUI container, for example, a text item, ends up being too small for its content. Overflow can happen horizontally or vertically.

B. *Overlap*

Overlap happens when two elements, who are siblings (meaning that they belong to the same container or parent element), are drawn partially over each other. Some controls have an intrinsic minimum size to maintain usability. This includes controls, like buttons, sliders, and switches, which should have a minimum size to be easily hit. If a designer puts too many controls together horizontally or vertically, when the screen is smaller than the screen used for the initial design, the controls may overlap with each other. Some overlap may be difficult to spot even by manual visual inspection. For example, a hit box is an invisible shape (usually a rectangle), which is used by controls to define the area on screen through which the controls are hittable. In a GUI, three buttons spread out horizontally might be too many for a smaller screen. We can end up with overlapping on the hit boxes even though the buttons do not seem to overlap visually. This can severely decrease usability. Figure 2 shows an example of buttons whose hit boxes are larger than their visual appearances. In this example, the buttons are placed so close to each other that their hit boxes overlap.

C. *Lost alignment*

Elements in a layout are often aligned to each other in one or several ways. This alignment is usually intentionally added by the developer using layout containers, for instance, Grids or StackPanels. Although we can expect the children of a Grid to be aligned to each other (as this is the main function of a Grid), we cannot easily verify alignment when elements are not direct siblings of each other. It is possible that each grid item has its children, which should be aligned to the children of the other grid elements. This is, however, a common limitation of many hierarchical layout systems. Thus, we can easily end up with a situation where layout elements lose alignments because of changes in screen size or data. In some other cases, the alignment was not explicitly declared by the developers' choice of layout containers but emerged because of the correct choice of margins, paddings, and sizes. In these cases, the alignment is still deliberated but can more easily be broken because of unforeseen data sizes or changes in screen sizes or aspect ratios.

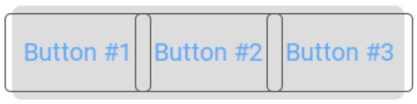

Fig. 2. Example of hit-box overlapping. The grey-stroked rectangles are not a part of the app. It is here to visualise where the user can tap to hit the button. The figure shows how we can have overlapping hit boxes without the buttons being overlapped.

D. *Exceptions of GUI layout errors*

GUI designers sometimes use overlap for aesthetic purposes. Such kinds of overlaps should not be regarded as GUI layout errors.

IV. Design and Implementation of LBH

To identify the overflow, overlap, and lost-alignment layout errors, we developed a tool called Layout Bug Hunter (LBH). LBH processes the GUI information through several phases. Each phase performs a set of tasks. Results of a phase is fed into the next phase.

A. *Phase 1 - Data collection*

The purpose of this phase is to collect two kinds of information. One kind of information is a list of GUI elements and sizes and positions of the elements. Another kind of information is alignment information of the various elements in the GUI. To collect alignment information, we collect 'tab stops' and which elements the 'tab stops' are connected to. 'Tab stop' is a term borrowed from linear programming–based GUI frameworks [16]. A 'tab stop' is a vertical or horizontal line that divides the screen in two half-spaces. This line is used by linear programming techniques to align GUI elements to create all types of layouts, like grids, stacks, and docks. In LBH, each element with a width and height larger than 0 is aligned to 4 different 'tab stops'. Figure 3 shows an app screen with all 'tab stops' marked with striped lines.

To collect the data, we developed a crawler component. The crawler collects data by running all test cases. Running a test case here is to render the GUI in a screen of certain size or resolution. The crawler navigates the hierarchy of GUI elements and collects position and size information of the elements. The data collected by the client is converted to JSON format and is represented 'in-line' as a hierarchy of nodes where each node has a list of child nodes. So far, we have developed only the crawler for Fuse, which uses hierarchical GUI layout approach. To apply LBH to validate GUI using other hierarchical layout approaches, such as WPF, HTML, or Android UI, we need to develop a new crawler for each of them. LBH is designed using client-server architecture. The data collection (i.e., the crawler) runs at the client side. The collected data are then sent to the server side to be processed by Phases 2 to 5 of LBH.

B. *Phase 2 - Layout validation*

We use the information collected from Phase 1 to detect layout error candidates of various types. Each error type has specific assumptions and algorithms associated with it. Results of this phase are a set of layout error candidates containing information about the violating elements and how the error is manifested. The validation is rule-based. Various rules are applied to the data gathered from Phase 1.

The overflow validator checks whether GUI elements are drawn within the bounds of their parent. The assumption here is that a GUI element, which has fixed size and is inside a dynamically sized container, can easily lead to visual glitches when it is tested on a smaller screen or on a screen which has a different aspect with the one used by the designer. The overflow validator first extracts size and position information of all pairs of elements that have a parent-child relationship. Then, the validator checks whether the child is within the bounds of its parent by comparing the rectangles represented by the two elements. If the child is outside the bounds of its parent, an overflow error candidate will be generated.

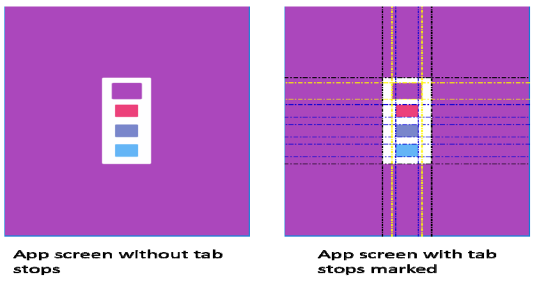
App screen without tab stops
App screen with tab stops marked

Fig. 3. All tab stops for an example app screen.

The overlap validator checks whether two GUI elements are partially drawn on top of each other. The overlap validator focuses only on nodes that have a sibling relationship. The overlap validator first extracts size and position information of all unique pairs of elements which have the same parent. Then, the validator checks whether one element is within the bounds of another by comparing the rectangles represented by the two elements. If one element is within bounds of another, an overlap error candidate will be generated.

The lost-alignment validator works under the assumption that elements that belong to the same 'tab stop' do so deliberately. This rule applies for GUI elements arranged in a Grid or in a StackPanel. The rule is also valid for GUI components who end up in alignment independently from the layout structure they belong to. In Phase 1, we have collected information of all 'tab stops'. In addition, for each GUI element, we link each of its four sides (top, bottom, left, and right) to a certain 'tab stop'. For example, for a certain element in a certain test case, we know that the element is align to, for example, 'tab stop A' at the left side and 'tab stop B' at the right side. Such relationship information between 'tab stops' and GUI elements makes it possible for us to identify whether two GUI elements are aligned in different test cases. If in one test case one element links to the same 'tab stop' as a certain side (e.g., left side) with another element, then we know that the two elements are aligned as the left side in this test case. The lost-alignment validator first calculates 'in which test case, an element is linked to which "tab stop" at which side'. Then, the lost-alignment validator checks if an element is linked to different 'tab stops' in different test cases. If different 'tab stops' are linked to the same element in a different test case, a lost-alignment error candidate will be generated.

C. *Phase 3 - Error sorting*

Here, the error candidates identified in Phase 2 are sorted based on their types and severities. Different severity calculations are performed for different error types. The overflow error sorting is based on the total overflow areas. The total overflow areas are calculated by adding together the overflow rectangle area at four directions, that is, top, bottom, left, and right. The larger the total area, the higher the severity rating assigned to the overflow error. The area of the intersection rectangle is used to determine the severity of overlap errors. The larger the area, the higher the rating. The difference in position between 'tab stops' is used to determine the severity of lost-alignment error. The bigger the difference in position, the higher the rating. If two candidates have the same amount of position difference but different orientation, vertical orientations are given a higher severity rating than the horizontal one. The reason for ranking vertical alignment high is that most apps have a natural vertical flow, which makes vertical alignment changes more noticeable than horizontal alignment changes.

D. *Phase 4 - Error filtering*

Here, we filter out duplicated layout error candidates reported in Phase 3. An example of duplicated error candidate is that for each overlap, two errors are reported: one from the perspective of the first element and another one from the perspective of the second element. With respect to lost-alignment error candidates, we merge all errors that are reported for the same pair of elements, and maintain a list of pairs of test cases at which these errors occur and the number of times these errors occur. Then, we filter out false-positive layout error candidates. Many other false-positive layout error candidates are related to certain edge cases. An edge case is a special layout made by the designer deliberately (e.g., for aesthetic purposes), but the layout looks like an error. For different error types, we apply different error-filtering algorithms for filtering out different edge cases.

For overflow, we have identified two types of possible edge cases. One type of overflow edge case is overflowing for aesthetic purposes. Many applications use drop shadow and blur for aesthetic purposes. When we validate using the rendering size (the size of the bounds that are used for drawing), a drop shadow or blur effect can cause the element to report a size that is larger than the available size given by its parent. Another type of an edge case of overflowing is intentional clipping. In some cases, a container is used not only as a means of performing layout on its children but also as a means to intentionally perform clipping to hide certain parts of the element. This happens very often when using scroll views.

Scroll view allows users to interact with it to move the content around. Thus, content contained in the view can be bigger than the view itself. To eliminate false-positive overflows, we do statistical analysis of the error candidate and follow the 'majority wins' philosophy. We establish a baseline first. A baseline is a subset of candidate errors received from Phase 2 and Phase 3. If an error presents in a certain amount (usually a majority) of the test cases, it will be included in the baseline. If an error is included in the baseline, it may indicate intentional design, and it is probably related to a certain type of edge case. The rationale behind this is that we believe layout errors due to change of screen size and resolution happen but not frequently. If the same layout error happens in many test cases, it is most likely that the error (i.e., overflow here) is not a true error but is added intentionally by the designer. Therefore, those error candidates included in the baseline are mostly likely to be false positive. To establish the baseline, we compare data collected from all test cases, that is, rendering the GUI in screens of all different sizes and resolutions we test. If we observe that a certain overflow error candidate appears in all or majority test cases, we consider the error candidate to be an edge case, and we will add the error candidate to the baseline. Once the baseline is established, we filter error candidates in the baseline and keep only those that are not in the baseline for processing in the next phase.

For overlap, we have identified two types of possible edge cases. One type of edge case of overlap is complete containment. If one element is completely inside another element, we consider it to be an intentional design choice. This is because a common design pattern is to make a background fill for elements. However, if the intersection rectangle is smaller than the smallest of the two elements, we know that we do not have a case of complete containment. Thus, the intersection should be a layout error. Another type of edge case is overlapping for aesthetic purposes. Overlapping sibling elements is quite commonly used in app designs and thus is a big potential source for false positives. To eliminate false positives in overlap, we apply a similar strategy as that in overflow. We establish baseline first and then filter out all error candidates that are included in the baseline.

To filter out false-positive lost-alignment errors, we also use the 'majority wins' policy. Based on how many times the node sides are aligned relative to the total number of test cases, we decide whether to put the error into the baseline or not. For example, if we have two elements which are aligned in 80% of the test cases, the 20% of cases in which the two elements are not aligned are probably real layout errors and therefore should not be included in the baseline.

To further decrease the risk of reporting false positives and false negatives, we allow users to configure certain parameters of LBH. We allow users to set a threshold for 'tab stop' merging. For complex GUIs, we may end up with a lot of tab stops which are almost identical but are only separated by one or a few pixels. By making the threshold larger, 'tab stops' will be merged, and the chance of elements being reported as having lost alignment will be lowered. We also allow users to set a threshold for defining when errors should be added to the baseline. By letting the user set this parameter, we give them control over how represented a certain error should be to be included into the baseline. One could, for example, decide to include an error into the baseline if the error presents in more than 80% of the test cases.

E. *Phase 5 - Error rendering*

Here, we generate a visual representation of detected errors and a textual report. We first present all GUI elements in a lighter colour so that the error nodes stand out more. Overflow and overlap violations are highlighted as dark-coloured rectangles. Lost-alignment violations are highlighted by drawing their common 'tab stop' with a black line and by drawing a line between the two involved elements. The involved elements are also highlighted with darker colours. A textual report is generated that includes all the errors identified by LBH. The report starts with a summary of the number of errors of each type and shows the number of test cases used. Then, the report presents a detailed list of all errors. Each error type has its own format, which contains information about which elements were involved as well as which lines the error corresponds to in the source code of the GUI. For overflow and overlap error, the textual reports include the test case the error occurred in and the two elements involved in the error. Lost-alignment errors not only have a similar structure but also include information about which side of the element is not aligned in how many test cases.

V. Empirical Evaluation of LBH

To evaluate LBH, we focus on answering three questions.

A. *Can LBH detect all the errors we have classified?*

We first made an app and inserted several errors such as overflow, overlap, and lost alignment. We then ran LBH on the app to see if it could detect all the inserted errors correctly. The focus of this evaluation is to verify the correctness of Phase 2, that is, layout validation of LBH. During this evaluation, we did not configure LBH to establish a baseline because we treat all errors here as true errors. If we apply baseline here, some of the inserted errors (which were present on all test cases) may be regarded as special graphic designs and therefore be filtered. Results show that LBH detects all the errors correctly.

B. *Can LBH detect GUI layout errors automatically?*

We made a small *to do* list mobile app using Fuse to evaluate whether LBH can automatically detect all the GUI layout errors while developing the app. We deployed the app into seven mobile devices with different screen sizes. Results of this evaluation show that if overflow or overlap happens in any of the experimented mobile devices while developing the app, LBH reports the errors correctly. If the errors are fixed, LBH can correctly remove the error from its report.

C. *Can LBH outperform existing tools?*

We compared LBH with AppliTools [5] by presenting a screenshot in different screen sizes and by letting LBH and AppliTool to detect layout errors caused by changing the screen sizes. We chose to compare LBH with AppliTool because AppliTool is supposed to be as cost-effective as LBH. AppliTool does not require human visual inspection (e.g., Screenfly [2]) or writing scripts (e.g., Galen Framework [3]).

We used a small single-page app GUI to compare LBH and AppliTools. The app was implemented in Fuse and contained one case of overflow. We compared AppliTools and LBH by testing the app on two screen sizes. AppliTools requires the compared screenshots to have identical sizes. This means that we must scale one of the images either down or up. To test the scale-down case, we used the original screenshot of the app running on an iPhone 5/5S screen size, and then we downscaled the iPhone 6 Plus version so that it fits with the iPhone 5/5S size. To test the scale-up, we used the original iPhone 6 Plus screenshot and enlarged the iPhone 5 screenshot to match. AppliTools reports errors by highlighting the areas of the compared screenshot with a purple colour. AppliTools marks almost all the elements of the app as layout errors. The difference between the two screen sizes causes the icons and text to become bigger relative to the screen size. As AppliTools works on image comparison pixel by pixel, it sees all these changes as errors. We tested LBH on the same app using the same screen sizes. LBH correctly pointed out the overflow error inserted without false positives. However, LBH reported 31 lost-alignment candidates; it reported so many error candidates because it cannot establish a good baseline of lost-alignment errors with only two test cases. When we increased the number of test cases to 4, LBH reported only 2 lost alignment errors, which could be confirmed as true errors by our visual inspections.

VI. Discussions

A. Configuration of the baseline parameter

When using LBH, we can configure the parameter to define the percentage an error should present in test cases to be included into the baseline. We conducted experiments to see the variations in layout errors reported when changing the value of this parameter. Results show that setting the parameter at 0.8 will give the most optimal results.

B. Comparison with existing tools

When LBH is compared with tools relying on human visual inspections, LBH is more automated and requires fewer human intervention. It can also be more accurate because humans can miss or overlook GUI layout errors which are not obvious. Tools based on image diffing algorithms rely on information stored in the pixels of the screenshots and have no knowledge about where in the code the errors originated from. LBH combines information in the UX code and the images. Thus, LBH can point out the possible origin of image errors in the UX code. When compared with Galen Framework, LBH does not require manually writing any test specifications.

C. Threats to validity of evaluation

1) Internal validity threats. GUI layout is a relative, subjective manner. In our evaluations, we made our own definitions of correct and incorrect layouts in the example apps we used. Our definitions of layout errors can be different from those of others. For example, a small alignment loss may not be regarded as an error by some designers.

2) External validity threats. We evaluated LBH using only several small apps. The shapes of elements included in the GUI of the apps are mostly rectangles. More evaluations are needed to verify LBH in complex apps with more types of shapes (e.g., circular, triangle, and so on).

D. Limitations of LBH

Currently, LBH can only test apps made with Fuse. However, it is not difficult to extend LBH to other platforms. We only need to implement a client component for each platform to gather layout information. Currently, LBH cannot perform its validation on animations. It means that some errors might be wrongly reported because LBH does not consider the fact that an element might be in the middle of an animation.

VII. Conclusions and Future work

With the growing variety of screen sizes and resolutions of mobile devices, the need to validate GUI layout correctness in different screen sizes is increasing. We have developed a tool to automatically detect GUI layout errors. The tool can be used as plugins of IDE and testing environments. Evaluations show that the tool is more accurate and is potentially more cost-effective than existing tools for similar purposes.

References

[1] Screen Size Comparison: DVD, iPhone, iPad, MacBook, Blu-ray, 2017, http://xahlee.info/comp/relative_screen_size.html.

[2] 9 Best Tools to Test Websites in Any Browser Sizes and Screen Resolutions, 2017, http://www.quertime.com/article/9-best-tools-to-test-websites-in-any-browser-sizes-and-screen-resolutions/.

[3] 'Galen Framework', Galen, 2017, http://galenframework.com/.

[4] 'Automotion', ITArray, 2017, https://www.itarray.net/ automotion/.

[5] 'AppliTools', AppliTools, 2017, https://applitools.com/.

[6] 'PhantomCSS', Huddle, 2017, https://github.com/Huddle/ PhantomCSS.

[7] O. E. Ariss, D. Xu, S. Dandey, B. Vender, P. McClean, and B. Slator, 'A Systematic Capture and Replay Strategy for Testing Complex GUI Based Java Applications', Proc. of 7th International Conference on Information Technology: New Generations, 2010, 1038–1043.

[8] A. Jaaskelainen, A. Kervinen, and M. Katara, 'Creating a Test Model Library for GUI Testing of Smartphone Applications', Proc. of the 8th International Conference on Quality Software, 2008, 276–282.

[9] 'Windows Presentation Foundation', 2017, https://msdn.microsoft.com/ en-us/library/ms754130%28v=vs.110%29.aspx.

[10] 'Hyper Text Markup Language', https://www.w3.org/html/.

[11] 'Android User Interface Overview', 2017, https://developer.android. com/guide/topics/ui/overview.html.

[12] 'Fuse - Cross Platform App Development Kit', 2017, https://www. fusetools.com.

[13] 'Understanding iOS AutoLayout', 2017, https://developer.apple.com/ library/content/documentation/UserExperience/Conceptual/ AutolayoutPG/.

[14] 'Webdriver', Webdriver, 2017, http://webdriver.io/.

[15] Michael Tamm, 'Fighting Layout Bugs', Google, 2009, https://www. youtube.com/watch?v=WY3C6FHqSqQ.

[16] 'Linear Programming Based GUI Framework', Apple, 2017, https://developer.apple.com/library/content/documentation/UserExperie nce/Conceptual/AutolayoutPG/index.html.