ISBN 978-981-11-3671-9
Proceedings of 2017 the 7th International Workshop on Computer Science and Engineering (WCSE 2017)
Beijing, *25-27 June*, *2017*, *pp. 544-548*

An Automated Detection Framework for Testing Visual GUI Layouts of Android Applications

Yen-An Shih, Yi-Ping Chang, and Cheng-Zen Yang[1]

Dept. of Comp. Sci. and Eng., Yuan Ze University, Taiwan, R.O.C.

Abstract. As Android is one of the most popular mobile operating systems, software quality maintenance of Android applications becomes an important issue. Although many studies have been conducted for detecting software bugs of Android apps, the chaotic layout problem appearing in different screen resolutions has not been discussed. In this paper, we propose a novel detection framework that employs two detection schemes for testing visual GUI layouts. The empirical experiments were conducted with three Android apps. The preliminary results show that the proposed detection framework can effectively discover all chaotic situations of the test apps.

Keywords: Android, Chaotic Layouts, GUI testing, Empirical Study.

1. Introduction

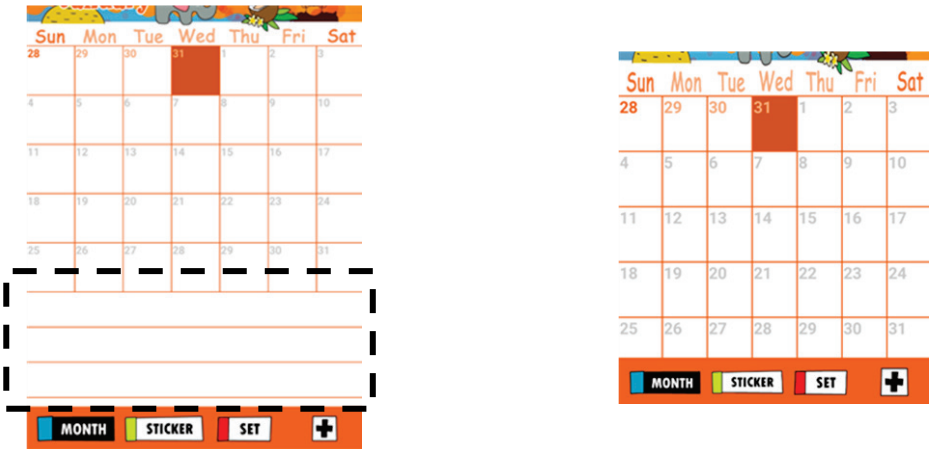
Android is one of the most popular mobile operating systems. As shown in the statistics of AppBrain, more than 2.7 million Android apps have been available in the market in February 2017 [1]. Maintaining the quality of Android applications becomes an important subsequent issue because AppBrain also indicates that 13% of these available Android applications are low quality apps [1]. With the increasing demand of software quality maintenance, various testing tools have been proposed for Android applications, e.g., [2,3,4]. However, most of these past testing tools mainly focus on the issue of discovering crash bugs or logical errors. The *chaotic layout* problem appearing in different screen resolutions has not been discussed.

The *chaotic layout* problem is resulted from the serious device diversity because Android can run on a large variety of devices that offer different screen sizes and densities. According to Android Developers API guides [5], Android developers should take notice of optimizing the UI design for different screen configurations. One example is to have different layouts for different screen densities, not to mention the portrait and landscape orientations. However, such *chaotic layout* problem can be still found in many apps on Google Play. For example, Fig. 1 shows two screenshots of an Android app, Girl's Note[2], in different resolutions. In Fig. 1(a), the app is correctly displayed in 768×1280 and 320 DPI (dots per inch). When the DPI is decreased to 240 in Fig. 1(b), the note component in the bottom area disappears.

The process of manually detecting the chaotic layout problem can be very cumbersome because its time complexity is $O(m \times n)$ for m layouts and n screen configurations. Assume that inspecting each layout needs 5 minutes. If an app has 10 layouts and it will run with 50 screen configurations, it may take 42 hours to inspect all GUI layouts.

In this paper, we present a novel detection framework for automatically testing visual GUI layouts of Android apps. The proposed detection framework deals with two chaotic layout situations: (1) layout components are missing in different screen configurations, and (2) cropped text appears in different screen

[1] Corresponding author. Tel.: +88634638800; fax: +88634638850
E-mail address: czyang@syslab.cse.yzu.edu.tw
[2] https://play.google.com/store/apps/details?id=com.apm.android.girlscalendarcht_hd



(a) 768×1280, 320 DPI. Correctly displayed. (b) 480×800, 240 DPI. The bottom note disappears.

Fig. 1: Screenshots of Girl's Note in different resolutions.

configurations. We have implemented the proposed framework based on the app exploration techniques used in PATS [4] that can automatically discover the GUI layouts of the app under test (AUT). The empirical experiments show that the proposed schemes can effectively find the chaotic layout situations in the AUTs.

The rest of this paper is organized as follows. Section 2 briefly reviews the past research on the visual effects of GUI layouts. In Section 3, the proposed detection schemes and the framework architecture are described. Section 4 presents the experimental details and the results. Finally, Section 5 concludes the paper.

2. Related Work

As the number of Android devices is rapidly growing, the chaotic layout problem of Android has not been comprehensively discussed. To the best of our knowledge, there is no systematic study discussing the chaotic layout problem for Android. In the past, most of the studies focus on the aesthetic arrangement issues of the layout design in a window-based environment. They do not consider the chaotic layout problem which is incurred because of the diversity of screen densities. For example, Ch'ng and Ngo propose a dynamic symmetry grid based approach for screen layout design [6]. The layout scheme utilizes the dynamic symmetry based on the rediscovery of Hambidge [7] and can reformats screen layouts automatically. However, this scheme does not consider the heterogeneity of various screen densities.

For the aesthetic arrangement issue, the computational model proposed by Bauerly and Liu calculates the aesthetic quantity of GUI elements in the screen layouts by considering three indicators: balance, symmetry, and number of groups [8]. They use these indicators to investigate the relationship between the arrangement of GUI elements and the aesthetic judgments. However, they do not study the chaotic layout problem for various screen configurations.

3. Detection Framework

This section describe the proposed detection framework that deals with two chaotic situations. The definitions of these two situations and the corresponding detection schemes are first elaborated. The detection framework architecture is thereafter presented.

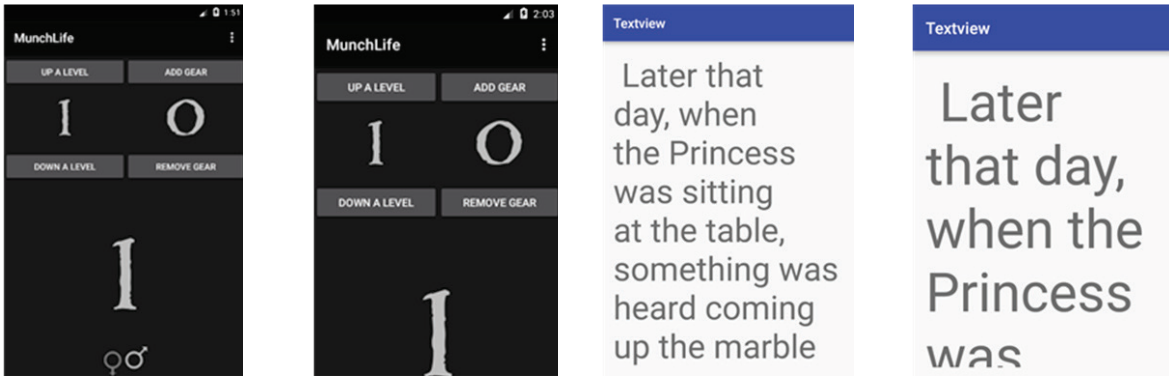
3.1. Problem definitions and detection schemes

Two chaotic situations for GUI layouts in different resolutions are studied in the proposed schemes:

- Layout components are missing in another screen density.
- Textual strings are cropped in another screen density.

Fig. 2 shows these two chaotic situations. Fig. 2(a) illustrates the first situation in which the bottom component of an app MunchLife[3] is correctly displayed in resolution 768×1280 and 320 DPI, but it is missing when the resolution is changed to 480×800 and 240 DPI. Even the worse, users cannot operate on

[3] https://play.google.com/store/apps/details?id=info.bpace.munchlife&hl=zh_TW



768×1280, 320 DPI 480×800, 240 DPI 768×1280, 320 DPI 480×800, 240 DPI

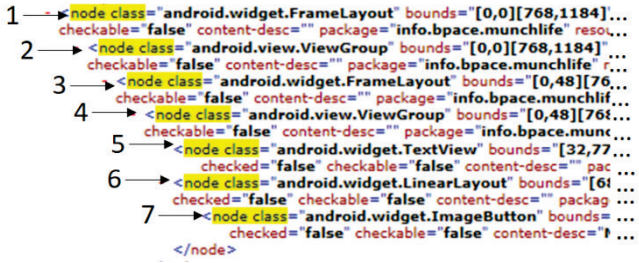
(a) Component missing. (b) Cropped text.

Fig. 2: The studied chaotic layout problem.

this component. In Fig. 2(b), a simple app Textview is designed to show that the text is cropped if the layout design is not adaptive to the change of the screen resolution. Therefore, it could be difficult for users to perceive the meaning of the text.

For these two kinds of situations, we propose two detection schemes, respectively. Before the detection process, a resolution that can correctly display the GUI layout is defined as the baseline resolution R_b. With the layout L_b in resolution R_b, the layout under test L_u in another resolution R_u is then compared to decide whether L_u has the chaotic layout problem.

For the first chaotic situation, the number of the GUI components in L_b is extracted as N_b. The number of the GUI components in L_u is also extracted as N_u. If $N_b \neq N_u$, this chaotic situation in L_u is detected. To achieve the extraction operation, we design a UI-Explorer framework to parse the Android XML structure of each GUI layout. Fig. 3 shows a snippet of the XML structure of MunchLife and the analytic results.



(a) A snippet of the XML structure of a layout.

```
1 FirstLayout.txt The number of standard layout components: 21
2 FirstLayout.txt The number of another screen resolution layout components:20
3 The number of layouts are different. It maybe has layout chaos: The reduction in
  the number of total componenets.
```

(b) The detection result of the chaotic situation.

Fig. 3: The component missing problem in MunchLife.

For the section chaotic situation, UI-Explorer first extracts the coordinate data of all text-based components in both L_b and L_u and then invokes a screen snapshot tool to get the individual snapshot of each corresponding text-based component in L_b and L_u. Finally, UI-Explorer invokes an OCR (Optical Character Recognition) tool to identify the text string of each component in different resolutions and decides their similarity to the original text string by calculating the edit distances. For a text-based component, if its edit distance in L_u is obviously different with its edit distance in L_b, the text-cropping situation may most likely occur in L_u.

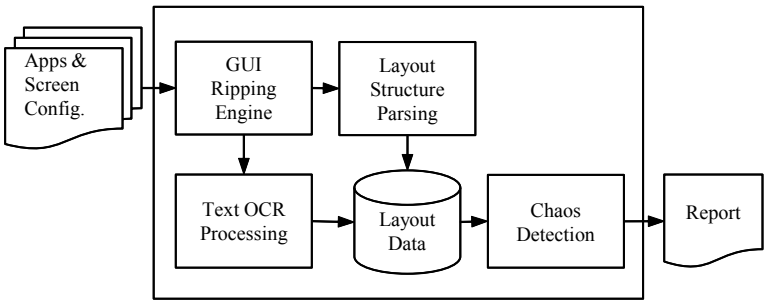
```
1 FirstLayout_com.example.yip.test:id_textView.txt
2  The standard text:
  Laterthatday,whenthePrincesswassittingatthetable,somethingwasheardcomingupthemarble,
  and  the standard edit distance：7
3
4 FirstLayout_com.example.yip.test:id_textView.txt
5  Another display text: LaterthatdaywhenthePfincess\AIQR, and the edit distance: 65
6 This layout maybe has layout chaos: cropped text on components.
```

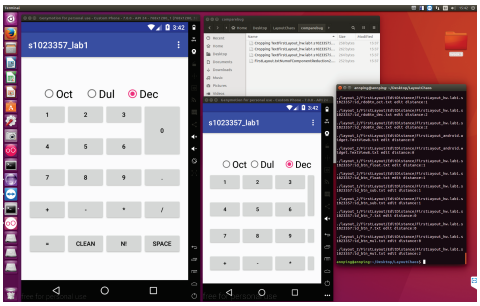
Fig. 4: The text cropping problem in Textview.

Take a text-based component c_i as an example. Its original text string $t_{o,i}$ can be extracted from the XML-based text attribute of c_i. In L_b, its OCR-identified string is $t_{b,i}$, and the corresponding edit distance between $t_{o,i}$ and $t_{b,i}$ is $ed_{b,i}$. Similarly, the OCR-identified string in L_u is $t_{u,i}$, and the edit distance between $t_{o,i}$ and $t_{u,i}$ is $ed_{u,i}$. Given a threshold T for considering the possible marginal OCR errors, text cropping may most likely occur if $| ed_{b,i} - ed_{u,i} | \geq T$. Fig. 4 shows the detection result for the app Textview.

3.2. Detection Framework Design

The proposed detection schemes are integrated in the UI-Explorer framework illustrated in Fig. 5. As shown in Fig. 5 (a), the developer inputs the AUT and screen configurations of L_b and L_u to UI-Explorer. UI-Explorer has a GUI ripping engine to automatically extract the XML information of the GUI components on all layouts. UI-Explorer then analyses the layout structure information and the OCR results for the final chaos detection process. Fig. 5 (b) is the screenshot for detecting a Postfix Calculator app.





UI-Explorer Framework

(a) The Framework (b) Working screenshot

Fig. 5: The UI-Explorer framework for detecting chaotic situations.

4. Experiments and Discussion

To study the effectiveness of the proposed detection schemes, we have implemented the UI-Explorer based on the exploration techniques used in PATS [4]. In addition, OpenCV [9] and Tesseract [10] were used as the snapshot engine and the OCR engine. We collected three apps for the empirical experiments:

- Textview: This app is custom-designed to have the text cropping problem.
- Postfix Calculator: This app is a student programming work. It is used to compute the postfix expression. It has both the component missing problem and the text cropping problem.
- MunchLife 1.4.4: This app is on Google Play for keeping track the character level for the card game Munchkin. It contains the component missing problem.

The empirical experiments were conducted for two resolutions. The resolution of 768×1280 and 320 DPI was used as the baseline resolution R_b. The resolution of 480×800 and 240 DPI was used as R_u. As shown in Table 1, the UI-Explorer framework effectively discovered the chaotic situations of these test apps as the manual detection. Although the experimental results are preliminary, they are promising for the future development of the detection schemes.

Considering the threats of the internal validity, the proposed schemes assume that the developers want to have the same layout in different resolutions. However, if the developers have different layout designs with regard to various resolutions, the proposed schemes will generate false alarms. However, the occurrences of

Table 1: Experimental results of three test apps

	Manual Detection		UI-Explorer Detection	
App Name	Component Missing	Cropped Text	Component Missing	Cropped Text
Textview		✓		✓
Postfix Calculator	✓	✓	✓	✓
MunchLife 1.4.4	✓		✓	

the false alarms should be uninfluential because the notification can be switched off. For the threats of external validity, the proposed schemes focus on XML-based layout design. If the AUT uses other graphical engines, such as OpenGL, for layout rendering, the proposed schemes cannot properly detect chaotic layouts. For this problem, other detection schemes for graphical engines are needed.

5. Conclusion

In this paper, we propose two novel detection schemes for testing visual GUI layouts of Android applications. The schemes deal with two chaotic layout situations: (1) layout components are missing in different screen configurations, and (2) cropped text appears in different screen configurations. The schemes have been implemented in a framework to help developers discover the potential GUI layout problems. The empirical experiments show that the proposed schemes can effectively find the chaotic layout situations. In the future, we will discuss more chaotic layout problems and design new detection schemes.

6. Acknowledgements

This work was supported in part by Ministry of Science and Technology, Taiwan under grants MOST 105-2815-C-155-037-E and Yuan Ze University under grant 105-HRD-02. The authors would also like to express their sincere thanks to anonymous reviewers for their precious comments.

7. References

[1] AppBrain. Android apps on Google Play. https://www.appbrain.com/stats/number-of-android-apps. Last acceded on February 10, 2017.

[2] D. Amalfitano, A. R. Fasolino, P. Tramontana, S. D. Carmine, and G. Imparato. A Toolset for GUI Testing of Android Applications. In: *Proc. of the 28th IEEE International Conference on Software Maintenance (ICSM 2012)*, pp. 650-653, 2012.

[3] B. N. Nguyen, B. Robbins, I. Banerjee, and A. Memon. GUITAR: An Innovative Tool for Automated Testing of GUI-driven Software. *Automated Software Engineering*. 2014, **21**(1): 65-105.

[4] H.-L. Wen, C.-H. Lin, T.-H. Hsieh, and C.-Z. Yang. PATS: A Parallel GUI Testing Framework for Android Applications. In: *Proc. of the 39th Annual International Computers, Software & Applications Conference (COMPSAC 2015)*, pp. 210-215, 2015.

[5] Android Developers. Supporting Multiple Screens. https://developer.android.com/guide/practices/screens_support.html. Last acceded on February 10, 2017.

[6] E. Ch'ng and D. C. L. Ngo. Screen Design: a Dynamic Symmetry Grid based Approach. *Displays*. 2003, **24**(3): 125-135.

[7] J. Hambidge. *Elements of Dynamic Symmetry*. Dover Publications Inc, New York, 1926.

[8] M. Bauerly and Y. Liu. Computational Modeling and Experimental Investigation of Effects of Compositional Elements on Interface and Design Aesthetics. *International Journal of Human-Computer Studies*. 2006, **64**(8): 670-682.

[9] Open Source Computer Vision (OpenCV). http//opencv.org. Last accessed on February 10, 2017.

[10] C. Patel, A. Patel, and D. Patel. Optical Character Recognition by Open source OCR Tool Tesseract: A Case Study. International Journal of Computer Applications. 2012, **55**(10): 50-56.