

Text Semantics and Layout Defects Detection in Android Apps Using Dynamic Execution and Screenshot Analysis

Šarūnas Packevičius^(✉), Dominykas Barisas, Andrej Ušaniov,
Evaldas Guogis, and Eduardas Bareiša

Software Engineering Department,
Kaunas University of Technology, Kaunas, Lithuania
sarunas.packevicius@ktu.edu

Abstract. The paper presents classification of the text defects. It provides a list of user interface text defects and the method based on static/dynamic code analysis for detecting defects in Android applications. This paper proposes a list of static analysis rules for detecting every defect and the tool model implementing those rules. The method and the tool are based on the application of multiple Android application emulators, execution of the application through certain execution paths on multiple hardware and software configurations while taking application screen-shots. The defects are identified by running analysis rules on each taken screen-shot and searching for defect patterns. The results are presented by testing sample Android application.

1 Introduction

The current Android applications market demands developers to build applications that can work on a wide array of mobile devices that have various hardware and software configurations. One of the common problems is a clipped text that does not fit on the applications window. This defect can become evident when an application is localized to other language and text messages become longer than it was originally anticipated. Those texts usually do get clipped and contain three-dot endings. Some of the other defects are: a text is too small to see or too large and it takes too much of the screen space on some devices.

The problems become evident when developers create applications on one primary development device and make its appearance consistent and refined. However, when the application is used on other devices (for example, on the ones having smaller screens) with a different configuration (for example, the application is internationalized to a Hungarian language, in contrast to the primary language it was developed, i.e. English), the text gets clipped and most of it is replaced by dots (...) as the text becomes twice longer than originally intended. To detect those defects, testers would have to test every application functionality in all application languages and on a wide array of device configurations.

2 Related Work

In the previous paper [1] we presented the initial method for detecting various types of user interface defects on mobile devices. The focus in this paper lies in text defect types, by further extending the text defects classification, proposing the text defect detection technique.

There are several works that address graphical user interface defects detection. They fall into two categories: detecting defects from the application images and detecting defects from the application source code.

The defects detection by analysing an application code falls into a category of static code analysis. However, these methods search for graphical user interface defects instead of coding errors [2]. Lelli et al. presented the method [3] for detecting defects by analysing an application source code and analysing event listeners.

Moran et al. presented the method [4] for detecting user interface defects by analysing application mock-ups and screenshot images. This method relies on having application mock-ups that are not always available or must perfectly match a developed application. Chang et al. presented the method for user interface testing using computer vision [5, 6]. The image recognition was used for detecting controls in an application interface and using them in automated testing scripts. Baek et al. presented the model based method [7] that detects user interface defects by comparing user interface to its model.

3 The Proposed Method

For the proposed method we combine two approaches for defects detection. We use application code and application screenshots to identify defects and pinpoint their location in situations where a user would fail. In addition, the application source code and corresponding assets are used as an application model that defines what kind of messages should be visible on the screen.

The method description is presented in further sections:

1. Defining defect classes – defect types are identified.
2. Rules set – a set of algorithms capable of detecting defects of each identified type.
3. Situation modelling – the method that must be automated by a tool, for generating situations where defects could arise.

The method is based on the principle of a source code static analysis: to gather data to analyse, run a set of rules for detecting each type of defect, and present findings.

The common static analysis tools analyse application source code. We propose to analyse the application screenshots in a similar manner. We define a set of defects and rules for detecting those defects in the application screenshots.

3.1 Classification of the User-Interface Text Defects

There are several defects classification methods such as presented in [8–10] that identify common defect types, such as data, interface, logic, function, and

documentation. The interface defect types are divided into user-interface and component-interface defect types. Lelli et al. [3] provided the scheme for classification of user-interface defects. Under the presented scheme defects fall into two categories: GUI Structure and Aesthetics, and Data Presentation. Based on the defect classifications [1, 4, 11] we concentrate on user-interface defects in the presentation category. Furthermore, we provide the list of defect types and methods for automatically detecting defects that are related to textual defect sub-types. Further on, we divide text defects into two categories:

1. Presentation – message presentation. The questions “is it visible”, “is it unobscured”, etc. are raised.
2. Semantics – defects that are related to a text message itself. This refers to an incorrect message, a misleading message, difficult to understand message, etc.

Presentation defects are related to applications text visibility on the screen. The main aspects are correct location, visibility, clashing with the background. The list is presented in Table 1.

Table 1. Text presentation defects.


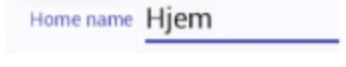

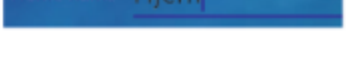
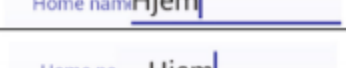



No	Type	Description	Example
TP1	Text placement	Text and surrounding controls are not aligned (text field and its label are not aligned vertically)	
TS1	Font sizes	The sizes of the text and surrounding controls mismatch (text size of the text field and its label are too different)	
TS2	Unreadable text	Text is too small to read on a device (with small physical screens and low resolution)	
TB1	Clashing background	Text colour clashes with a background image or colour, making the text difficult or impossible to read	
TC1	Partial text	Text is obscured by other controls or screen edges	
TC2	Clipped text	Text is too long to be presented in an allocated screen area, text is truncated by a user interface toolkit and is replaced by “...”	
TE1	Wrong encoding	Messages lack some characters, wrong characters are displayed, characters are replaced by question marks or a space character	
TM1	Missing text	Text is not visible	

Table 2. Text semantics defects.


No	Type	Description	Example
SD1	Synonyms	Different words are used to describe the same object	<i>The schedule</i> is assigned. <i>The week program</i> is not set.
SD2	Wrong terminology	Inconsistent terminology, different terms for the same concept, the same name for different concepts is used	<i>Loyalty card</i> is expired. (instead of a "gift card")
SD3	Unclear terminology	Ambiguous terms, terms for different concepts overlap in the meaning, too similar concepts, the same name used for different concepts	Heater is off. (meaning "electric one") Heater is on. (meaning "thermostat")
SD4	Jargon	Technical jargon, programmer jargon	Establishing connection to remote web socket.
SS1	Bad text style	Inconsistent writing style, misspelled words, incorrect grammar and/or punctuation	press und hold button for fivie sek chek red LED.
SS2	Too much text	Messages are too long to read	No program is assigned to this zone. To program this zone, you should either create a new program or select an existing program from the list below.
SS3	Vague error messages	Messages do not provide any meaningful information	Connection was reset by peer
SS4	Untactful messages	Offensive, impolite messages	Error!!! Invalid input fields. Enter correct data!!!
SS5	Misleading text	Erroneous messages	Device firmware does not support this method, please use the first method
SM1	Meaningless text	Text makes sense in an isolation but is meaningless in a GUI	
SU1	Long instructions	Instructions are too long, unstructured, not illustrated	Press and hold [WiFi] button for 15 seconds until red light starts blinking and then starts blinking very fast. Then click "Next". If LED only blinks slowly, your WiFi Smart Plug does not yet support WPS mode, please try AP registration option.
SU2	Too difficult to understand the text	Text writing style makes it difficult to understand	As many automated test input generation tools for Android need to instrument the system or the app, they cannot be used in some scenarios such as compatibility testing and malware analysis.
SU3	"..." misuse	The command with "..." should denote a link to additional window but it does not lead to it	Save... Exit...
SL1	Missing translations	Not all application messages are translated, translation placeholders are visible	error_message_no_wifi
SL2	Wrong language	The application shows a message in a wrong language, shows messages in mixed languages at once	You could reset device to factory defaults and try again. Trykk og hold nede [+], [-] and [OK] knapper på golvvarmestaten i 5 sekunder.

Table 3. Defect detection rules.

No	Defect Type	Rule source	Algorithms Input: <i>recognizedTexts</i> , recognized texts Input: <i>textAreas</i> , recognized texts areas Input: <i>actualTexts</i> , actual window texts in current language Input: <i>i18nTexts</i> , actual window texts in all app languages Input: <i>window</i> , analysed window Input: <i>device</i> , device screenshot was taken Input: <i>path</i> , execution path through app Input: <i>textPlaceHolders</i> , placeholders for translations Output: <i>defect</i> , detected defect
TP1	Text placement	S, W	foreach <i>area</i> \in <i>textAreas</i> do if <i>area</i> is not centred against surrounding controls then <i>defect</i> \leftarrow <i>window, device, area</i>
TS1	Font sizes	S, W	foreach <i>area</i> \in <i>textAreas</i> do <i>fontSize</i> \leftarrow <i>calculateTextFontSize(textArea)</i> if <i>fontSize</i> differ from surrounding controls text sizes then <i>defect</i> \leftarrow <i>window, device, area</i>
TS2	Unreadable text	S	foreach <i>text</i> \in <i>actualTexts</i> do if <i>recognizedTexts</i> not contains <i>text</i> then <i>defect</i> \leftarrow <i>window, device, text</i> foreach <i>area</i> \in <i>textAreas</i> do <i>fontHeight</i> \leftarrow <i>calculatePhysicalSize(textArea, device)</i> if <i>fontHeight</i> < 2mm then <i>defect</i> \leftarrow <i>window, device, area</i>
TB1	Clashing background	S, W	foreach <i>text</i> \in <i>actualTexts</i> do if <i>recognizedTexts</i> not contains <i>text</i> then <i>defect</i> \leftarrow <i>window, device, area</i> if $ text.color - window.backgroundColor < 5$ then <i>defect</i> \leftarrow <i>window, device, area</i>
TC1	Partial text	S, W	foreach <i>text</i> \in <i>actualTexts</i> do if <i>recognizedTexts</i> not contains <i>text</i> then foreach <i>recognizedText</i> \in <i>recognizedTexts</i> do if <i>text</i> contains substring <i>recognizedText</i> then <i>defect</i> \leftarrow <i>window, device, area</i>
TC2	Clipped text	S, W	foreach <i>text</i> \in <i>actualTexts</i> do if <i>recognizedTexts</i> not contains <i>text</i> then foreach <i>recognizedText</i> \in <i>recognizedTexts</i> do if <i>text</i> contains substring <i>recognizedText</i> and <i>recognizedText</i> ends with "..." then <i>defect</i> \leftarrow <i>window, device, area</i>
TE1	Wrong encoding	S, W	foreach <i>recognizedText</i> \in <i>recognizedTexts</i> do if not <i>spellCheck(recognizedText)</i> <i>text</i> then <i>defect</i> \leftarrow <i>window, device, area</i> if <i>recognizedText</i> contains "?" and not end or start on "?" then <i>defect</i> \leftarrow <i>window, device, area</i>
TM1	Missing text	S, W	foreach <i>text</i> \in <i>actualTexts</i> do if <i>recognizedTexts</i> not contains <i>text</i> then <i>defect</i> \leftarrow <i>window, device, area</i>
SD1	Synonyms usage	A, W	foreach <i>text</i> \in <i>actualTexts</i> do foreach <i>word</i> \in <i>text</i> do if <i>isNoun(word)</i> and <i>hasSynonym(word, actualTexts)</i> then <i>defect</i> \leftarrow <i>window, device, text</i>

SD3	Unclear terminology	A, W	$uniqueWords \leftarrow \text{extract distinct words from } actualTexts$ foreach $word \in uniqueWords$ do $translatedWord \leftarrow \text{word translated to another language}$ $placeholders1 \leftarrow \text{set of } textPlaceholders$ where $actualText$ contains $word$ $placeholders2 \leftarrow \text{set of } textPlaceholders$ where $il8nText$ contains $translatedWord$ if not $placeholders1$ matches $placeholders2$ then $defect \leftarrow \text{window, word, } placeholders1, placeholders2$
SD4	Jargon	A, W	foreach $text \in actualTexts$ do foreach $word \in text$ do if not $isStopWord(word)$ and $isNoun(word)$ and $isTechnical(word, actualTexts)$ then $defect \leftarrow \text{window, device, text}$
SS1	Bad text style	A, W	foreach $recognizedText \in recognizedTexts$ do if not $spellCheck(recognizedText)$ or not $grammarCheck(recognizedText)$ then $defect \leftarrow \text{window, device, } recognizedText$
SS2	Too much text	W, A	foreach $text \in actualTexts$ do $sentences \leftarrow \text{text split into sentences}$ foreach $sentence \in sentences$ do if $tooLongSentence(sentence)$ then $defect \leftarrow \text{window, sentence}$
SS3	Vague error messages	W, A	foreach $text \in actualTexts$ do $sentences \leftarrow \text{text split into sentences}$ foreach $sentence \in sentences$ do if $isVague(sentence)$ then $defect \leftarrow \text{window, sentence}$
SS4	Untactful messages	W, A	foreach $recognizedText \in recognizedTexts$ do if $isOffensive(recognizedText)$ then $defect \leftarrow \text{window, device, } recognizedText$
SS5	Misleading text	W, S	foreach $recognizedText \in recognizedTexts$ do if $isMisleading(recognizedText)$ then $defect \leftarrow \text{window, device, } recognizedText$
SU1	Long instructions	W, S, A	foreach $area \in textAreas$ do if $area$ takes all window space then $defect \leftarrow \text{window, device, area}$
SU2	Too hard to understand text	A	foreach $text \in actualTexts$ do if $readabilityIndex(text) \geq \text{CollegeLevel}$ then $defect \leftarrow \text{text, window}$
SU3	"..." misuse	W, A	foreach $text \in actualTexts$ do if $text$ ends with "..." and $text$ is on $link$ then if not $leadsSomeWhere(link, window, paths)$ then $defect \leftarrow \text{text, window}$
SL1	Missing translations	S	foreach $recognizedText \in recognizedTexts$ do if $textPlaceholders$ contains $recognizedText$ then $defect \leftarrow \text{window, device, } recognizedText$
SL2	Wrong language	S	foreach $recognizedText \in recognizedTexts$ do if $actualTexts$ not contains $recognizedText$ then if $actualTexts$ contains $il8nTexts$ then $defect \leftarrow \text{window, device, } recognizedText$

3.3 The Scheme for the Defect Detection

The main driving principle for detecting defects is to automate testing process. Here we propose a testing tool that executes the application under tests, gathers data, performs analysis and identifies possible defects. The main principle of the defect detection is structured as:

1. Generate tests that drive the application under test execution. The goal at this step is to generate tests that would cover 100% of the application under test code. There are several methods for generating tests [20–28] and tools [12, 29, 30].
2. Select a set of mobile devices to execute tests. The main selection criteria are to select as large as possible set of mobile devices with screens of varying physical sizes, resolutions, pixel densities, and colour depths. The devices could be emulated or real devices on a device cloud [31, 32].
3. Execute generated tests on each selected test device.
4. Gather screen-shot of each test step on each device.
5. Gather text messages using a live application window instrumentation of each test step on each device.
6. Run text recognition on each captured screen-shot and extract texts and their attributes:
 - a. font size,
 - b. position,
 - c. size.
7. Extract application texts from its source code and corresponding application assets.
8. Run defects detection algorithms on each captured screen-shot.
9. Run defects detection algorithms on all gathered text messages from the application.

The scheme for defect detection describing tests generation, screenshots generation, and text analysis is presented in Fig. 1.

The main components of the scheme are:

1. DroidBot:
 - a. Generates executable tests for navigating through the application under test.
 - b. Extension to DroidBot that extracts application messages from the source code and corresponding assets.
2. Text executor:
 - a. Runs all execution paths on all phones and emulators.
 - b. Takes screen-shot at each path step on each device.
 - c. Records device info, path step info, reference text.
3. Defect analyser:
 - a. Analyses gathered screenshots and executes all defects detection algorithms.
 - b. Provides a list of possible defects and their locations.

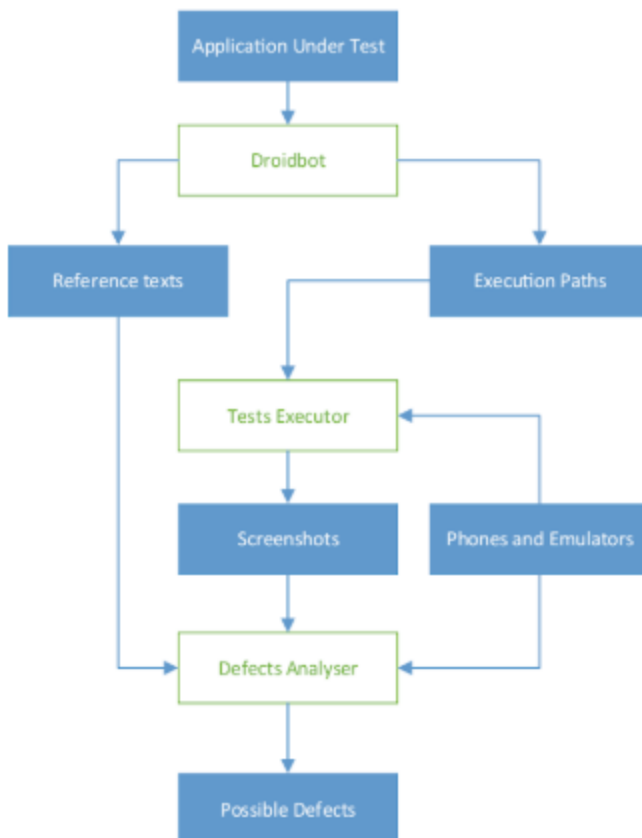


Fig. 1. User-interface defect detection scheme.

3.4 Testing Example

To present the proposed method of text defects several tools have been used. For test case generation the DroidBot tool has been applied.

For analysing captured screenshots custom tool has been developed: texts regions have been detected using the OpenCV [33] library by applying filters and afterwards the Tesseract [13] library has been used for extracting actual texts from screen-shots.

As a demo example the Android application with a multi-language interface has been selected. The Droidbot generated a set of execution paths and screenshots have been taken at every step. The created analysis tool has examined the captured screenshots and marked suspicious areas on them. Sample screen-shots are presented in Fig. 2.

The green bounds denote the detected text regions and the texts inside identify the discovered messages. Here, it is clear that the image A lacks several translations, while the image B has title and body messages in different languages (title is Norwegian, body is Czech languages).

The rules of text analysis were executed on each extracted message and its properties and defects were detected. The sample report is presented in Table 4. The tool detected a “too small text defect”. An internationalization problem was also detected. The message looked correct in the English language, but it was too long in Czech.





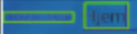


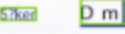
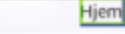

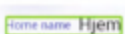
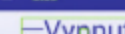
a)



b)

Fig. 2. Sample application defects.

Table 4. A report of the defects.

Defect	Device	Step/Window	Screenshot	Detected text	Expected Text
TS1	Jolla (EN)	Settings		Home name Hjem	Home name Hjem
TM1	Jolla (EN)	Settings		Hjem	Home name Hjem
TB1	Jolla (EN)	Settings		Home name Hjem	Home name Hjem
TC1	Xperia Ray	Settings		Hom name Hjem	Home name Hjem
TC2	Jolla (EN)	Settings		Home na... Hjem	Home name Hjem
TE1	Jolla (NO)	Settings		S?ker Hjem	Søker Hjem
TM1	Jolla (RU)	Settings		Hjem	Home name Hjem
SL1	Jolla (CZ)	Devices		Kapcsoló hozzáadása plug_product_na mes	Kapcsoló hozzáadása plug_product_na mes
TS2	Xperia Ray (2")	Settings		Home name Hjem	Home name Hjem
SL2	Jolla (CZ)	Control		Stue -Vypnuto-	Stue AV

28. Miguel, J.L.S., Takada, S.: Generating test cases for Android applications through GUI modeling, usage modeling, and change analysis. In: Proceedings of the Eighth International C* Conference on Computer Science and Software Engineering, pp. 146–147. ACM, Yokohama (2008)
29. Su, T.: FSMdroid: guided GUI testing of android apps. In: Proceedings of the 38th International Conference on Software Engineering Companion, pp. 689–691. ACM, Austin (2016)
30. Zeng, X., et al.: Automated test input generation for Android: are we really there yet in an industrial case? In: Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp. 987–992. ACM, Seattle (2016)
31. Baride, S., Dutta, K.: A cloud based software testing paradigm for mobile applications. SIGSOFT Softw. Eng. Notes **36**(3), 1–4 (2011)
32. Starov, O., Vilkomir, S., Kharchenko, V.: Cloud testing for mobile software systems-concept and prototyping. In: ICSOFT (2013)
33. Chiatti, A., et al.: Text extraction from smartphone screenshots to archive in situ media behavior. In: Proceedings of the Knowledge Capture Conference, pp. 1–4. ACM, Austin (2017)