APPLICATION LAYOUT TESTING FOR ANDROID MOBILE DEVICES

DANIËL KOK

Challenges & Solutions

Supervisor:
Prof. Dr. Alexandru C. Telea
Second Supervisor:
Prof. Dr. Marco Aiello

ABSTRACT

The mobile phone today, is integrated into everyday life. Different operating systems and hardware, fragment the mobile device space. This fragmentation makes it difficult to ensure that a mobile application (app) will be displayed correctly without manual testing on devices. Errors in the layout instance might cause functional problems, rendering an app unusable. The high number of hardware and platform combinations, make it worth the effort to automate layout testing for apps. Because the Android OS and device space are highly fragmented, we focus on Android.

From our research on testing tools for mobile devices we conclude that support for layout testing in existing tools is very limited and there is no universally accepted standard for layout testing. For this reason we analysed the requirements for a generic layout testing framework.

In the requirements analysis we abstracted from both platform and operating system. Important general requirements are automation and device/platform independence. The requirements that are specifically directed towards layout testing include the ability to specify layout issues in an easy and precise way and apply these rules to an app. The application of rules must obtain a clear report on the errors that were detected.

With these requirements, we implemented a solution that is capable of automated detection of problems in layout instances. The solution consists of an XML-format to specify rules that use XPath expressions to indicate the set of components to which a rule should be applied. Boolean expressions can be added as constraints on these components. When a constraint is not met, the violating component is indicated visually on a screenshot. The results show that the implemented solution enables users to easily specify rules and constraints on user interface components and detect actual problems in layout instances. By using scripting to extract the layout instances from an app, the whole process has been automated.

Although we have shown that this solution is able to detect errors in layouts, more work is needed to cover all interface components and problems that might occur. Moreover, research is needed on what is generally considered wrong in layouts to be able to specify more useful rules for the system.

ACKNOWLEDGMENTS

I would like to thank the following people: Prof. Dr. Alex Telea for supervising me and for giving me very detailed feedback. Rix Groenboom and Jan Tijmen Udding for their help at the start of this project and support during the process. Julian Harty, for sharing his expertise on the subject and providing feedback. Prof. Dr. Marco Aiello, for reviewing my work. Finally, thanks to my girlfriend, family and friends for their unlimited support.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LISTINGS

ACRONYMS

API	Application Programming Interface
OS	Operating System
UI	User Interface
GUI	Graphical User Interface
URL	Uniform Resource Locator
API	Application Programming Interface
XML	Extended Markup Language
LI	Layout Instance
LS	Layout Specification
LE	Layout Engine

1 INTRODUCTION

The mobile phone today is integrated into everyday life. The smart phone, PC and laptop have evolved into a wide range of devices varying in size, input types and available sensors. Since the introduction of the iPhone with the iOS [22] Operating System (OS) in 2007 we have seen a change in the way that we interact and use mobile devices. Some key factors in this change include the switch from user input with physical keys to a touch screen and the concept of a central application store where everyone can offer mobile applications for purchase.
The iOS OS was followed by the introduction of Android [18] in 2008 and Windows Phone [25] in 2010 which follow similar design principles. While the main principle of a touch interface and central application have remained, the number and types of devices have grown significantly. Different studies [34] showing the differences between traditional software development and mobile software developement, present the need for an adapted approach. The mobile device space is fragmented on the OS and device level. This fragmentation is causing apps to look and behave different on different devices. These differences can be both functional and visual. Users of an app might not get the appearance and funtionality that was intended when developing the app. The large number of possible hardware and software combinations pose a problem for testing apps. This thesis discusses the need for a mobile layout testing framework and how it can be implented. This chapter introduces the concepts of apps, testing, layouts and why testing for mobile devices is difficult.

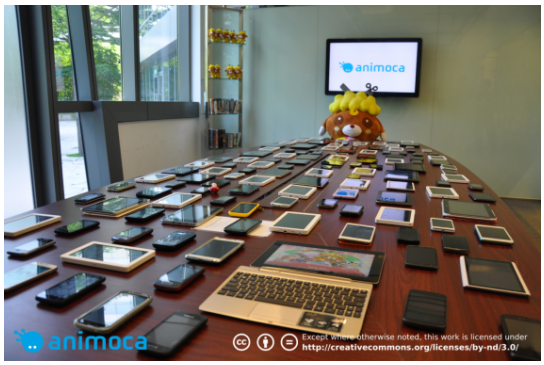


Figure 1: Device Fragmentation *(Image from Animoca)*

1.1 APPS

The word "app" is an abbreviation of the word "application". A mobile app is considered a software application designed to run on mobile devices, commonly referred to as "app". Applications running on mobile devices are usually downloaded and installed through a central location like the App Store for iOS [21] devices and Play Store for Android devices [24]. From now on we will refer to a mobile application as "app".

1.2 FRAGMENTATION

A finished app is eventually downloaded and installed on *a device*. The context in which the app will run differs from device to device. Different devices can be equipped with different types of hardware, different types of OSes and versions of the same OS. The existence of different contexts in which an app can run is called *fragmentation* [32].

1.2.1 *Hardware Fragmentation*

Hardware fragmentation is caused by variation in components that are physically available on a device: display, communication devices, sensors (input methods) and storage devices. The display is the most prominent component. A display can have different properties like width, height, resolution, refresh rate, color depth etc. A device can have a range of communication devices like BlueTooth, WiFi, infrared or GPRS/3G. Also, there are different input methods like a touchscreen or a physical keyboard. Furthermore, a device can have additional sensors like a camera or movement sensor. Finally, a device can provide different methods of storage like built-in flash memory or swappable SD-cards with varying sizes.

1.2.2 *Operating System Fragmentation*

Different mobile devices run different OSes. Not only are there different OSes but the OSes themselves are continuously changing because of the introduction of new features, bug fixes and adaptions for new devices. There are multiple reasons why different versions of the same OS coexist: New operating systems are developed, support for newer hardware is implemented, older hardware is not supported anymore or manufacturers and sellers of a device do not offer newer versions to their users, leaving users with no other choice then to keep using an older version.

1.2.3 *Field Study: Android*

We now know which properties of a device can vary. Is this realy a problem in practice? Android has gone through 4 major versions [19] over a period of four years. Figure 1 illustrates only a selection of different hardware devices running Android. A study of devices downloading an OpenSignal maps app over a period of 6 months showed a number of 3997 distinct Android devices running different versions of Android[23]. More than 50% of the devices were running Android 2.3.3, 20.5% with Android 2.2, 5.7% on Android 3.3 whilst Android 4.0.4 was the most recent version at the time. The range of screen sizes varied from 240*180 pixels to 1920*1200. We see fragmentation on both device and OS level amongst devices that are being used. This provides a good base for further research of app layout stability. For this reason, we use the Android OS for further analysis.

1.2.4 *Development and Testing*

The existence of fragmentation amongst devices and software means that "write once, run everywhere" is a hard task for multiple reasons. First of all we can not assume that a device will have all properties that an app requires. Most devices have a camera, but is the camera resolution high enough to support the functionality of our application? Does the available version of the OS support the User Interface (UI) controls that we are using? These configurations need to be taken into account when developing. The existence and concurrent use of all these devices also means that testing is not a trivial task. The large variety of devices makes it impossible to have all variations in house. This is especially the case for Android devices. Further, if more devices are used for testing, more results have to be interpreted. Hence, we see that fragmentation does not only cause problems for end-users, but also for testers. We elaborate on this in the next section.

1.3 TESTING

As an app is a piece of software, ensuring a certain level of quality can be achieved by various testing methods. Different characteristics of applications require different testing methods and measures for software quality. Multiple models of software quality for general applications already exist like the the ISO/IEC standard for Software Quality [28]. Different characteristics of applications are mentioned here: Functionality, reliability, usability, efficiency, maintainability and portability. To state anything about these characteristics, apps need to be tested. This can be done by using different types of

testing. We can divide testing methods into two main groups: static and dynamic.

1.3.1 *Static Testing*

Static analysis is done on code and is performed without executing the application [30]. It can be used to verify that the code adheres to certain standards or rules. This can be done by either manually reviewing the code or using (automated) tools. Static testing can be used to verify syntax and coding standards, optimize code, detecting (potential) programming mistakes. By using tools to automatically verify the state of the system regularly, errors can be spotted early on in the development process. This verification is done by defining a set of rules. These rules define specific patterns that can occur in any program. Usually, there is no notion of what the code is supposed to do. This means that no test cases have to be written. The disadvantage of this is that only the rules that are defined in the static analyser are checked. Each rule can have a low or high priority depending on the context. For example: an unused variable may take up unnecessary memory, but is generally not harmful for the execution of the program, a possible null-pointer dereference can cause a crash. It is up to the programmer to select the rules that should be checked. It is possible that some rules generate so many errors that they are not used anymore.

Another important limitation of static testing is that it can only test properties which are measurable from the source code. However, many properties of an application can only be observed when actually running the application - for example, actual memory consumption, interaction with third-party libraries via dynamic binding, or behaviour of the application which is driven by user input. Such aspects cannot be measured or assessed using static testing only.

1.3.2 *Dynamic Testing*

Dynamic testing is done when the application is compiled and running [27]. In contrast to static testing, test cases have to be defined that specify the expected output given a certain application state. This type of testing analyses the runtime behaviour of an application. Examples of dynamic testing methods are unit, integration and system testing. For example, we could test a function that takes an image and returns its average color value, and a method that returns an image captured by a camera. Testing these two components in combination with eachother is called integration testing. When all the necessary components are tested together, this is called system testing.

While dynamic testing works for testing those application properties which are only observable at run-time, this type of testing also has its limitations. The dynamic execution depends on environment and human input variables that can be hard to mimic.

1.3.2.1 *White Box Testing*

White box testing is testing *with* knowledge of the application code. Test cases can be written based on the execution of specific parts of code. An example of this is unit testing. A method that should return the sum of two numbers can be tested by writing a test case that calls the method with two predefined values and compares the return value to the precalculated sum. Typically, static ananlysis is a form of white box testing because rules are often applied to the source code.

1.3.2.2 *Black Box Testing*

Black box testing is testing *without* knowledge of the application code. In this case the internal workings of a component are not known and we can only influence the variables and program state through the externally available interfaces. An example of this is an app that applies a filter to a chosen image from the file system. An example test would be to select an image and compare the returned image to a prepared image. We can not test the inner workings of the filter algorithm because it does not expose other interfaces.

Dynamic analysis is typically a form of black box testing, where the external interfaces are used to perform testing.

1.3.3 *Difficulties*

When we look at testing with regard to device and OS fragmentation, we can identify a number of issues that need to be overcome. For static analysis there is no need for devices because the analysis is performed on code and not a running app. This is an advantage over dynamic testing. The downside is that we need to know every difference and issue that may occur on a device/OS combination to be able to spot them before runtime. Not only is this a difficult task, it might also overload the developer with errors to which there might not always be a direct solution.

Moreover, there is the possibility of generating a lot of false positives [30]. Because the actual system context and values of certain variables are not known before runtime, some rules might generate a lot of warnings. Leaving it to the developer to inspect every warning. With dynamic testing all variables are known. The downside is that an actual device or emulator is needed. To be absolutely sure that an

app functions as expected, testing needs to be done on all devices that the app must run on.

1.3.4 *Usability*

To ensure that an app will be functional and usable, the following points need to be taken into account:

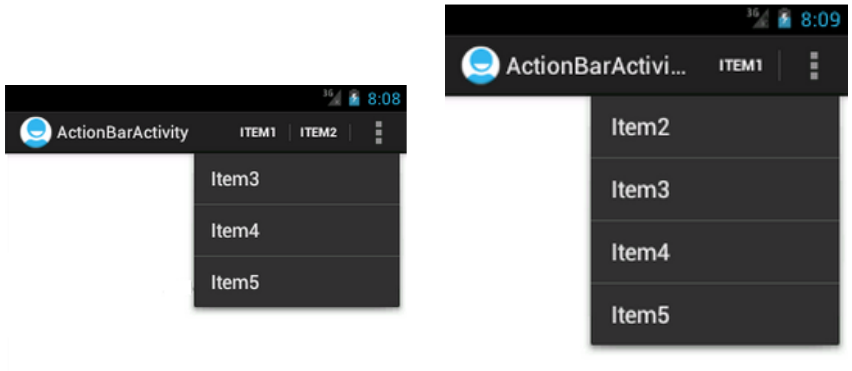
- Device Fragmentation
- Operating System Fragmentation
- Input methods

1.3.5 *Conclusion*

The combination of OS version and available devices makes it very hard to test apps on all of these devices. The process of manually testing and visually inspecting on a device is tedious and hard to maintain because of the continuous introduction of new devices. Because apps are software applications, the same principles and practices can be applied up to some point. What makes desktop applications really different from apps is the devices they run on, the environment in which they are used: in the train, while walking etc, and a different application lifecycle. Add to this the previously mentioned device and OS fragmentation and different input methods, we conclude that this calls for a different approach. Because mobile devices use the screen to present data to the user as well as being a direct input method, the layout of components is an important part of creating a functional and usable app [34].

1.4 LAYOUT TESTING

The appearance of an app is determined by the way in which UI components are layed out. A mobile layout consists of a collection of UI elements having different properties. When we want to test the layout we want to know whether it will be presented in the way that is was intended. Differences in device hardware can cause layouts to be presented differently and maybe not as they were intended. Figure 2 shows an example of how the same layout can cause different outcomes on different devices. The outcome of the final rendering is generated by three components: Layout Specification (LS), Layout Engine (LE) and Layout Instance (LI) (see figure 3).



(a) Landscape mode

(b) Portrait mode

(c) Portrait mode with hardware menu button

Figure 2: Three layout instances for the same layout specification

1.4.1 *Layout Specification*

In most mobile operating systems, layouts can be specified in an Extended Markup Language (XML) format, code or a combination of these [1][15]. We call this a *layout specification* (L_S). We define LS as the entire set of layout specficiations that can be written in a specific

language. L_S specifies how the operating system should layout the content of an app. Mostly, the specification does not contain actual content. Unless precise measurements and locations are given in the specification these are not known before runtime. Even if they are given, the layout engine might decide that they need to be changed due to restrictions from parent components or conflicting constraints.

1.4.2 *Layout Engine*

How the eventual layout instance (L_I) will be rendered is not certain until the app is compiled and run on an actual device or emulator. Even then, we have only determined the layout instance for a specific device and operating system combination (D from all possible combinations DS). The layout engine takes the layout specification and calculates their sizes based on a number of parameters. These parameters consist of properties that are set in the specification and properties that are determined by the device hardware. For example: a text component has a property *width* that is set to *fill_parent* which means that this component takes up the full width of the parent component. If this is the root component it takes up the full screen space that is reserved for the app by the OS. If the app is running in a window this can be the size of the window. This means *fill_parent* is not a fixed size and depends both on the structure of the specification as well as the device hardware. We can define the behaviour of a layout engine as follows:

$$I : LS \times D -> LE, I(L_S, D) = L_E \tag{1}$$

1.4.3 *Layout Instance*

The final outcome of the calculations from a layout engine L_E, generate a *layout instance* L_I. The operating system takes this specification and renders a layout instance. The layout instance can be seen as the final result: what is presented to the end-user. But the instance on itself is not something that can be changed. It is a product of layout specification and layout engine. When changes are introduced by other processes e.g. user interaction or other background processes the instance might needs to be recalculated.

1.4.4 *Windowing and GUI toolkits*

Traditional OSes provide options for changing the appearance of an application layout through a window manager like the X-window system [26]. Most window managers include actions like moving, minimizing, maximizing, resizing and switching windows. Depending on

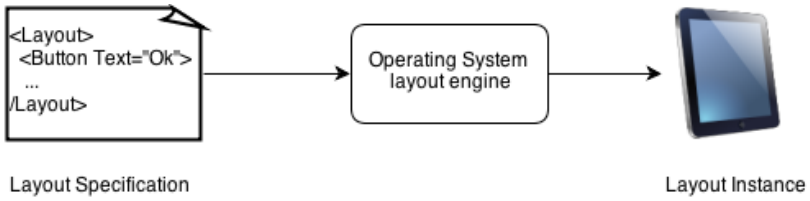


Figure 3: From layout specification to layout instance

the UI components inside the window, these are often resized within the window based on these actions. A mobile window manager is very limited in this area. Its functions are limited to assigning memory for an app to draw its components. Furthermore, the available UI widget toolkits are often limited to components for which the size is not adjustable by the user but dictated by the size of the display and the components' content and/or preset size values. The exact meausurements depend on the implementation of the specific renderer for a specific OS.

1.5 LAYOUT TESTING TOOLS

A layout testing tool is a tool that can indicate potential or actual problems that may occur in a layout specification *or* layout instance. They are tools that can (partially) automate the detection of problems in one of the components that are involved in layout testing. Over the years multiple tools and methods have been developed for testing various parts of an app for different OSes [31]. Some tools focus on instance and some tools focus on specification. There is generally no specific focus on layout testing. Also, there is no universally accepted testing standard for layout testing. Layout testing can be done on different levels. To be able to differentiate between them we distinguish three types:

- *Conformance*: Does a layout instance conform to the layout specification.
- *Uniformity*: Does the layout specification provide a *uniform* appearance accross different devices and operating systems.
- *Specification Standard Checking*: Does the layout specification conform to a set of rules.

Here we will present these types along with some examples.

1.5.1 *Conformance Testing*

A layout specification defines relations between Graphical User Interface (GUI) elements. It defines how and where they should appear on

the screen but it does not dictate the final appearance. When given a layout specification L_S and a layout instance L_I, does this layout instance match what we would expect from the specification?

1.5.2 *Uniformity Testing*

When a single layout specification L_S is rendered on different devices $D_1, D_2, \ldots$ this results in an equal number of layout instances $L_{I1}, L_{I2}, \ldots$. When testing for uniformity we would like to know if the initial specification L_S results in a *uniform* appearance accross these devices.

1.5.3 *Specification Standard Checking*

Different UI components have their own unique properties. For example, an image component might have a pointer to an Uniform Resource Locator (URL) for the image to be displayed whereas a text component has a text property for the text to be displayed. These properties of the layout specification can be checked against a set of rules that define which properties are permitted and how they should be formatted.

1.6 DISCUSSION

With the current growth of mobile devices and apps it is becoming more and more important to support developers on how a layout will be presented. The final presentation is important for the end-user as it must convey how an app is to be used. Function 1 shows that this final presentation can be varied in two parameters, namely L_I and D. What we want to know is how much this function varies when changing these parameters. When we have a single specification L_S, but change the device D, do we obtain a different instance L_I? In other words: we want to be able to find differences between the different instances. This can be defined as follows:

$$\mathrm{diff} : L_I \times L_I -> R \tag{2}$$

Where R is a real number indicating the difference between to instances that are generated by two different layout engines from different devices from the same layout specification.

$$\begin{aligned} LI_1 &= I(LS, D1) \\ LI_2 &= I(LS, D2) \end{aligned} \tag{3}$$

Given the difficulty of performing layout testing because of device and OS fragmentation and the lack of dedicated layout testing tools, we conclude that further analysis of layout testing towards a layout testing framework is a valuable addition to the mobile testing spectrum.
newline
In this thesis we analyse the requirements needed for a generic layout testing framework. This set of requirements can be used as a base for any layout testing activities. From these requirements, we have implemented a selection which enables developers to specify and detect layout issues in apps.

1.6.1 *Thesis Structure*

This thesis is structured in chapters containing the following. Chapter 2 starts with an analysis of existing tools that provide functionality for layout testing. We discuss the advantages and disavantages of these tools and determine which types of testing are covered. This is followed by an analysis of requirements for a generic automated layout testing framework in Chapter 3. From these requirements a selection is made based on what is feasible in general and within the given time for this thesis. We then propose an architecture for a layout testing framework given these requirements in Chapter 4. This is followed by a discussion of the results of an implementation of this framework in Chapter 5. Finally we discuss how the other requirements may be fulfilled in Chapter 6.

2 RELATED WORK

In recent years different methods and tools have been developed and researched to aid in mobile layout testing. In this section we describe a number of tools that are available for mobile testing and specifically layout testing of Android apps. Some tools focus completely on layout analysis whereas others provide some functionality to perform layout testing. For each tool, a description of functionality is presented along with the type of testing that they support. The testing types are mentioned in chapter one, Section 1.5. We also discuss the advantages and disadvantages of each tool. This chapter has been split up into two main sections for tools that perform static testing and dynamic testing. In the final section we discuss common advantages and disadvantages of all tools and methods described in this chapter.

2.1 STATIC TESTING TOOLS

Performing static analysis is the analysis of source code without actually running it. Analysis can be performed on either source or object code [30]. Static analysis tools can be used to find common programming mistakes, dictate specific programming styles or detect weak security constructs e.g. the use of a non-secure connection. Static analysis tools have been around for quite some time. Because Android applications are developed with the Java programming language, existing static analysers for Java like JTest or FindBugs can be used to perform analysis with respect to Java coding conventions. Unfortunately these tools are not focused on Android and Android's specific UI system although they do provide some support for Android [7][4]. With respect to layout testing we are specifically interested in analysing the *layout specification* by scanning the code for predefined patterns. This section describes a number of tools that can aid in static analysis of the layout specification.

2.1.1 *Android LINT*

Android LINT[8] is a tool that performs static analysis on both layout resources (XML) and Java source code. This tool scans an Android project source for potential bugs and potential optimization improvements for security, usability, accessibility and internationalization problems. We are interested in the usability aspects of this

tool. As mentioned before, this tool has a number of built-in checks. An example of a usability check:

```
TextFields
----------
Summary: Looks for text fields missing inputType or hint settings

Priority: 5 / 10
Severity: Warning
Category: Usability

Providing an inputType attribute on a text field
improves usability because depending on the
data to be input, optimized keyboards can be
shown to the user (such as just digits and
parentheses for a phone number). Similarly, a
hint attribute displays a hint to the user for what
is expected in the text field.
```

This an example of a rule that affects the final presentation to the user. If a textfield does not have a inputType attribute set, the presentation of the keyboard to the user can be different and affect the user experience. When the user is asked to enter a number but is presented with an alphabetic keyboard this changes the user experience.

2.1.1.1 *Advantages*

An advantage of LINT is that errors can be caught in an early stage before deploying on a device or emulator. Also, all files can be checked subsequently and there is no need to run the application. The checks can be done without actual execution of the app. Furthermore it is not necessary to write test cases. By using the plugin for the Eclipse IDE, LINT warnings can be presented within the IDE in the same fashion as compiler errors, giving developers almost instant feedback on their code. On the other hand LINT is not dependent on these plugins. It is a stand-alone command line tool which can easily be integrated into other environments.

2.1.1.2 *Disadvantages*

A disadvantage (See Section 1.3.1) of using static analysis for layout testing is that not all variables are known at compile time. Most apps contain some form of dynamic data e.g. lists that are filled by data from a file or a database. This means that the layout has to scale to fit the data. The size of UI components can only be known for sure at runtime and depends on the renderer that is used. Though static analysis can be used to detect patterns that are known to be unwise or cause layout problems, it can not do more than give indications. A

problem with this is that a lot of indications might prove to be false positives. Developers have to inspect each warning and determine for themselves if they think this will be a problem or not.
The advantage of not having to create test cases can also be a disadvantage. The absence of errors does not mean that no problems can occur.

2.1.1.3 *Testing type*

We consider this type of testing, specification standard checking. Lint has defined a number of standards to which the code should adhere. As in the example above, all textfields should specify an inputType. This is a specification standard. When an textfield without a inputType attribute is encountered, a warning is generated.

2.1.2 *MOTODEV App Validator*

The MotoDEV App Validator[11] is a tool similar to Android LINT but customised for specific Motorola Devices. It performs a range of checks to see if an app is compatible with Motorola devices. It contains a set of predefined device profiles that contain different properties of a device like the size of the screen, the availability of WiFi or the ability to perform Phone Calls.

2.1.2.1 *Advantages*

The main advantage of App Validator is its ability to perform checks against a specific device profile or multiple profiles. This enables developers to adapt their apps to be compatible with specific devices before the app is run. Although not every problem can be detected before runtime, the detection of at least some problems can save time spent on dynamic testing.

2.1.2.2 *Disadvantages*

This tool is tailored towards Motorola devices which are just a small subset of all the devices available on the market. However, it does provide the ability to add custom device profiles, they would have to be added manually.

2.1.2.3 *Testing type*

Like LINT, this is a type of specification standard checking. Although, this tool does contain rules that are very specific for the different devices that Motorola delivers, we can also place it under uniformity testing.

2.1.3 *Graphical Layout Editor*

A Graphical layout editor lets developers construct their layouts in a point and click fashion and enables developers to preview layouts on different screen sizes. The graphical layout editor for Android[5] is part of the Android Development Toolkit. It enables developers to switch between the specification and a graphical rendition of this specification. It consists of four major components: The canvas, hierarchy view, component palette and a configuration chooser. The canvas resembles the screen of the device. Components can be dragged and dropped from the component palette onto the canvas to visually construct a layout. The hierarchy view shows the hierarchy of the components on the canvas. For example a linear layout component contains three buttons that are layed out in a linear fashion. Finally the configuration chooser lets the user select different screen sizes, screen orientations, themes and OS versions changing the canvas accordingly to these settings. What must be kept in mind is that this rendition is not a runtime version of the layout. It is based solely upon the specification and does not contain dynamic data or components that are generated programmatically. It is a rendition of the layout specification and does not enable any interaction on the layout.

2.1.3.1 *Advantages*

An advantage of this type of rendering is that it gives the developer a visual preview of the layout specifcation. This gives the designer a better impression of the eventual layout than a static text file. However, the eventual rendering can not be presented because not all variables are known beforehand. The possibilty of choosing a different configuration also enables the designer to see how components adapt to changing configurations. For example, we place a textfield on a canvas in portrait mode and stretch it to the full width. Now we change the configuration to landscape mode and the component is only half the size of the screen. This happened because the width of the component is specified in a fixed number of pixels.

2.1.3.2 *Disadvantages*

The main disadvantage of this tool is that it does not explicitly indicate potential errors. The user has to manually select different configurations and visually inspect the appearence of the layout. Further, the absence of runtime data means that we can not see how the specification will display on an actual device. Another disadvantage is that, although the user can select a platform version this does not always exactly resemble the renderer that is used on a device.

2.1.3.3 *Testing type*

This tool enables users to do a type of uniformity checking, manually. It lets the user specify the display size or sometimes an actual device. In the latter case, display properties are predefined. In this way the developer can get an impression of how components will change position and scale on different devices.

2.1.4 *Image Based Testing*

Image based testing is a method that analyses images that represent the mobile screen at a specific point. The most basic form of image based testing is comparing screenshots pixel by pixel from an app that were taken at a different point in time. The obvious problem with this is that the images have to be aligned exactly the same, and the data has to be static. [33] This is not very practical. Although, screenshots can be used to locate information that we are wanting to see on the screen by using other image processing techniques like Optical Character Recognition.

The Android Developer Tools provide a tool called Pixel Perfect[6] which is a part of the Hierachy Viewer. It enables users to inspect the screen at the pixel level providing information about location and color and the UI component in which it is contained. It also provides the functionality to load a bitmap image which can represent a specific layout design. The layout design can be used to test whether the layout conforms to the intended design.

2.1.4.1 *Advantages*

The advantage of using image based testing is that what we see in the screenshots is exactly what the user sees in terms of data on the screen. This means that there are no more abstractions between the data that we are analysing and the presentation to user except for different external lighting conditions and screen contrast. So, the data is not subject to any further interpretation for the type of device that it is retrieved from.

2.1.4.2 *Disadvantages*

The main advantage is at the same time a disadvantage. We need to perform additional operations to be able to tell what the pixels actually represent. The pixel-by-pixel comparison is very sensitive to change[33] to be of real use in detecting errors in the layout. Furthermore, there is no support for automation in the Pixel-Perfect tool. All inspection has to be done manually.

2.1.4.3 *Testing Type*

We can put this type of testing under conformance testing. We can use pre-made images to test whether the current view on the screen conforms to this design. This design can be seen as a layout specification. However, we can not directly match it to the layout specification in code.

2.2 DYNAMIC TESTING TOOLS

Dynamic testing tools are tools that analyse an app at runtime. It is less prone to false positives because it inspects the actual output of the code. On the other hand it involves more work designing and maintaining test cases that both find crucial problems and produce a reliable confidence level in detecting bugs in all parts of the app [27].

2.2.1 *Hierarchy Viewer*

The Android hierarchy Viewer[6] enables users to inspect the the relation of interface components on the screen along with their properties at runtime. It displays the layout that is currently displayed on a mobile device or emulator in a tree form where each component is represented by a block and the connecting lines indicate what components are children of other components. The properties are basically all values that make up the size, location and appearence of a component. Properties shared by all UI components are x and y-position and width and height. The hierarchy viewer also provides information about performance. It states for each component the time it took to measure, layout and draw the component on the screen. Before a component can draw its children it first has to know how large they are going to be. This is the measuring part. Each parent invokes the measure method on all children. When all sizes are known, the parent can layout its children accordingly. When the layout pass is done, all components can be drawn to the screen.

2.2.1.1 *Advantages*

Advantages of the hierarchy viewer are that we can inspect the different properties as they are actually displayed on the screen. It gives more insight in the layout tree behind the actual representation which might give valuable insights into why components are displaying the way they are: The layout tree does not match what is supposed to be. Also, the composition of the view tree might give insights in performance problems.

2.2.1.2 *Disadvantages*

The main drawback of the hierarchy viewer is that it can only be used for emulators or devices that have a custom OSes installed. Another disadvantage is that it does not provide functionality for test automation. The user must run and use the app and inspect the hierarchy manually. It is an inspection tool that enables users to inspect various properties but leaves the interpretation to the user. Also, because the viewer is run on a specific emulator or hardware device we only test the interpretation for this device.

2.2.1.3 *Testing type*

We can use the hierarchy viewer to perform conformance testing. The view hierarchy can be matched against the layout specification to check whether it is conform the specification. With this method we check if the renderer interprets our specification correctly by generating the correct UI objects. Furthermore, the corresponding layout instance can be inspected: do the UI objects on the screen match the expectation of the specification.

2.2.2 *GUI Test Automation*

GUI Test automation is testing by actually using the GUI as a human would interact with it [33]. This can be done by scripting the actions that a user performs and use an Application Programming Interface (API) that enables developers to interact with the GUI programmatically. Record and Playback tools are a type of tooling that accomplishes this functionality by recording the actions that a user performs when using the app. Every action is recorded in a script. This script can then be run at a later point in time. Some tools enable either the generation of a script through using the app or by manually writing the scripts.

2.2.2.1 *Scripting*

Most tools rely on the use of some scripting language. These languages can be common programming languages like Java which is used by Robotium[12] or a more natural type of language like Calabash used by LessPainful[10]. When we look at some examples, the difference is obvious:

Listing 1: "Calabash test script"

```
Given I am on the "login screen"
Then I should see "Enter Your ID or Email"
Then I enter "KamalAnand"
Then I should see "Password"
Then I enter "password"
Then I touch "Sign In"
```

Listing 2: "Robotium test script"

```
assertTrue(solo.searchText("login"));
assertTrue(solo.searchText("username"));
solo.enterText(1, "TestUser");
assertTrue(solo.searchText("password"));
solo.clickOnButton("Sign In");
```

The first is directed towards the perspective of the user and the actions that a user takes. Looking at the Java code we see a similar pattern, though it is hidden amongst typical Java language constructs. The advantage of the first is that is easy to understand for everyone, but lacks the functionality of a language like Java. The Robotium version can make full use of the Java language but is harder to maintain and understand.

2.2.2.2 *Advantages*

The scripting abilities of these tools enable to run the app automatically and perform assertions on the results. An advantage of using scripting is that it can be reused for different devices and possibly for different operating systems. Also, because we are doing assertions on the actual output, we can be sure that a specific error is actually occurring.

2.2.2.3 *Disadvantages*

A problem with this type of testing is that the UI can be functioning correctly but not displaying correctly. Although it is possible to add code to check visibility, size or other UI component properties, this all has to be done manually and specified per component. There is not a general method for detecting errors in the layout. Another issue is the maintenance of test cases. Small changes in the app can break existing test cases because UI components have been renamed, removed or otherwise changed. This means that the validity of test cases has to be guarded cautiously as the app is being developed.
Another issue is that either an actual device is needed which means you can only test for that specific device, or an emulator which is never exactly the same as a real device which might make tests unreliable[31].

2.2.2.4 *Testing type*

With regard to layout, this type of tool does not really fit into any of the defined types because it is not specifically designed to inspect the layout. However most tools have the option to take screenshots at predefined moments and methods are offered to programmatically inspect the layout like the size or position of components, but the user has to write his own test cases. With respect to layout it provides the ability to a form of semi-automated conformance testing. The tool can run and use the app whilst taking screenshots at specific moments. The user has to manually inspect all the screenshots to check whether they conform to the intended specification.

2.2.3 *Testing in the cloud*

Testing in the cloud is not a single tool like the previously mentioned methods, but more a different environment in which these tools might be used and offered as a sevice. Mostly this is a form of a record and playback type of tooling. Users can write or generate testscripts, upload them to the cloud along with an app and let them run on chosen set of devices. This type of service is provided by TestDroid[13], LessPainful[10] and DeviceAnywhere[3]. Another service provided by UTest[14] is crowd sourcing. The app is distributed amongst subscribers to the service along with test scripts or a description of the functionality that needs to be tested.

2.2.3.1 *Advantages*

A huge advantage of this type of service is that you do not have to buy a large number of devices. Where a record and playback tool enables users to reuse testscripts for different devices, these services enable users to deploy them to multiple devices without having to build and maintain an infrastructre for this as well as having to buy the devices. It has all the advantages of GUI testing along with the advantage of easily deploying the tests on a range of devices.

2.2.3.2 *Disadvantages*

Although most providers have a large set of devices, the set of available devices is still limited. Depending on the provider you might not find the device that you need and adding one is not always an option. When using crowd-sourcing, we have to take into account if users are actually executing and interpreting the tests correctly. If we can not be sure that it is performed in the way we want, the tests are unreliable.

2.3 DISCUSSION

When we look at the advantages and disadvantages of each tool we can extract a number of items that seem to apply generally for both static and dynamic tools. In static analysis, we see the following recurring advantages and disadvantages:

+ No need to compile and run app on a device (LINT, Graphical Layout Editor, MotoDEV App Validator)
+ No need to write specific test cases (LINT)

And recurring disadvantages:

- Missing runtime data: The layout instance can not be generated with just the specification (LINT, MotoDEV App Validator)
- Coverage of rules: Do the rules cover the problems that we want to find (LINT)

In dynamic tools we see the following recurring advantages and disadvantages:

+ The layout instance is known, which means that what we detect is actually occurring on a specific device (Robotium, TestDroid, LessPainful, DeviceAnywhere)
- Need to test on actual devices (Robotium, Image Based Testing)
- Coverage of an app: Need to write test cases or manually use the app (Robotium, TestDroid, LessPainful, DeviceAnywhere)

The problem with static tools not being able to generate the layout instance is that we do not have access to each possible layout engine. Because each combination of OS and Device is able to generate different instances, this means that each of these engines is needed to be able to predict the outcome. Even though the layout engines are not directly needed, knowledge of them is necessary to be able to predict errors with static analysis. Even if we have this knowledge we are still missing the ability to check for consistency amongst these devices. Dynamic tools are able to test the layout instance, but they need to run on actual devices. Cloud platforms solve this problem by providing services to run an app on multiple devices but it is still necessary to check for conformity by looking at generated screenshots or feedback from actual users.

In the next chapter we are going to examine the requirements for a layout testing framework that takes these disadvantages into account. Based on the requirements we make a selection based on the advantages and disadvantages of implementing these requirements.

3

TESTING TYPES ANALYSIS

In this chapter we discuss the requirements that we expect to see in a generic layout testing framework. Generic meaning that we do not initially set any constraints to the environment for which the testing framework is to be used. We start with the notion of an app A for which we want to know whether its layout specification (L_S) will cause problems for usability or functionality across the whole range of mobile devices (D). These requirements are based on the types of testing mentioned in Chapter 1: Conformance, uniformity and specification standard checking. Because these testing types have a number of requirements in common we start with general requirements for such a framework. These are followed by the type specific requirements. Each requirement is first discussed and described. This is followed by a list of requirements. Finally we discuss, compare and select a subset of requirements based on different choices and the feasibility of implementing each requirement.

3.1 PURPOSE

The purpose of a generic layout testing framework is to provide a standardised way of performing layout testing for mobile applications. Our research on the existing tools (Section 1.5) points out that there is no standardised method of doing so for all platforms and not even for a single platform. It is clear however, that being able to detect whether the function $I(L_I, D_N)$ will cause problems is an important aspect of mobile testing. The tools mostly provide handles to make the process easier but the actual validation mostly relies on visually inspecting a layout instance. Because the issues may occur on each mobile platform existing at the moment of writing and possibly platforms to come, it is useful to define standards for this specific purpose. This system aims to set that standard.

3.2 CONTEXT

Several different researches have been conducted on the subject of mobile testing frameworks. Franke and Weise [29] describe a general software quality framework for mobile applications. They pinpoint that flexibility and adaptability are important drivers for mobile applications. The layout testing framework is a framework that falls into the category of a mobile software quality framework. Its focus lies on testing a specific aspect of apps: the presentation to the user.

3.3 ANALYSIS

Most of the tools mentioned in Chapter 2 only provide partial functionality for layout testing and most include a lot of manual handling. In this chapter we want to combine the advantages of the different tools as mentioned in Section 2.3 and incorporate solutions for the disadvantages into a single layout testing framework that overcomes these problems.

3.4 GENERAL REQUIREMENTS

This section describes requirements that any generic layout testing framework must comply with to be useful and usable. These requirements are for any testing framework regardless of platform, language or device type. We start off by discussing the necessary requirements. This is followed by a table containing an overview of all requirements.

3.4.1 *Layout specification Coverage (RQ1)*

Firstly, the framework must cover all UI elements that are supported by the OS that testing is done for. If for example we only support testing of buttons we are not able to validate the complete layout. In that case a layout without buttons would be considered correct. So, we want to cover all elements that can occur in the layout specification. If the coverage of UI elements is only partial, parts of the layout remain untested. In other words: given an App A, it is enough to cover all elements of the domain LS.

3.4.2 *Platform Independence (RQ2)*

We want to achieve a level of platform independence. We want to be able to specify test cases for different OSes without having to replicate them for each OS. In other words we want to evaluate the function *diff* (2) on all possible inputs. This does not only apply to different OSes but also to different versions of the same OS. When specifying rules or test scripts, these rules must contain platform independent elements. This means that if we want to automate an action (see Section 3.4.4) to press a button or specify a rule to which a button must comply this specification is generic enough to specify a button press e.g. iOS, Android and Windows Phone.

3.4.3 *Device Independence (RQ3)*

We want the test cases to be able to run on different devices without the need to change anything within the test case. Given a Platform P,

we want to be able to run tests on all devices D from DS that run P. These can be different types of devices like smartphones and tablets but also different device brands.

3.4.4 *Automation (RQ4)*

Many of the tools seen in Chapter 2 do provide different views of the application but do not indicate any errors themselves. Also, most tools do not provide the ability to specify test cases and automate the functionality. We want to be able to specify tests in advance. Once our tests are created we want the tests to be executed automatically i.e. not having to manually run and use apps to obtain the results. With respect to layout we want the framework to automatically indicate were problems occur.

3.4.5 *App Coverage (RQ5)*

We must be able to test all instances of layout elements that *can* occur within an app. Given a layout specification L_S we want to test on all possible outcomes of the function $I(L_S, D)$. Most UI components in an interface framework have properties that can be set or changed by the developer. This means that, although some items are programmatically the same, their instances can vary depending on how the properties are set. The value of all properties depends on the program state which is determined by the subsequent steps of user interaction and external data.

3.4.6 *Error reporting (RQ6)*

The framework must present a comprehensive error report to the user. Instead of just indicating that something has gone wrong it must provide suggestions for where to make changes to fix the problems that are found or provide means of indicating the problem visually.

3.4.7 *Environment selection (RQ7)*

The layout framework must provide a method of specifying the environment in which the app will run. This might be a specific device, a device type (smartphone, tablet, hybrid) or a set of devices. This can be used to "simulate" the situation for which an app is developed. This can be a single type of device or a set of devices D from DS. The selection will automatically switch off any tests that do not apply to these devices. This functionality can be used to not overload the developer with an unnecessary amount of messages.

3.5 CONFORMANCE TESTING REQUIREMENTS

In conformance testing a layout instance L_I is matched against a set of predefined rules defining constraints on the instance.

3.5.1 *Specifying constraints on layout instance (CRQ1)*

A method is needed of specifying constraints on layout instance components. This set of constraints can range from constraints on a single UI component e.g. specifying that a text has a minimum font size to combinations of UI components like specifying that a text element on a button must fit within the bounds of this button and thus must be completely visible to the user. The rule engine must support the setting of constraints on multiple possibly nested UI components.

3.6 UNIFORMITY TESTING REQUIREMENTS

When testing for uniformity in mobile applications, we mean we expect a uniform interface across different device models and OSes. This problem can be tackled from different perspectives: Comparing results from different dynamically obtained layout instances or predicting whether a specific layout specification will generate different layouts on specific devices.

3.6.1 *Automatic Device Configuration (URQ1)*

If we want to predict errors, given a layout specification, we need to know the properties for the devices that the app will run on. For this it is preferable that configurations can be generated automatically given a device. For this to be functional and usable it is necessary to include a device configuration generator that generates a configuration profile from an existing device for our layout testing framework.

3.6.2 *Comparison (URQ2)*

Different devices and OSes have different hardware properties that can cause a layout to look very different at the pixel level. If we look at it from a higher level we might say that a layout is "the same". It is all in the interpretation of what we consider to be the same. When we have a layout with one button that stretches the whole width of the screen, this button will be a lot larger on a tablet than it will be on a smartphone. If this tablet is running a theme that provides different colors to standard UI components the button itself might look different too. Regardless of these changes we could consider this layout to be the same. Our layout testing framework would need the

ability to compare layout specification or instances and set rules for what we consider equivalent. The set of these rules is basically the implementation of *diff* function (Equation 2). To be able to compute *diff* we need a diff function for each component in LS. This means we can specify it as follows:

$$\mathrm{diff}(LI_1, LI_2) = \frac{\sum_{n=0}^{n=\#components} \mathrm{diff}(C_n \in LI_1, C_n \in LI_2)}{\#components} \quad (4)$$

3.6.3 *Prediction (URQ3)*

With prediction we want to know in advance whether a single layout specification will cause problems on specific devices or OSes. This can be done in two different ways: Indicate potential problems in general or with respect to a specific device configuration. An analysis of a layout specification in combination with a specific device configuration generates a report on any problems that might occur in this combination. When multiple device configurations are available a report can be generated stating on which devices the layout will cause problems or unexpected results.

3.7 SPECIFICATION STANDARD CHECKING REQUIREMENTS

3.7.1 *Specifying constraints on layout specification (SRQ1)*

A method is needed of specifying constraints on layout specification components. Certain components must adhere to certain properties to be usable and functional. The layout framework must supply a method of composing rules on elements that can occur in a layout specification. These constraints may contain meta-data that specify OS, platform and device type. This is needed to include functionality to encompass specific device peculiarities.

3.8 REQUIREMENTS SUMMARY

When summarizing all the previously mentioned requirements we obtain the list in table 1.

Requirement	Identifier
General Requirements	
Coverage	RQ1
Platform independence	RQ2
Device independence	RQ3
Automation	RQ4
App coverage	RQ5
Error Reporting	RQ6
Environment Selection	RQ7
Conformance testing requirements	
Specifying Constraints on layout instance	CRQ1
Uniformity testing requirements	
Automatic Device Configuration	URQ1
Comparison	URQ2
Prediction	URQ3
Specification standard checking requirements	
Specifying constraints on layout specification	SRQ1

Table 1: Requirements overview

3.9 DISCUSSION

Now that we have a set of requirements that our generic layout testing framework must comply with and a set of requirements for specific types of testing, we are going to make a selection based on what is doable and feasible in general and within the time requirements for this thesis. Some requirements can be implemented partially. Here we describe this selection for each type of testing in combination with the general requirements for each testing type

3.9.1 *Conformance Testing*

In conformance testing, rules are applied to the *layout instance*. To comply with platform and device independence (RQ2 and RQ3) these rules would need to be stated in a generic way and operate on a generic layout instance. To be able to create a generic view of a layout instance we need to know what is on the screen. For this functionality

screenshots could be used, but as mentioned in Chapter 2 determining what is on the screenshots is not a trivial task.

A better solution would be to convert screen information into a textual format in which the components are described in a similar way to a layout specification (Section 4.2.1.2). Testing frameworks often use an object representation of UI elements to make changes and assertions about information on the screen. If we were to use these components to generate a dynamic tree we could possibly automate the extraction (RQ5). We probably have to compromise on app coverage (RQ6) depending on the methods used to traverse the app and all layout instances (Section 4.2.1.1). These steps are further analysed in Section 4.2.1.

3.9.2 *Uniformity Testing*

Uniformity testing has a number of problems that constrain this type of testing. Coverage of all layout elements in combination with platform and device independence is a huge task. This means an implementation for each possible layout component that can exist on any operating system. Of course most OSes have a lot of common components, but we need to abstract from them to create a uniform testing interface. Whereas this is complex task, we do analyse the possibilities of comparing layout instances as extracted for conformance testing in Section 4.2.2.

Predicting (URQ3) where and when an error occurs means having knowledge about the layout engine that is being used. Providing both platform independence and device independence creates the necessity to have all these engines available. Apart from the layout engine we also need additional information to fill in the layout specification. In most apps data is obtained from databases or other network locations. We need this information to create the final rendition. This data is retrieved in code and not in the layout specification. To be able to find out what data can possibly be contained in certain UI elements in the specification we need to do a symbolic execution of the code to find paths were references to UI components are used to change properties of these components.

As we see, prediction is a very complex and hard task. The implementation of the aforementioned may take years and even then we still do not have everything that is needed. We conclude that predicting errors without any form of dynamic execution is a lot of work with possibly little result. For this reason we chose not to analyse this path further within the scope of this thesis. How this part can be researched further is mentioned in Chapter 5.

3.9.3 *Specification Standard Checking*

In specification standard checking we want to check constraints on the layout specification. To create platform and device independent (RQ2 and RQ3) constraints a conversion is needed from either the platform specifications to a generic specification or from generic constraints to platform specific constraints. Either way, to fulfil the coverage requirement (RQ1) the conversion needs to hanlde every possible specification element. Automation (RQ4) can be achieved by automating the conversion process. To achieve app coverage (RQ5) we want to test the possible instances of all UI elements in an app. As mentioned in the previous section on uniformity testing, a lot of work has to be done to obtain the possible instances of UI elements. If we are able to populate static information with dynamic information we can at least be sure that the data is an actual instance that the app generates. We will analyse this option further in Section 4.2.3.

3.10 CONCLUSION

We conclude that we are going to analyse further, the conformance testing requirements and standard specification checking requirements (CRQ1 and SRQ1) in combination with the general requirements. Because a lot of knowledge is needed for each specific platform to be able to create platform independent adapters for all of them, we are going to focus on a single platform (Android) keeping in mind that we want to be able to use it for other platforms as well. For the other general requirements we try to fulfill them as best as possible.

4

ARCHITECTURE OF A TESTING FRAMEWORK

In the previous chapter (Chapter 3) we discussed and selected a number of requirements to which we want a generic layout testing framework to comply. In this section we take the selection of requirements, analyse them and work towards a solution of implementing these requirements. For the requirement Platform Independence (RQ2) we chose to focus on the Android system. We therefore start by analysing how the Android layout system is implemented and which elements of this system impose restrictions on the requirements. Detailed in this section is the layout specification (1.4.1) for Android which is either an XML-file or Java program. The grammar of both the XML layout resources and Java UI API together form the specification of all possible layout specications LS. This is followed by an overview of the architecture and implementation of a testing framework for Android with the remainder of the requirements.

4.1 ANDROID LAYOUT FRAMEWORK

An Android app is programmed in the Java language. An Android layout can be defined in Java or a layout resource file. A layout resource file is an XML file specifying the layout components[1]. The Android Layout Framework consists of a range of standard interface widgets called Views[2]. By grouping these components different layouts can be made.

4.1.1 *Layout composition*

An Android user interface is built up from a collection of View and ViewGroup objects[1]. A View is an object that is represented by a graphic on the screen with which the user can interact. A ViewGroup is an object that holds other View and/or ViewGroup objects. A ViewGroup defines how the children of the ViewGroup need to be aligned. A ViewGroup is also called a Layout Manager because it manages the arrangement of views that it contains. Figure 4 shows an example of a view hierarchy containing two ViewGroups and five Views.

4.1.2 *Layout Specification*

Layout objects can be created in code by creating new View and ViewGroup objects and adding them to existing ViewGroups. However, the easiest and most effective way of defining a layout is with XML.

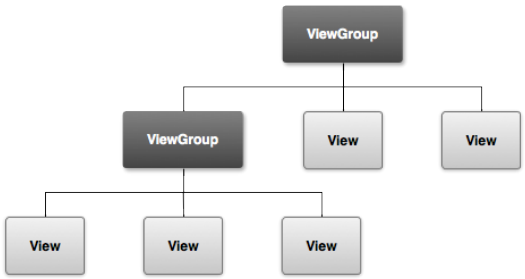


Figure 4: Android View Hierachy *(Image from Google)*

Each ViewGroup is defined as a single XML-element that can contain other XML View or ViewGroup elements. Listing 3 shows an example of a specification in XML. In Android this is called a *layout resource*.

Listing 3: Android XML layout specification example

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/
    android"
              android:layout_width="fill_parent"
              android:layout_height="fill_parent"
              android:orientation="vertical" >
    <TextView android:id="@+id/text"
              android:layout_width="wrap_content"
              android:layout_height="wrap_content"
              android:text="I am a TextView" />
    <Button android:id="@+id/button"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="I am a Button" />
</LinearLayout>
```

4.1.3 *Layout Specification Coverage*

The larger part of UI components that the Android framework provides can be defined in XML-files but it does not allow for a complete specification of the layout. There are some components that need some additional Java-code to be used. These are described here.

4.1.3.1 *Tabs*

The positioning of the tab bar and the layout of the tab content can be defined in the specification but the actual adding of tab content to the tab bar has to be done in Java.

directory	Specification Type
drawable	images
layout	layout definitions
menu	menu configurations
values	string, integer and array values

Table 2: Android resources directory structure

4.1.3.2 *Action Bar*

Like the Tab bar, the individual action bar items can be specified in a separate XML-file, but the linking of this bar to an Activity has to be done in Java.

4.1.3.3 *Dynamic content*

This is not an actual component but represents content that is not known until the app is actually running because it is either generated or has to be retrieved from a database or other resources. The XML-files can still give an indication of which data goes where, but the actual content might not always be known before runtime.

4.1.4 *Themes and defaults*

Although the XML-files specify properties to customize UI-components, not all properties of each XML-element are commonly specified. When a TextView is created with only the text property set, default values will be used which are inherited from the default theme of the system like font size, type and color. If properties are not set explicitly they are determined by the runtime environment. Even if properties are set explicitly it is still possible for them to be changed at runtime. This might happen when property values are contradicting.

4.1.5 *Layout Specification Categorization*

The Android system lets developers specify all resources for an app in the 'res' folder. This folder contains the following resources: Each of these properties represents a directory. The property names can be appended with properties that specify hardware or software properties. An example property is the screen size (small, normal, large, xlarge) or platform version (v11,v12 etc.). Android selects the appropriate resources for a specific device at runtime, based on the values of these properties. Table 3 shows some example device configurations.

Name	Description
drawable-large-v15	image resource for large screen API 15
layout-small	layout specs for small screens
values-en	values devices set to english

Table 3: Resource configuration examples

4.1.6 *Layout Construction*

When all layout resources are loaded, the actual calculations for the placement of components on the screen can be executed. Different layout components like *LinearLayout* will determine the ordering of UI-elements whilst the screen size bounds the placement and size.

The categorization properties mentioned above are selected before the layout instance is constructed. They are needed to select the correct layout specfication. When a resource is requested, first, the appropriate layout directory is located and the the properties of the available layout specifications are compared to the properties of the current device. For example, a layout is requested for a specific View:

1. The layout directory is located, there is a layout-normal and layout-large directory (only screen size property is defined)
2. Now, the screen size of the device is compared to these properties, if it is large, the large directory is selected; If it normal, the normal directory is selected.
3. Any resources within the layout definition will be handled like steps 1 and 2.

The best match system uses the properties as described in the section layout categorization. Android specifies a large number of properties which are all handled in a specific order.

4.1.7 *API Level*

Android has gone through a number of revisions bringing the total up to 16 API versions[19]. Surely some of the older levels (1-4) are almost not used anymore and it would not make much sense developing a new app for these versions. Although they might still be needed for existing apps that need maintenance. On the other hand API levels 5 and up still have a wide spread of use as can be seen in figure 5. More than 50% is still using Gingerbread which covers API level 9 and 10. Together with Eclair (and lower) and Froyo (API level 8 and lower) that still makes up about 75% of all devices. Fortunately most

components have stayed consistent along the different versions of Android but there have been some additions. Below a list of changes is presented from different levels that can affect the look and behaviour of an app directly.

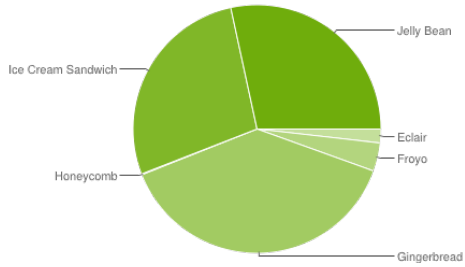


Figure 5: Android Version Chart: Collected from Play Store for 14 days starting on May 1, 2013 *(Image from Google)*

4.1.7.1 *Android 2.1.x (level 7)*

The framework now supports multi-touch input on the touchscreen.

4.1.7.2 *Android 2.2.x (level 8)*

New configurations for car-mode and night-mode. Lets an app display a different UI to the user based on the time of day or if the phone is sitting in a car dock.

4.1.7.3 *Android 2.3 (level 9)*

Change of standard theme. Menu's used to have a gray background with black letters, now they have a black background with gray letters. Something to take into account when developing graphics and choosing colours.

4.1.7.4 *Android 3.0 (level 11)*

Update for Tablets: The UI-Framework has been updated with components for better tablet support. These components are called Fragments and are in some ways the Android equivalent of browser frames. Action Bar: A menu bar that stays at the top of the screen. It can act as a tab bar but can also contain other components like checkboxes. Animation Framework: Lets users animate standard UI-components

4.1.7.5 *Android 4.0 (level 15)*

New standard font and colours (introducing the Holo Theme) New Layout definition: GridLayout. Allowing components to be placed in a grid whilst maintaining a flat layout structure. New TextureView component. Lets developers integrate OpenGL textures into the standard Android-layout.

4.1.8 *Discussion*

The basic structure of an Android layout specification and instance is well defined. The most prominent problem with the layout specification is that not all UI elements can be specified and it is missing dynamic content. To be able to detect layout errors, dynamic information is necessary.

The point of the best match system is to enable developers to specify different layout specifications for different devices. On the one hand this system makes it easy for the developers to create different resources for different devices. They just have to be placed in a directory with the correct naming and the rest is done by the system. On the other hand, adding more specifications means more maintenance. If we create two different specifications for a complete app, this means we now have to maintain twice as many specifications. Since the Android layout system is built with dynamic sizing in mind, it is far more desirable to create a single specification that can adapt well to different sizes over creating a lot of different specifications for each device.

Having different API-levels in use at the same time means that certain components may work for some API-levels and not for others. This is something that can be detected before runtime and is checked by tools like LINT[9]:

```
res/layout/switches.xml:35: Error: View requires API level 14
(current min is 4): <Switch> [NewApi]
    <Switch android:id="@+id/monitored_switch"
```

When a component exists in the layout instance, this means that it is supported by the OS.

When combining all of the above a layout instance is generated. From this instance we want to able to tell what is good and what is bad. The main issue with layouts is that something can be bad in one layout and good in another. It depends on the context. Although we can reason about issues that would be probably be considered bad in every context.

We are going to assume that we have a single layout specification for each screen. We do however want to be able to support different API-levels.

4.2 OVERVIEW OF TESTING FRAMEWORK

Now that we know how the Android layout system works we need to design a system that is able to analyse this framework keeping in mind the selection of requirements from the previous chapter. In this section we describe the design and implementation of a layout testing framework. We use a combination of existing tools with some additional components to bring it all together. In each separate component we try to achieve layout specification coverage, platform independence, device independence, automation and app coverage. Figure 6 shows an overview of the architecture of the framework.

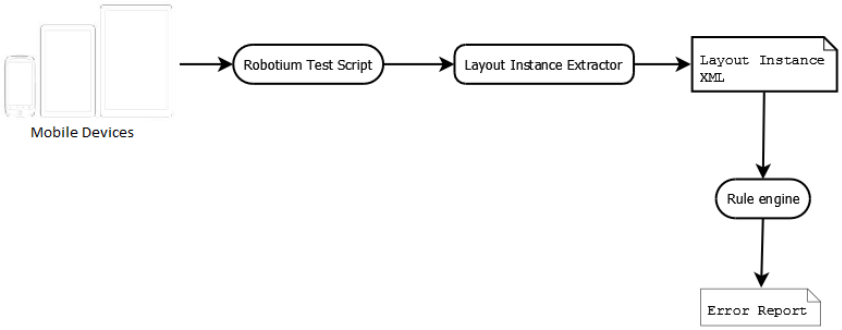


Figure 6: Architecture Android Layout Testing Framework

The following paragraphs describe the design and implementation of the separate components.

4.2.1 *Conformance Testing*

In conformance testing we want the layout instance to conform to a set of rules. If we think about the rules that we want to apply to a layout to be usable and functional on a high level we might think about:

- Buttons need to be large enough for a users fingers
- Text on components that are not resizable need to be fully visible
- Text fonts need to be large enough to read

These are high level definitions of rules to which we want the layout to conform. When performing manual inspection of a layout e.g. by using an app or looking at screenshots from automated tests, these

are the kind of rules that you might want to inspect. Because Automation is an important requirement for the framework and these rules can not easily be used to automatically detect problems on the screen, we need to perform some additional transformations to the layout instance. A formal representation of an instance L_I is needed so formal rules can be applied to it which in turn is needed to compute the function *diff* (2).

To obtain this formal representation we chose to convert a layout instance to an XML file on which we will apply rules in a similar way to standard specification checking. This XML file is similar to the layout specification but it is annotated with dynamic information which is obvious because L_I is an instance of L_S. We can consider L_S to be the program source and L_I to be a single exeuction trace of this source.

To accomplish this and incorporate automation we need components that can a) traverse an app like a human does, b) extract all layout components and c) validates this extraction against a set of rules. We want to achieve some level of automation on these three components.

4.2.1.1 *App traversal*

Most apps consist of multiple screens which can appear in different instances depending on external data or user input. In order to automate the capturing of different instances, some form of automated traversal is needed.

There are a number of tools that can automate the traversing of an app. What is important to us, is both the traversing and the extraction of the View hierarchy as mentioned in section 4.1 on which rules will be applied. This hierarchy represents the *layout instance*. To comply with requirements for device and platform independence (RQ2 and RQ3) we are looking for a solution that is able to run on devices independent of their API level and device type.

The Robotium[12] testing framework is able to provide us with this functionality. The user still needs to write a script that performs user interactions although these scripts can also be used to perform functional testing. This means that App Coverage (RQ5) will mainly depend on the writer of the test scripts. At certain points t in the scripts, a method must be called that performs the layout instance extraction. The implementation of this method is explained in the next section.

4.2.1.2 *Property extraction*

There are two types of properties that we want to extract: Device properties and layout properties. We need to extract the view hierar-

chy and along with each view the properties that we are interested in. These properties consist of at least the position and size for each view component. We can extract extra properties that are specific for a view.

The selected tool for app traversal, Robotium, also provides methods for iterating over the Android view hierarchy. We built a new component that is able to iterate over this hierarchy, extract the properties and output them as an XML file. This component is callable from a Robotium script by a single method call. The output file will contain a layout instance for a specific screen as can be seen in the following listing (some properties have been left out for readability):

Listing 4: "Example layout instance file"

```
<?xml version='1.0' encoding='UTF-8' standalone='yes' ?>
<LayoutInstance>
  <Device>
    <APILevel>10</APILevel>
    <Density>
      <classification>240</classification>
      <fontScale>1.5</fontScale>
      <xdpi>254</xdpi>
      <ydpi>254</ydpi>
    </Density>
    <dpi>254</dpi>
    <width>480</width>
    <height>800</height>
  </Device>
  <Layout>
    <LinearLayout id="0"
     x="0" y="0" w="480" h="800" visible="true">
      <FrameLayout id="1"
       x="0" y="38" w="480" h="37" visible="true">
        <TextView id="2" text="TextviewScroller" fontsize="21"
         x="6" y="39" w="468" h="34" visible="true" />
      </FrameLayout>
      <FrameLayout id="3"
       x="0" y="75" w="480" h="725" visible="true">
        <LinearLayout id="4"
         x="0" y="75" w="480" h="725" visible="true">
          <Button id="5" text="Button1" fontsize="21"
           x="0" y="75" w="120" h="73" visible="true" />
        </LinearLayout>
      </FrameLayout>
    </LinearLayout>
  </Layout>
</LayoutInstance>
```

We can see the similarity with the layout specification. The layout instance is annotated with dynamic information. The file starts

with the device specification containing the device specific properties: screen size and depth, API level and font scaling. We consider this the context in which this specific instance L_I was generated from L_S.

4.2.2 *Uniformity Testing*

We chose to analyse the comparison requirement further (URQ2). To be able to specify equivalence rules, we first need a formal way of specifying the instance which we have now. To be able to compare two instances we need to convert higher level reasoning about what we consider to be equivalent to formal specifications. The problem with this is that what we consider to be equivalent again depends on context. To make a useful comparison of layout specifications or instances, a function is needed that can give us a measure of how similar two components are i.e. returns 1 when two components are exactly the same, 0 when we have different component types and anything in between depending on the values of the component properties. There are a number of issues that need to be addressed in constructing such a function.

A component on a different device with a slightly different screensize might have slightly different properties e.g. 1 pixel larger. We still would consider these components to be visually equal. Comparing properties exactly does not work for this reason. There must be some range in which we still consider properties to be equivalent.

Even if we give this comparison some range, it is still possible that the properties vary too much, but we would still consider them being the same. An example of this is a button that strectches to the full width of the screen. When we change orientation of the device it is possible that the button will be almost twice the size.

What we learn from this is that we must not compare properties on their own, but relative to the device specifications. What we need to do is compare component A from L_I1 to component A from L_I2. For this it is necessary that we obtain the instance in the same state of the app on different devices.

The function is not the same for all UI components. Different UI components might have different properties that need different methods of comparing. To get full coverage we would need to implement comparison rules for all these components.

A layout instance has a hierarchical structure for which the internal nodes represent layout elements that describe the positioning of child

elements. The leaf elements contain UI components like buttons or text fields. It is on these leaf elements that we want to define a similarity function. Because each element has different properties we need to have specific functions for each element. We need to apply the specific similarity function for each pair of components $C_1 \in LI_1$ and $C_2 \in LI_2$. For this the we need to know which component in LI_1 and LI_2 results from the same specification. We can achieve this by using an identifier called "Content Description" which has to be added in the Android specification file. The XPath expressions used in the rule engine (4.2.3.1) can be used to select components of a specific type e.g. Buttons in LI_1. By using the unique identifiers of these components the corresponding component can be tracked down in LI_2 and be compared using a specific similarity function for that component. To be able to say how equivalent these instances are, each component needs to be compared.

We did not implement this solution further because of time constraints.

4.2.3 *Specification Standard Checking*

Given a layout instance file generated by the property extraction, a rule engine is needed that is able to take this file, apply the rules and output any warnings. In this case we do not check the layout specification standards but the layout instance conformance standards.

4.2.3.1 *Rule engine*

When combining the app traversal with property extraction we can obtain a layout instance at different steps in the traversal. To perform automated validation we need to be able to check one of these trees against a set of rules.

A rule defines constraints on a UI component or a combination of UI components. If an XML instance file does not adhere to one of these rules an error will be generated. So what we need is a way of specifying the set of components C_S from the domain of LS to which a specific rule applies. Then we need a way of specifying constraints.

To encompass all of this we developed another XML-format specifically for the rules:

Listing 5: "Example layout instance file"

```
<Rule>
  <Name>Enforce minimum width on buttons</Name>
  <Path>//Button</Path>
  <Return>NodeList</Return>
  <Error>This button is not large enough</Error>
  <ConstraintList>
    <Constraint>((w / dpi) * 2.54) > .7</Constraint>
  </ConstraintList>
</Rule>
```

For the selection of C_S we chose to use XPath[16] which is a query language that enables easy selection of parts of an XML file. We chose XPath because it enables us to both easily select sets of components as well as very specifically select a component with an easy syntax. To create constraints for components we introduced the *ConstraintList*. This list can contain boolean expressions that say something about the component. Each of these constraints is evaluated by the rule engine. They must all evaluate to `true` for this rule to comply. The rule engine intially loads the device specific properties and all properies for the selected component. These can all be used in the constraints. When the XPath expression selects multiple components the rule is applied to each component. This makes it very easy to create global rules for each UI component.

To comply with the coverage requirement (RQ1) rules need to be implemented for all UI components within the Android UI Framework.

There are however some limitations to this way of specifying rules. Once the elements are selected we do not have any control over the way they are returned. We obtain a list with elements that match the expression. If this query would for some reason return elements of different types they will all be returned in a single list without further context of the document. This means that we can only apply our constraints to the selected element. Also, tracking the element back to the original file can be a problem. Because our extraction component stores a unique id with each element, we can easily track back the original position in the file.

4.2.3.2 *Error Reporting*

Since the rule engine applies the rules to the instance files, it is the rule engine that detects errors. The rule file contains an element to specify an error message. Although we can simply output this message it does not indicate the actual problem very well. It would be nice to visually indicate the area of the problem. For this functionality we take a screenshot together with the extraction of the instance.

We can use this screenshot together with the positional properties of each element in the instance file to visually indicate the component on which the error occured.

4.2.4 *Automation*

Now that we accomplished some level of automation on the individual components, we want to bring it all together. Because we can automate the app traversal and layout extraction as well as the application of rules, a script can be used to call the individual components to automate the entire process.

4.3 CONCLUSION

In this chapter we described the Android Layout Framework as well as the implementation of our Layout Testing Framework. The requirements for specifying constraints on layout instance (CRQ1) were fulfilled whereas the general requirements are all implemented partially with automation being the most prominent one. The requirement coverage (RQ1) can achieve a higher level within this framework with implementation of rules for all components. The next chapter will discuss the results of some implemented rules.

5 RESULTS AND DISCUSSION

This section describes the results obtained from the layout testing framework described in the previous chapter. This chapter starts with an overview of the setup of the experiments. This is followed by a number of examples, each containing a rule and output. Finally, the results are discussed with respect to the requirements.

5.1 EXPERIMENT SETUP

The setup for experiments consists of an Android device which is connected to a computer system containing the Robotium[12] test cases, rule engine (4.2.3.1) and error reporter (4.2.3.2). We performed the experiments with two devices: An HTC Desire Z smartphone running Android 2.3.3 (API 10) and a Asus Transformer TF101 tablet running Android 4.0.3 (API 15). As stated in the previous chapter (Section 4.1.7) Android 2 and 4 are still quite common and the difference in screen sizes provides a good environment for obtaining different layout instances. In this chapter we shall refer to these devices as phone and tablet respectively for readability.

We created a sample app for these experiments for which a layout instance can be seen in figure 8 and 7. The sample app contains a single layout specification. The layout specification is tailored towards a tablet size which is clear from the appearance on the phone. Although, the tablet version presents other problems.

A single run of a test is initiated by a batch script. The script starts a Robotium test case that runs the app on the connected Android device, extracts the layout (as described in the previous chapter) and makes a screenshot. The layout instance and screenshot are copied to the desktop system. Here, the rule engine is started and applies the rule that whas given as an argument to the script. Any errors are automatically indicated in red on the screen shot.

We perform this procedure for each of the rules below for both phone and tablet. Each example is accompanied by screenshots from the devices and the results of applying the test rules. The results are then discussed with respect to the requirements.

5.1.1 *Rules*

Each rule is specified in an XML-file as explained in section 4.2.3.1. A rule can have two return types: `NodeList` and `Number`. The former being the default value. It returns the set of components selected by the XPath[16] expression in `Path`. The return type `Number` can be used in combination with the XPath-function `count()` which provides the ability to count the number of elements returned by the XPath expression.

5.1.2 *Selecting components*

To demonstrate the use and functionality of the rule engine we start by showing how elements or sets of elements can be selected with an expression. As mentioned in Chapter 4 we use XPath expressions to select the elements to which the rules should be applied. An example rule:

```
<Rule>
 <Name>Selecting all buttons</Name>
 <Path>//Button</Path>
 <Return>NodeList</Return>
 <Error>Button</Error>
 <ConstraintList>
  <Constraint> <<Javascript Expression>> </Constraint>
 </ConstraintList>
</Rule>
```

The expression in this rule selects all buttons in the layout instance. We can see them marked by a red rectangle in figures 8 (phone) and 7 (tablet). The images shown in this section are generated by the tool. This illustrates the ease with which we can target an element or group of elements.

The element `ConstraintList` can contain zero, one or more constraints. A constraint must be a valid javascript boolean expression. The expression may use variables that are attributes of the elements that where selected in `Path` or one of the global variables `dpi`, `width`, `height` or `api`. Each constraint will be applied to the currently selected element. If a set of elements is selected by the path expression each constraint in the list will be applied to each element separately.

Figure 7: Selection of all Buttons on tablet

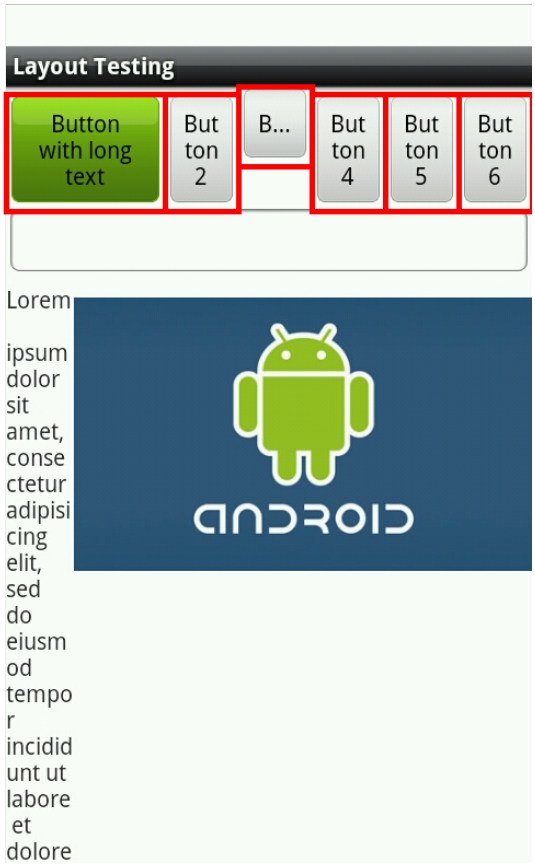


Figure 8: Selection of all Buttons on phone

5.1.3 *Counting elements*

The rule engine also has the option to count elements in a set. This can be achieved by using the XPath `count` function. If we want to count all buttons in a layout instance we can use the following expression:

```
<Path>count(//Button)</Path>
```

The resulting value is the number of elements that are returned by the expression.

5.1.4 *Error Reporting*

When a constraint of a rule is not met, the error associated with a rule is displayed in the console along with the ID of the component and the specific constraint or constraints that caused the error. The ID can be used to track the component back in the XML file. Finally, the element is indicated on the screenshot by a red rectangle. For these examples we chose to apply one rule at a time for clarity.

5.2 RESULTS: EXAMPLES

In this section we are going to demonstrate the features of this framework with a number of examples. Each section is devoted to a specific rule and starts by explaining a problem, followed by a rule and its output. Finally, the results are discussed with respect to the requirements.

5.2.1 *Size constraints on UI components*

The size of touch areas in an app is very important for the usability. For example, buttons that are very small can be very hard to initiate and if they are in the vicinity of other touchable elements it can lead to performing an unwanted action. To prevent these situations we want to specify minimum sizes for these components. By using the screen size and density in combination with the component size, we are able to calculate the actual size of a component on the screen. We use the following rule to state that a button must have a width and height of at least 0.7cm. The rule has two constraints that use the global variable `dpi` (screen density) to calculate the width and height in centimers. Also, a visibility constraint is added to ensure that the component is actual displayed. We use this constraint on each rule in this chapter. Although these constraints can be specified in a single constraint, we specify them seperately so we can check which constraint was not met:

```
<Rule>
 <Name>Enforce minimum width on buttons</Name>
 <Path>//Button</Path>
 <Return>NodeList</Return>
 <Error>This button is not large enough</Error>
 <ConstraintList>
  <Constraint>((w / dpi) * 2.54) > .7</Constraint>
   <Constraint>((h / dpi) * 2.54) > .7></Constraint>
   <Constraint> visible == true </Constraint>
 </ConstraintList>
</Rule>
```

5.2.1.1 *Output*

The rule did not cause errors for the tablet but did for the phone. We can see that 5 buttons did not meet the width constraint. The buttons are indicated in figure 9. The report in the console:

```
ERROR (id = 7 button2 ) : This button is not large enough
Failed(1):  ((w / dpi) * 2.54) > .7
ERROR (id = 8 button3 ) : This button is not large enough
Failed(1):  ((w / dpi) * 2.54) > .7
ERROR (id = 9 button4 ) : This button is not large enough
Failed(1):  ((w / dpi) * 2.54) > .7
ERROR (id = 10 button5 ) : This button is not large enough
Failed(1):  ((w / dpi) * 2.54) > .7
ERROR (id = 11 button6 ) : This button is not large enough
Failed(1):  ((w / dpi) * 2.54) > .7
```

In this case the rule targets buttons, but by using different path expressions the rule can be easily applied to other components.

5.2.2 *Number of text lines*

Depending on the design of our layout, we might not want texts on buttons to span more than one line. If this is the case it will change the size of the button or part of the text might not be fully visible. In our example app there is one button that has a longer text than the other buttons. Since we have set these buttons on a row, we want them to have equal hight. The following rule will generate an error when a Button has more than 1 line of text:

```
<Rule>
  <Name>Enforce single text line on buttons</Name>
  <Path>//Button</Path>
  <Error>This button has multiple lines of text</Error>
  <ConstraintList>
    <Constraint> lineCnt <= 1 </Constraint>
  </ConstraintList>
</Rule>
```

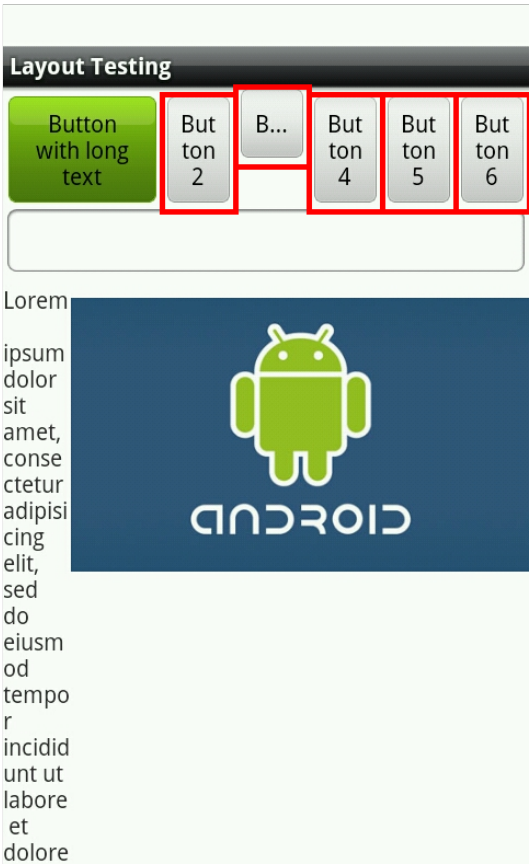


Figure 9: Button Size

5.2.2.1 *Output*

This rule does not generate any errors for the tablet. For the phone, 5 buttons generate an error as can be seen in figure 10.

The rule used here is applied to *all* buttons. We can however create a specific rule that is tailored towards a specific button row so we can still allow other buttons with multiple text lines. We can achieve this by specifically selecting the LinearLayout in which these buttons are contained by using its `uid`:

```
<Path>//Button[ancestor::LinearLayout[@uid='ll4']</Path>
```

5.2.3 *(Partially) unreachable components*

Some components might not be visible because they are drawn outside the visible screen. This only poses a problem if their parent views are not scrollable, meaning there is no way to get the view into the visibile screen. We can detect this by using a combination of the expressional power of XPath and constraints. We will present the example for TextViews although it is similar for other UI components. We select TextView elements that do *not* have a ScrollView ancestor. This

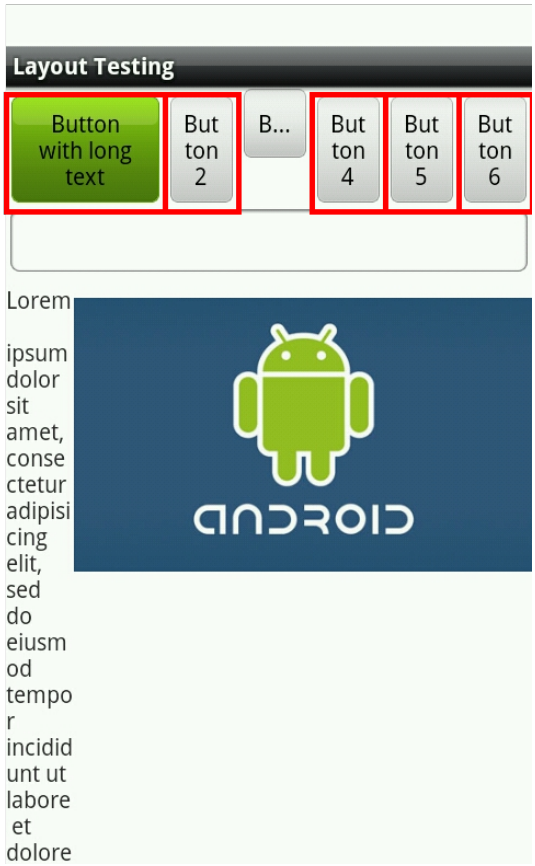


Figure 10: Rule for buttons having at most one line o f text

by itself is not a problem. We only want to obtain the elements that are not on the visible screen. If they are outside the visible screen *and* there is no ScrollView ancestor, the elements are unreachable. We use two constraints: One to detect if the element lies outside the visible screen and another one to determine if the text in the TextView is fully visible:

```
<Rule>
  <Name>Layout Ancestors</Name>
  <Path>//TextView[not(ancestor::ScrollView)]</Path>
  <Return>NodeList</Return>
  <Error>Unreachable TextView</Error>
  <ConstraintList>
    <Constraint> y < height </Constraint>
    <Constraint>(h / lineheight) >= lineCnt</Constraint>
  </ConstraintList>
</Rule>
```

5.2.3.1 *Output*

The tablet instance did not generate any errors. The result for the phone can be seen in figure 11. The console output for the Phone:

```
ERROR (id = 15 textview1 ) : Unreachable TextView
Failed(2):  (h / lineheight) >= lineCnt
ERROR (id = 18 textview2 ) : Unreachable TextView
Failed(1):  y < height
Failed(2):  (h / lineheight) >= lineCnt
```

It appears there is one textview that is completely outside the visible area. For this reason we do not see it on the screenshot, but we do see the error generated in the console. Moreover, we see that this components violates two constraints whereas the other one only violates one.

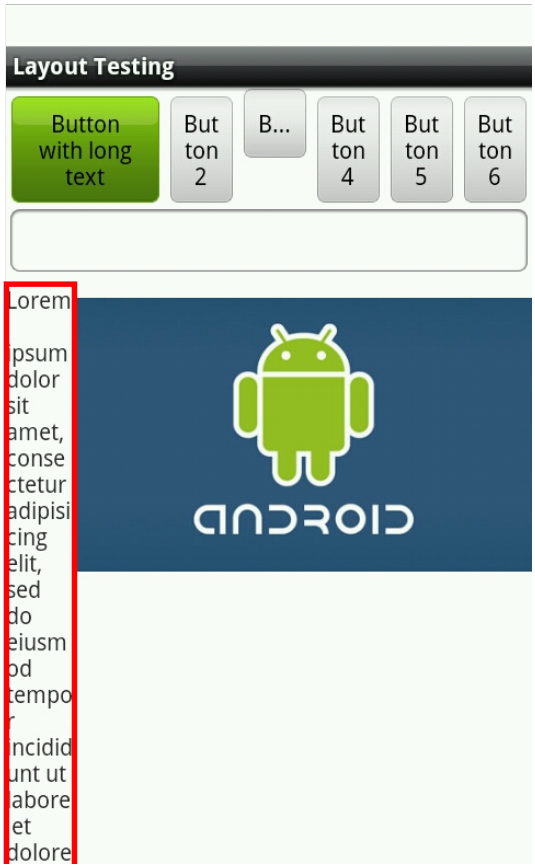


Figure 11: Rule for detecting unreachable TextViews

5.2.4 *Partially visible text*

In some cases components that contain text can not display all text that it is meant to display, because the parent component is constrained in size. This means that texts can be cut off and partially replaced by dots. We call this an ellipsized text. We do not always want this to happen. In this example we detect this by using the property `ellipsCnt` of a TextView. This property contains the number of

times text within a TextView is ellipsized. In this example we look for ellipsized texts in TextViews and Buttons. We want the `ellipseCnt` to be zero:

```
<Rule>
  <Name>Selects ellipsized TextViews</Name>
  <Path>//TextView | // Button</Path>
  <Return>NodeList</Return>
  <Error>TextView is ellipsized</Error>
  <ConstraintList>
    <Constraint> ellipseCnt == 0 </Constraint>
  </ConstraintList>
</Rule>
```

5.2.4.1 *Output*

Both phone and tablet generated errors on this rule. The result for the phone can be seen in figure 12 and tablet in figure 13. What is interesting here is that they generated errors on different elements: the title textview and button text. This is a good example of how problems are caused by a different environment. The tablet version also shows the menu items in the title bar (called Action Bar) which compress the title text of the app.

5.2.5 *Alignment of layout groups*

As can be seen in previous phone screenshots the button row at the top is not aligned but they are in the tablet instance. This is caused by the alignment property of the LinearLayout group in which the buttons are contained as well as different height values for the buttons. We would like to have the buttons inside a LinearLayout group to be aligned to the top of the group. To create a rule for this we make use of the power of an XPath expression to compare attribute values from different elements in the hierarchy.

```
<Rule>
  <Name>Check alignment of Buttons in LinearLayout</Name>
  <Path>
    //Button[ancestor::LinearLayout[@orientation='horizontal']
           and not(@y = ../@y)]
  </Path>
</Rule>
```

5.2.5.1 *Output*

This rule selects all buttons that are contained in a LinearLayout, which is layed out horizontally and where the y-value of the button does not match the y-value of the LinearLayout. The result can

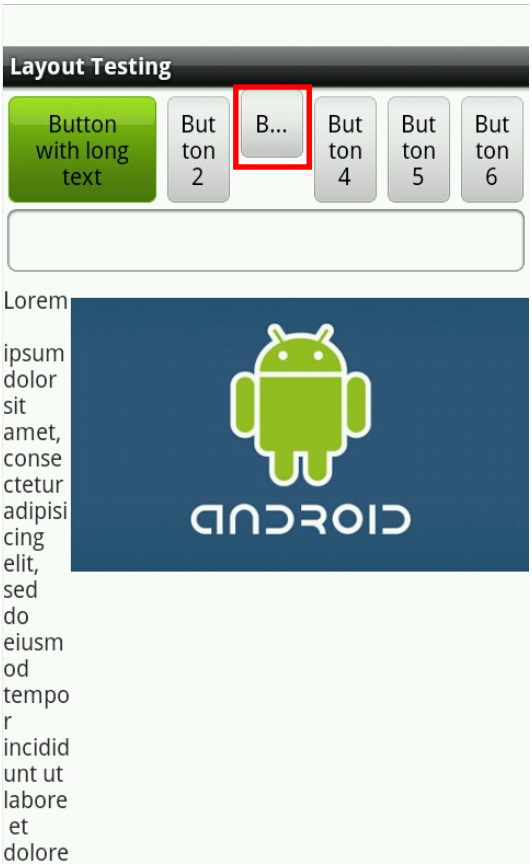


Figure 12: Rule for determining ellipsized text on phone

be seen in figure 14. This kind of rule can be used to test the functionality of specific layout instance elements. The rule specifies the behaviour we would expect from the LinearLayout group.

5.2.6 *Count number of items on the ActionBar*

The number of menu items on the actionbar depends on the width of the menu items, the available screen space and the visibility properties of each menu item. An item can be set to "showAlways" which means that items will be compressed when they do not fit. If this is not the case they will be moved to an overflow menu.

```
<Rule>
 <Path>count(//Button[ancestor::ActionMenuItemView])</Path>
 <Return>Number</Return>
</Rule>
```

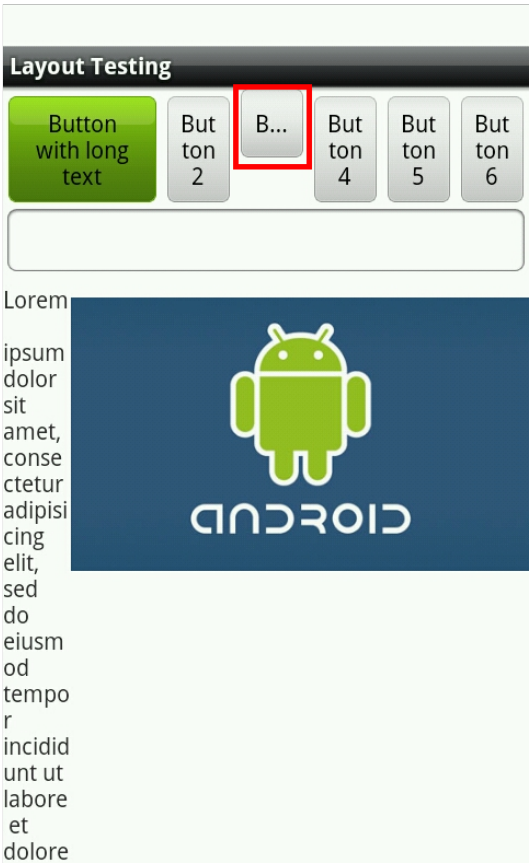


Figure 13: Rule for determining ellipsized text on tablet

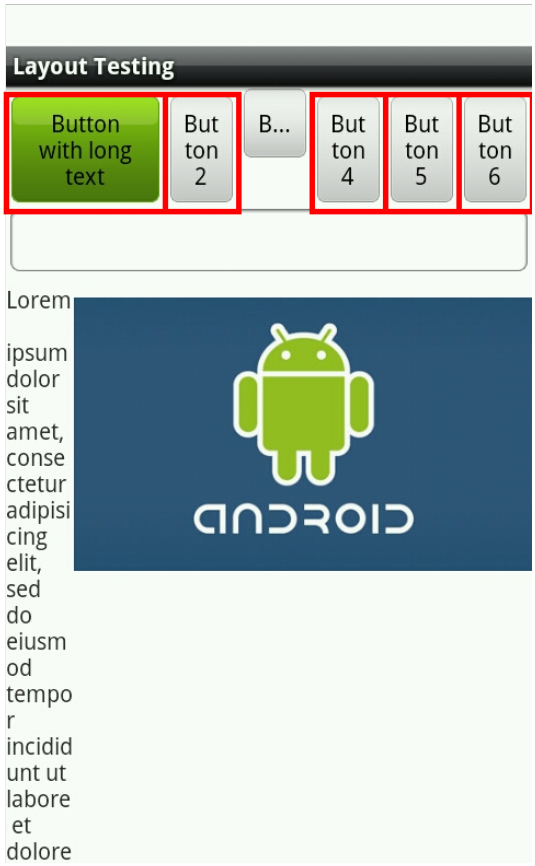


Figure 14: Checking LinearLayout Alignment

5.2.6.1 *Output*

In this case we see the result of the function in the console:

```
Result: 6
```

In this case the action bar contains 6 menu items. The components that where counted are also indicated by a red rectangle on the screenshot (Figure 15. If we would want to maximize the number of items we could add a constraint for a maximum value of the result.

Figure 15: Counting number of menu items on ActionBar

5.2.7 *Orientation*

Up until now we have used the same orientation for our tests: portrait mode. We can however rotate the devices to view the app in landscape mode. By running tests in both portrait and landscape mode we can detect whether changes in orientation cause more or less errors. Figure 16 shows an example of applying the rule for unreachable components (5.2.3) to the phone instance in landscape mode. We see that the text does still not fit on the screen and the error is indicated.

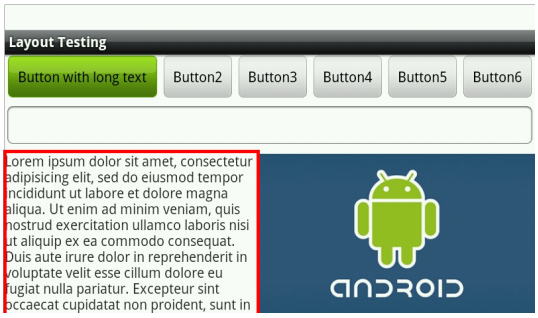


Figure 16: Rule for detecting unreachable TextViews (landscape mode)

5.3 APP FROM GOOGLE PLAY STORE: APPIE

The examples above were performed on a simple app that was created by us to show the capabilities of this system. However, the system can be used on any app with as many screens or elements as possible. The procedure for performing tests is similar to a single screen. The only difference is in the scripting. To demonstrate this we downloaded the "Appie"-app [20] from the Dutch supermarket Albert Heijn and used a script to extract the layout instance for multiple screens. The script performs a view clicks on menu items and texts. Multiple calls to `ui.grab()` extract the layout instance and a screenshot:

```
UIGrabber ui = new UIGrabber();
solo.clickOnButton(0);
ui.grab();
solo.clickOnText("Producten");
ui.grab();
solo.clickOnScreen(500,10);
ui.grab();
```

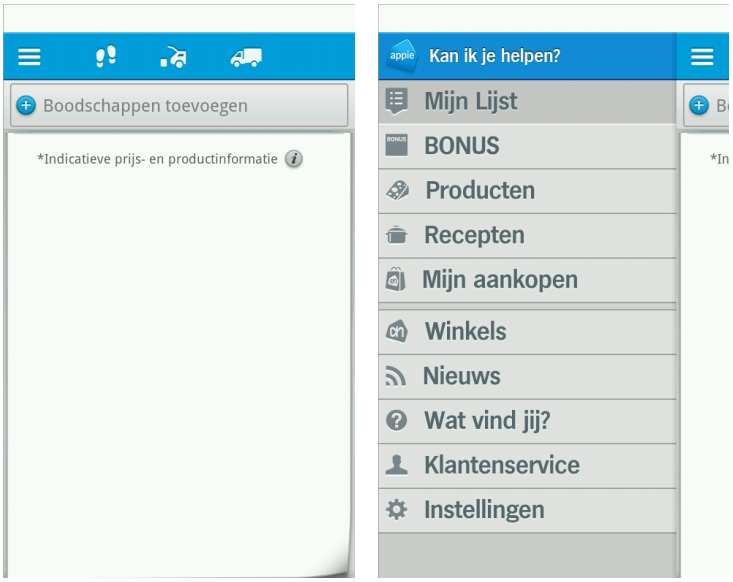
On each call to `ui.grab()` an XML file containing the layout instance and a screenshot is generated. These files are processed by the rule engine which produced the results in figure 17. We see that figure 17 (c) produced an error on the following rule:

```
ERROR (id = 9  ) : TextView is ellipsized
Failed:  ellipseCnt == 0
```

We also note that this app contains some custom elements. As we can see, the element still has the id, size and visible properties because these are inherited from the Android `View`-class:

```
<UpdateableImageView id="31" uid=""
  x="20" y="215" w="200" h="13
  4" visible="true" />
```

This is no problem for the current setup, but custom rules would have to be developed to test these elements.



(a) s (b)

(c)

Figure 17: Results from "Appie"-app
(a) no errors (b) no errors (c) ellipsized text

5.4 DISCUSSION

This chapter described a number of examples that were performed using two Android devices. We wanted to develop a rule set that could point out problems in a layout instance that can compromise functionality and usability of an app. Some rules, like the unreachable views (5.2.3), indicate a potential problem for functionality i.e. a view that is not or partially visible can not be seen or used. Although, this does not mean that we can not generate false positives. For example, a view is outside the visible area and not scrollable, while some additional code implements an animation that moves the view into the visible area when a button is pressed. We can not deduce this from the layout instance. It is usually specific to the app and not default behaviour for the standard components.

The main disadvantage of this method is the need to run the app on a device and writing the test scripts for it. On the other hand, different tests can be run at a later time once the views have been extracted. If at a later point in time we decide that we want to analyse different aspects of the instance and the program has not changed, we only need the instance files and there is no need to run the script again.

In the following section we discuss the resulting framework with respect to the requirements in Chapter 3.

5.4.1 *Requirements*

The rules used in the examples were specified using a self developed format for specifying constraints on layout elements. This fulfills the requirement CRQ1 (3.6.1). Although the focus in these experiments is on the layout instance, the rule engine can also be used to specify constraints on a layout specification (SRQ1 3.7.1). For Android it can be used easily because layouts are already specified in an XML-format. Again, the problem with the layout specification is that we are missing information to exactly point out layout issues.

In these examples we covered only a few elements that are available in the layout specification for Android (RQ1 3.4.1). But, it is easy to add elements to which the same rules apply by extending the path expression. In example 5.2.1 we specified a minimum width on Buttons. Most frameworks provide a lot more UI components that let the user interact with touch gestures. By simply adding expressions to find all these components in the instance file, they will also be covered. The problem is that having one rule for each component does not mean we have covered all potential problems. The rules in this chapter cover issues that either influence the functionality and usability of the app,

like component size (5.2.1), unreachable views (5.2.3) or partially visible texts (5.2.4), but also rules that have a more esthetic nature like the linear layout alignment (5.2.5). The latter would generally need to be tailored to the design of a specific app.

The rules in this chapter are not platform independent because they are written specifically for Android. However, the properties that they work on are so common for all platforms they can be easily used by targeting the correct elements in the path expression.
The layout extractor is the only component in this system which is platform specific. Both the rule engine and error reporter are completely platform independent (RQ2 3.4.2). However, the layout extractor is device independent. It was built by using API-calls that are available from Android API 1, meaning that it can be used on any Android device (RQ3 3.4.3).

The entire system can be automated once an app specific script has been generated. The writer of the test scripts only needs to incoporate calls to the layout extractor. Once this is done everything is automated from running the script on a device and extracting the layout instances and screenshots to applying the rules and generating visual reports (RQ4 3.4.4 and RQ6 3.4.6). In general, we covered a large part of the requirements that are important in specifying and identifying layout issues (CRQ1 3.5.1 and SRQ1 3.7.1). For the remaining requirements, except for uniformity testing, we have demonstrated a framework that is capable of overcoming their constraints, but mostly require a lot extra implementation work to become more useful. The next chapter discusses what has to be done to implement more of these requirements.

5.4.2 *Limitations*

In this chapter we have shown how we can specify constraints on sets of elements selected by an XPath expression. It is quite easy to select a set of elements specifying a UI component or with certain restrictions on the attributes. However, there are some constraints that are difficult or impossible to specify.

One of these is a relation or dependence on the existence of other elements. For example, if we wanted to select an element based on the existence of another element in the document.
Another type is that of setting constraints on a whole set of selected elements. We can specify that a certain expression must hold for all elements of a selected set, but we can not specify them being the same amongst eachother. This section showed an example of a LinearLayout having button child elements that were not on the same

row. We solved this problem by selecting the elements that did not have the same x-value as the layout container itself which will result in returning the children that have a different value. To be more precise we would like to say: Select all children of a LinearLayout if they do not have equal values for x. Because, we expect all children to be aligned. We can not specify an XPath expression that will return a LinearLayout if and only if the children not have an equal value for a specific attribute. We can easily specify that an ancestor or child of an element must have an attribute with a specific value or a value that matches the current element. It becomes difficult when we are talking about sets of children. Because our constraint rules only work on each selected element, we can only specify constraints within this element space, meaning the attributes of this specific element.

5.4.2.1 *Rule precision*

We need to keep in mind that false positives might occur. There are multiple reasons why false positives occur. First of all, if the view hierarchy does not match what we see on the screen. In our system, this happens when the operating system places views over the application under test. These are processes that we can not control. An example of this is a system dialog or the onscreen keyboard. Although it looks as if these are part of the app, they are in fact not and thus, do not influence our view hierarchy. This can also cause false negatives e.g. a view is partially obscured whilst this can not be deduced from the view hierarchy.

Another reason is that not all properties that may influence a component's appearance are taken into account. An example of this is the *visibility* property. A component might be present in the view hierarchy, but not visible on the screen. So, we need to specify constraints on each property that may influence the rule that we are creating. To do this, we need to make an overview of each property that might influence our rule, and then create constraints for each of them.

6 CONCLUSIONS AND FUTURE WORK

This section describes the general conclusions from this thesis as well as several proposals for future work. These proposals are based on the requirements that were not analysed further or partially implemented and problems that were encountered when implementing the framework.

6.1 CONCLUSIONS

The subject of generic layout testing is a very broad subject involving a lot of variables. From the outcome of the research on related work and tools we concluded that there was no standardised way of performing layout testing. In this thesis we identified the requirements that are necessary to perform layout testing for a generic testing framework. The analysis of these requirements suggested that implementing all of these requirements is a huge effort and each individual requirement needs additional research. To be able to say anything about a generic method of accomplishing this we first need to know how the individual components operate.

The conversion of a layout instance to an XML-format proved to be a good base for specifying and identifying problems. The XML-format makes it a lot easier to perform operations and also allows analysis without running the app, once the layout instances have been extracted. The advantage of this method is having the complete structure and properties of a layout instance at hand and easily traversable. The use of XPath expressions makes it easy to target a specific component or a group of components

The framework developed so far provides a base for expanding on layout testing. However, the rules that have been implemented so far need to be extended to provide support for additional layout problems using the information that is available in the layout instance file. When extracting the layout instance we might need to add more properties or precalculate certain properties to provide the rule engine a richer layout instance.

6.2 FUTURE WORK

This section describes possible future research topics. These topics are based on the requirements as described in Chapter 3. We propose suggestions as to how more subjects can be researched and implemented to cover more of these requirements. We cover these suggestions per requirement.

6.2.1 *General Requirements*

6.2.1.1 *Coverage (RQ1)*

At this moment coverage of all UI elements is achieved partially. We have created a partial coverage of the Android UI framework by creating rules that set constraints on these elements. To achieve full coverage of the Android framework, we need to implement rules for each component that exists in the Android framework. In this case, we do not consider custom components.

6.2.1.2 *Platform and Device independence (RQ3 and RQ4)*

One of the reasons the Android framework was selected as a base for further research and implementation was the large variety of devices running Android with different platform versions. It is a good candidate for experimenting with the requirements for a layout testing framework. To be able to abstract from this analysis and implementation at the OS level we need to create generic definitions for specifying layouts and then perform the analysis on this generic specification. Further research on creating generic layouts specifications would be a valuable addition to this framework.

The method of extraction with the Robotium framework proves to be adequate to obtain all properties of UI components on the screen. However there still is a discrepancy between these objects and the actual drawings on screen. More research on the extraction of layout information at runtime is needed to improve the quality of this instance extraction. We are able to extract all components from the app, but we can not extract everything that is visible on the screen. The reason for this is that some components are not really part of the app like the on-screen keyboard. They are overlays provided by the operating system.

Another subject for future work is matching up the layout instance file with the layout specification file. Comparing these two files can give insight into the conversion of specification to instance. It can provide information about the conformance of a layout instance to the specification. Does the specific layout engine do what we would

expect it to do? We experimented with this in one of the examples (Section 5.2.5).

6.2.1.3 *Automation (RQ4)*

In the results from analysis and implementation we covered quite a large part on the requirement of Automation. More work can be done on presenting a set of rules to the user. Now, the rules can only be applied one at a time. Also, automating this setup to run on multiple devices would be a valuable addition.

6.2.1.4 *App Coverage (RQ5)*

The app coverage requirement was not covered in this implementation and depends on the test scripts. If the test script does not perform the necessary steps to get to a specific instance, that instance will not be tested. A method for improving app coverage is model-based testing[33].

6.2.1.5 *Error Reporting (RQ6)*

At this moment error reporting is quite good in indicating the actual source of the problem. One thing that is missing is the link to the source. An id of the element is provided so it can be tracked back to the instance file which can be tracked back to a specification. However, this all has to be done manually. It would be nice if the error reporting would actually show the element in the layout specification.

6.2.1.6 *Environment Selection (RQ7)*

In this implementation we did not include environment selection. If we add extra information to the rules that specify device types for which this rules hold, we can combine this with information about the device. The device from which the instance was extracted is known. What is needed now is a selection of device types without the rule engine tester that only tests rules that conform to the selected device. This is an extra feature that provides the user with extra information that does not overload the user with rules that are not relevant.

6.2.2 *Uniformity Testing: Prediction and Comparison*

A section of the requirements that we did not include in the implementation are the uniformity testing requirements. Especially the requirements on comparison and prediction. A lot of work can be done on creating equivalence rules for layout components. When do we consider two layouts A and B to be equivalent? It is possible that layout A is extracted from device P OS X and layout B is extracted from

device L on OS Y. The important part is determining the properties of layout elements that need to be changed to consider something as beingt different. This also depends on the definition of different: They can be *visually* different and/or *functionally* different. Chapter 4 does provide an analysis of how this can be achieved (Section 4.2.2).

6.2.3 *Further suggestions*

When working on this master thesis Xamarin introduced the first version of Xamarin Test Cloud[17]. Xamarin is cross-platform development platform for apps. It enables developers to use a single programming language (C#) to develop apps for Android, iOS and Windows Phone whilst still being able to create native interfaces for each of these platforms. The Xamarin Test cloud continues this paradigm with the test cloud by enabling the writing of test scripts once for each and every platform. It would be valuable for the platform independece requirement to see if this system can provide the same information about the layout as Robotium is able to do. In this case it would only be necessary to write a single script and the platform independent requirement can be extended. Although, we might not be able to use this system on native apps.

BIBLIOGRAPHY

[1] Android layouts. http://developer.android.com/guide/topics/ui/declaring-layout.html, .

[2] Android ui. http://developer.android.com/guide/topics/ui/index.html, .

[3] Deviceanywhere. http://www.keynotedeviceanywhere.com/.

[4] Findbugs. http://findbugs.sourceforge.net.

[5] Graphical layout editor. http://developer.android.com/tools/help/adt.html.

[6] Android hierarchy viewer. http://developer.android.com/tools/help/hierarchy-viewer.html.

[7] Parasoft jtest. http://www.parasoft.com/jsp/products/jtest.jsp.

[8] Android lint. http://tools.android.com/tips/lint, .

[9] Android lint api checks. http://tools.android.com/recent/lintapicheck, .

[10] Less painful. https://www.lesspainful.com/.

[11] Motodev app validator. http://www.motorola.com/sites/motodev/library/menu_and_view_alternatives.html.

[12] Robotium: Android test automation framework. http://code.google.com/p/robotium.

[13] Testdroid. http://testdroid.com/.

[14] Utest. www.utest.com.

[15] Windows phone layouts. http://msdn.microsoft.com/en-us/library/windowsphone/develop/jj207042(v=vs.105).aspx.

[16] Xpath 2.0. http://www.w3.org/TR/xpath20/.

[17] Xamarin test cloud. http://xamarin.com/test-cloud.

[18] Android. www.android.com, .

[19] Google play store. http://source.android.com/source/overview.html, .

[20] Google play store: Appie van albert heijn. https://play.google.com/store/apps/details?id=com.icemobile.albertheijn, .

[21] Apple app store. http://www.apple.com/iphone/from-the-app-store/, .

[22] ios. www.ios.com/apple.

[23] Opensignal android fragmentation visualized. http://opensignal.com/reports/fragmentation.php.

[24] Google play store. play.google.com.

[25] Windows phone. www.windowsphone.com.

[26] X window system. http://www.opengroup.org/tech/desktop/x/.

[27] Ashish Aggarwal and Pankaj Jalote. Integrating static and dynamic analysis for detecting vulnerabilities. *Proceedings of the 30th Annual International Computer Software and Applications Conference*, 2006.

[28] International Organization for Standardization and International Electrotechnical Commission. Iso/iec 9126. www.iso.org.

[29] Dominik Franke and Carsten Weise. Providing a software quality framework for testing of mobile applications. *IEEE*, 2011.

[30] Rok Zontar Jernej Novak, Andrej Krajnc. Taxonomy of static code analysis tools. *Computer & Information Science (ICCIS)*, June 2010.

[31] L. Nagowah and G. Sowamber. A novel approach of automation testing on mobile devices. *Computer & Information Science (ICCIS)*, June 2012.

[32] Damith C. Rajapakse. Device fragmentation of mobile applications. http://www.comp.nus.edu.sg/~damithch/df/device-fragmentation.htm.

[33] Tommi Takala, Mika Katara, and Julian Harty. Experiences of system-level model-based gui testing of an android application. *IEEE*, March 2011.

[34] Anthony I. Wasserman. Software engineeering issues for mobile application development. *FoSER*, 8, November 2010.