# Automated extraction of augmented models for native and hybrid mobile applications in Android

Santiago Liñán Romero

*December, 2018*
Version: 0.3

Bogotá, Colombia


Systems and Computing Engineering Department
Faculty of Engineering
The Software Design Lab

Thesis submitted to the Faculty of Engineering in partial fulfillment of the requirements for the degree of MSc. in Software Engineering

# Automated extraction of augmented models for native and hybrid mobile applications in Android

Santiago Liñán Romero

*Advisor* PhD. Mario Linares Vásquez
Systems and Computing Engineering Department
Universidad de los Andes, Bogotá, Colombia

*1. Reviewer* PhD. Nicolás Cardozo
Systems and Computing Engineering Department
Universidad de los Andes, Bogotá, Colombia

*2. Reviewer* PhD. Gabriele Bavota
Faculty of Informatics
Università della Svizzera italiana (USI), Lugano

December, 2018

**Santiago Liñán Romero**

*Automated extraction of augmented models for native and hybrid mobile applications in Android*

Thesis submitted to the Faculty of Engineering in partial fulfillment of the requirements for the degree of MSc. in Software Engineering, December, 2018

Reviewers: PhD. Nicolás Cardozo and PhD. Gabriele Bavota

Advisor: PhD. Mario Linares Vásquez

**Universidad de los Andes**

*The Software Design Lab*

Faculty of Engineering

Systems and Computing Engineering Department

Cra 1 # 18A - 12

111711 and Bogotá, Colombia

# Abstract

Mobile software development involves significant challenges to developers such as device fragmentation (*i.e.,* enormous hardware and software diversity), event-driven programming (*i.e.,* programming based on user interactions, sensor readings and other events where the program must react) and continuous evolving platforms (*i.e.,* fast changing mobile frameworks and technologies). This can lead programmers to error-prone code, because of the multiple combinations of external variables that must be taken into account in an app development process. Thus, testing is an underlying necessity in mobile applications to deliver high quality apps. However, defining tests suites for app development is a difficult task that requires a lot of effort, because it must consider all the possible states of an app, its context (*e.g.,* device in which is running, sensors, touch gestures, screen proportions, connectivity), the technologies involved in the development of the app (*e.g.,* native, native written in Javascript, hybrid) and a large combination of mobile devices and operating systems.

Previous efforts have been done to extract models that support automated testing. However, as of today there is not a single model that synthesizes different aspects in mobile apps such as domain, usage, context and GUI-related information. These aspects represent complementary information that can be mixed into a single and enriched model. In this paper, we propose a multi-model representation that combines information extracted statically and dynamically from Android apps. Our approach allows practitioners to automatically extract augmented models that combine different types of information, and could help them during comprehension and testing tasks.

# Contents

# 1 Introduction

> *You can't do better design with a computer, but you can speed up your work enormously.*
>
> **— Wim Crouwel**
> (Graphic designer and typographer)

Modern mobile application testing is becoming more complex than ever before due to fragmentation and context-aware software. The majority of the mobile applications that we use daily (*e.g.,* Facebook, Waze, Uber, Web browser) rely on external web-services, connectivity methods (*e.g.,* Wi-Fi, Bluetooth, celullar networks) and sensors (*e.g.,* GPS, proximity sensor, cameras) that interact together. This amalgam of technologies increases the number of states and contexts to be considered when testing applications. Additionally to the previous examples, there is a growing number of IoT apps that have continuous interactions with their context (*e.g.,* health care applications in hospitals [23], tourist attractions [5]).

Moreover, devices where mobile applications run are becoming more powerful and capable to capture, process and transmit information of their context. To capture data, the Android Platform supports more than 12 different sensors (*e.g.,* accelerometer, gravity, light, magnetic field, pressure, temperature)[10]. Besides multiple sources of data, there are numerous possibilities concerned to connectivity. An Android phone can interact with another device via Bluetooth, NFC, Wi-Fi P2P, USB, and SIP. [7]

As a consequence of the complexity of and app and its context, testing is a hard labour for developers. Context changes can induce unforeseen faults and errors that traditional black-box or grey-box techniques do not take into account. Let us use a fictional app as an example to illustrate the case: A delivery driver is using an app that indicates her the route to deliver her packages; the battery of her smart-phone is draining fast, then, in order to extend battery life the device turns off background location and network services. The delivery driver thinks that she has finished her tasks because the application stopped showing her directions. However, the app crashed as a consequence of an unforeseen change in the device connectivity, due to battery saving configurations in the device.

In light of the time and effort required to take into account all the possible scenarios and generate the corresponding test cases, automation of testing processes become an essential job during app development, however, the most part of mobile testing is done manually by developers owing to several factors, including that current tools do not provide testers with testing capabilities for complex interactions and contextual events [13], [19], [14], [18].

We propose an approach for improving automated testing, by taking advantage of programmatic extraction of context, usage, domain and GUI models from an Android application. Along with the automated extraction, we propose the conception of a multi-model (or augmented model) that combines the aforementioned models and can be used to support testing related tasks such as test cases generation, execution, and documentation.

Manual creation of those models is time consuming; that is the reason why extracting models programmatically from static code analysis and dynamic exploration is ideal. Although models by their own are useful artifacts to document the application, we propose the generation of an augmented model mixing key information of different models. The augmented model we propose synthesizes aspects from the graphical user interface, domain and context. This multi-model has new information that only can be obtained by the intersection of the aforementioned individual models. That said, multi-models lay the foundations for future efforts to improve model-based testing in mobile platforms based on richer information contained in augmented models.

Throughout our experiments of automated extraction of models and exploration of Android apps, we found a significant amount of hybrid apps: applications that contain an embedded web browser in which all visual and logic components are contained. These are HTML, JS and CSS web applications that do not make use of the Android UI framework. Because of this singularity, we had to implement a technique to obtain the graphical hierarchy contained in applications of this kind.

We validated the extraction of multi-models exploring native open source Android apps. The resulting augmented models contain a significant number of states that were found due to induced contextual changes in the devices. Provided this, we confirmed that each of the models included in the proposal is relevant and have an impact on the behavior of the applications.

Additionally, we conducted an experiment with 30 hybrid applications available in the Google Play Store and tested our GUI ripping technique. We found that our implementation outperforms Google crawling tools such as Firebase Test Lab [8] and Android UI/Application Exerciser Monkey [11] in hybrid apps exploration.

## 1.1 Why Android?

Previously mentioned problems are common in all the mobile platforms. From the obsolete Palm WebOs, Windows Mobile or Firefox OS to the latest versions of iOS and Android. Indisputably, Google's Android and Apple's iOS are the mobile operative system that domain the market in this decade. We chose to work with Android over iOS due to several factors:

- Its open source nature
- The large amount of existing debugging and profiling tools
- Existing research in Android testing
- Access to APKs (Android packages) that contain the middleware and the developers compiled code
- The ability to launch emulators programmatically in Windows, Linux and macOs.

## 1.2 Thesis Structure

**Chapter 2**

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like "Huardest gefburn"? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

**Chapter ??**

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like "Huardest gefburn"? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written

in of the original language. There is no need for special content, but the length of words should match the language.

**Chapter ??**

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like “Huardest gefburn”? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

**Chapter ??**

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like “Huardest gefburn”? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

**Chapter 5**

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like “Huardest gefburn”? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

# 2 Related Work

> *A picture is worth a thousand words. An interface is worth a thousand pictures.*
>
> — **Ben Shneiderman**
> (Professor for Computer Science)

## 2.1 Native applications

Automated mobile app testing has been explored profusely in many areas (*e.g.,* frameworks, automation APIs, testing techniques, GUI exploration, security, usability, error reporting tools)[18], [28]. However, due to space limitations, in this section we focus specifically on two aspects: testing frameworks and GUI ripping tools (due to the nature of model extraction based on GUI events), which are closer to our proposal.

### 2.1.1 Testing frameworks

Frameworks for mobile testing define principles, concepts and architectures to structure testing processes. To this end, one of the frameworks where automated extraction of augmented models have been explicitly defined is CEL [18]. CEL is based on three main principles: *continuous*, *evolutionary* and *large-scale*. The first principle, *continuous*, refers to the fact that mobile apps should be tested under multiple environmental conditions and according to different goals. The second principle, *evolutionary*, sets forth that testing artifacts like the models, should adapt to changes in source code, environment, and in-the-wild usages. Lastly, the *large-scale* principle proposes an engine to execute test cases in real and emulated devices to tackle fragmentation.

With this in mind, the CEL framework argues that there must be a "models generator" component that combines multiple models such as GUI, usage, and contextual models to finally create a multi-model representation of an app under test. This multi-model can be used for the evolutionary generation of testing artifacts [18]. CEL affirms that current approaches for deriving representations of apps are severely

lacking a multi-model-based approach that might significantly improve the utility of model-based testing [18]. Having said that, our approach of extraction of an augmented model fits CEL's principles and architecture, enhancing GUI exploration with complementary models. Note that the CEL paper does not provide any detail regarding how a multi-model should be; thus, we are the first to instantiate the multi-model concept as proposed by CEL.

### 2.1.2 GUI ripping tools

GUI ripping tools simulate real user events on an Android device to explore an application GUI. The majority of these tools detect and report crashes generated during the exploration. Coupled with detection of crashes, others of these tools also reconstruct GUI models resulting from the exploration (*e.g.,* MonkeyLab [20], AimDroid[12], Android Ripper [3]). Previous effort has been also focused on tools that build testing suites based on their exploration strategy (*e.g.,* Sapienz [21], MobiGUITAR [2]).

The aforementioned tools have advanced significantly in terms of algorithms to explore apps' GUI. They systematically create state diagrams based on GUI information and GUI states exploration. However, GUI ripping tools lack information about the execution context that could help to determine contextual states, *e.g.,* a navigation app without GPS and Internet can not be as functional as it was intended to be, because turning GPS on and off in the same activity could result into two totally different states. GUI rippers also lack information about domain and usage of the application. However, effective mobile testing requires considering different types of information (*i.e.,* GUI, domain entities, contextual states, real usages) because combining more information could drive to exploring more states in an app.

As mentioned before, there are tools that reconstruct GUI models resulting from ripping and simulated user interactions. These approaches try to generate sequences of events under a particular strategy that drives and explores apps. [12]. Most of these tools generate finite-state machines with two purposes: (i) there is a need to generate sources of information to understand a system, whose models are nonexistent or too precarious. [16]; and (ii) model-based testing requires models to automate model generation processes.

Other approximations such as CrashScope [22] systematically explores Android apps and creates detailed crash reports in natural language. CrashScope enables context-aware input data and sensors and connectivity analysis. CrashScope makes contextual changes in the application based on API calls found statically in the code.

*Contextual fuzzing* is also an approach implemented in *Caippa* [17], a service for testing Windows mobile apps in a cloud environment. Contextual fuzzing refers to exercising apps with contexts observed in the wild (*e.g.,* eventual connectivity). In this case, GUI ripping is performed along the contextual fuzzing.

Injection of adverse conditions have been also explored not only with GUI ripping, but also using existing test cases. This is the case of *Thor* [1], a tool that injects unexpected events (*i.e.,* device rotation or incoming calls) in existing test suites.

Above all, previous research have used GUI models [20], [12]; combined usage and GUI models [21], [2]; GUI and context models (partially) [22], [17]; and context information with existing tests [1]. However, none of the aforementioned approached have covered the multi-model vision we are proposing.

## 2.2 Hybrid applications

# 3 Proposed Approach

Before describing the proposed approach we clarify the meaning of context, domain, usage, and GUI models. A domain model describes data, entities and their relationships in an application; it looks like a network of interconnected objects, where each object represents meaningful entities and concepts. Information in a domain model is useful to determine inputs and outputs in an application under test. A graphical user interface model (GUI) is also an informative model that represents all the graphic components and views of an application, and the events that trigger transitions among the views; it is commonly represented as a state diagram and it has been widely used for GUI ripping [3]. A context model [22] represents the surrounding conditions in which an app runs; in the case of mobile applications, it also includes sensors, networking, available hardware information (*e.g.,* device model, processor version), screen resolution and O.S version. An usage model describes how users can interact with an application and what functionalities are offered to them [20].

By augmented model (or multi-model) we mean the combination of the aforementioned models, in a single model that synthesizes relevant information. More formally, a multi-model is a directed graph. $G = (V, A)$, with $V$ a set of states, in which each state has a unique combination of contextual variables, GUI elements and domain entities; and $A$ a set of transitions, where each transition is a contextual change in the APP or an user interaction in the GUI that triggers a change in the app to a new state.

We illustrate the multi-model concept with the example presented in (Fig. 3.1, Page 10); the Figure depicts an abstraction for a multi-model generated dynamically and statically from a test app developed by the authors. The multi-model has 3 states (*S1*, *S2* and *S3*), and 2 edges (*T1* and *T2*). In this example, the app is impacted by contextual changes.

*S1* is the home window of the application. It is a state that mixes graphical, usage and context information. *S1* occurs when Wi-Fi and cellular networks are active. On average, when this state is active, the device has an availability of 80% of its memory. From *S1*, the app flow can go to two states (*S2* and *S3*). The transition from *S1* to *S3* (*T2*) occurs when Wi-Fi is turned off. *S3* is a state running the same

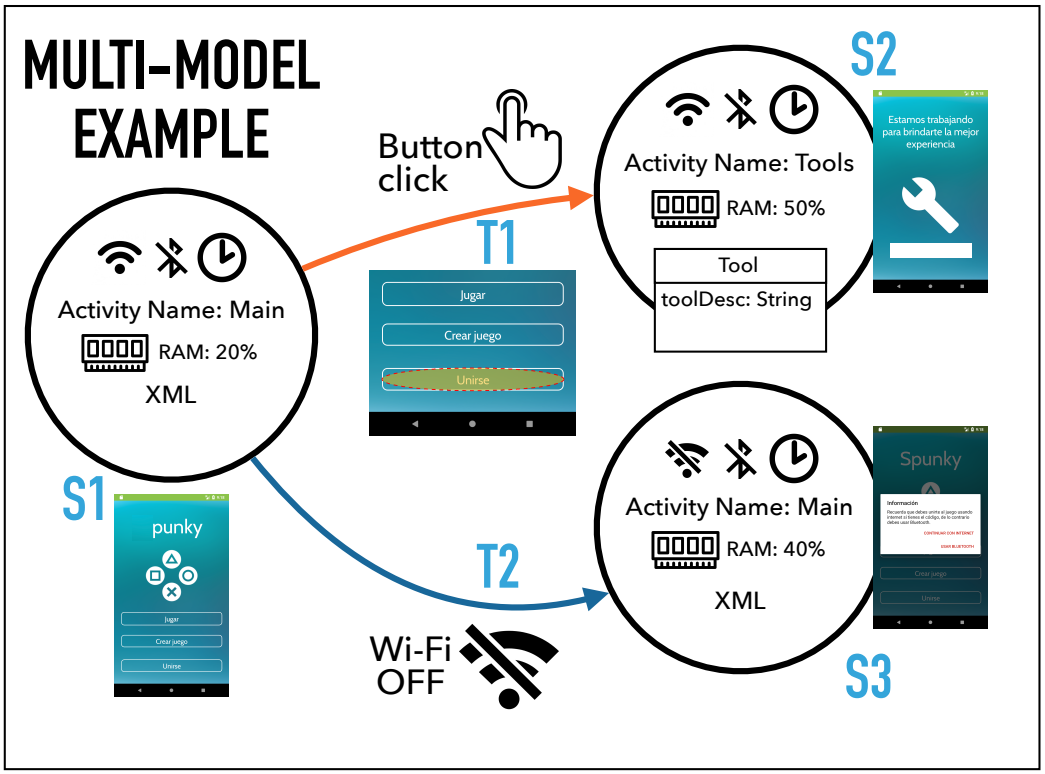


**Fig. 3.1:** Example of multi-model generated from an Android app. The states and transitions are a subset of the complete multi-model for the analyzed app.

activity as *S1*, however, it displays an alert message that fades out other buttons and the background. *S3* is more memory greedy than the initial state.

Transition *T1* changes app state from *S1* to *S2*. This transition is activated by clicking the button located at the bottom of the screen. Once this button is pressed, *S2* is activated. Context, graphical and usage information is also available in *S2*, however, it also includes domain-related information because there is a text-input for collecting information from the user; thus, we consider the activity (*i.e.,* the Android window related to *S2*) as an entity named "Tool" with a string field called "toolDesc".

## 3.1 General Approach

We have designed an architecture ((Fig. 3.2, Page 11)) that enables a series of stages which include: ripping, automatic extraction of static and dynamic models, generation of multi-models and model-based tests generation.

The multi-model generation starts by ripping the app under test. For this purpose, we have designed and developed a desktop tool called **RIP**. It is an automation software

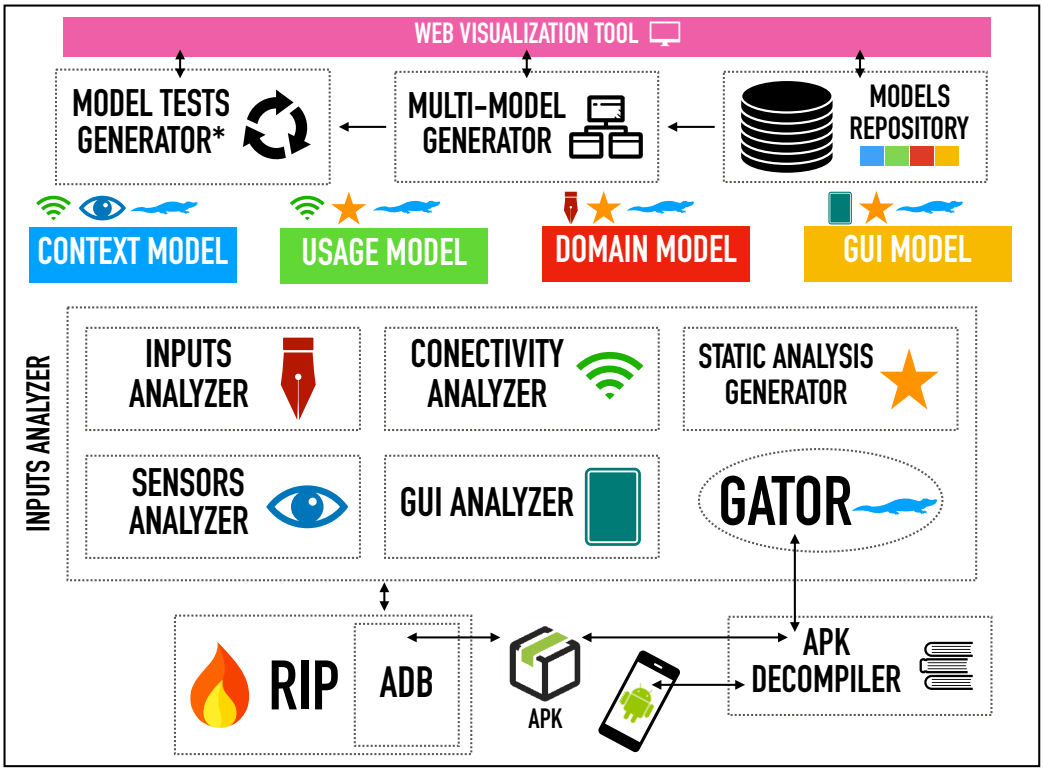


**Fig. 3.2:** Proposed architecture (**RIP**) for extracting augmented models for Android apps.

that remotely executes a series of actions and emulates user interactions into an Android device to extract models; this is done through the Android Debug Bridge (ADB) [6], a command-line tool that provides access to an Android device over USB or Wi-Fi. In comparison to monkey/random testing tools that generate random interactions to find bugs and corner cases, **RIP** follows an strategy to explore the application based on the dynamic content that appears on a device screen (like other rippers do). **RIP** not only simulates user interactions in the screen; **RIP** also executes contextual changes to the application that vary the network configuration and the readings of the sensors (*e.g.,* accelerometer, gravity, gyroscope, light, proximity, magnetic field). During the execution, **RIP** collects GUI-related, domain-related. sensors-related, and resources-related information. In addition, screenshots and GUI-hierarchies are collected for each state.

Our approach differs from existing ones because we are able to (i) generate a comprehensive list of contextual changes, (ii) extract a domain model from GUI states, and (iii) augment the dynamically-generated model with information collected statically. Due to security restrictions imposed by Android devices and the Android framework, in order to enable contextual events execution via ADB (*e.g.,* airplane mode, Wi-Fi), it is necessary to install additional software in the device. In particular, we developed a native app with a series of permissions for accessing system features (such sensors and connectivity). This component helps **RIP** to execute the intents [9] (*i.e.,*

operations to be performed by the Android device) required to activate/deactivate certain configurations that could not be altered solely with ADB.

**RIP** takes advantage of static code analysis to enrich the final generated model. When the ripping process finishes, RIP augments the collected model with a graph generated statically by GATOR [26], a static reference analyzer tool for GUI objects in Android. This tool has been chosen because its context-aware approach in static code analysis [27]. GATOR finds static references to GUI components and determines its control flow. If **RIP** identifies missing states (*i.e.,* states detected by GATOR but not by **RIP**), it adds the new states to the dynamically generated model. Thus, the GUI and usage models are represented by the combination of transitions and states information extracted from the ripping and GATOR.

## 3.2 RIP components

Our architecture defines a layer of analyzers that guide **RIP** during the models extraction:

### 3.2.1 GUI analyzer

It extracts the hierarchy of graphical components in the app. It is able to differentiate the states based on the analysis of the GUI hierarchy represented as an XML file. It means, the construction of the GUI model is done iteratively, according to the app exploration. To determine if two views are different, a decision process is followed. Firstly, if the activity names differ from each other, the two views are classified as different states. Otherwise, if the activity names are the same, then the XML is analyzed;each view is compared by the number of elements, checked boxes, buttons, labels content, alerts, and messages. In the case of menus or dialogs that are displayed on a view, we consider them as different states.

The set of states gathered from visual information are the source for the GUI model. **RIP** captures image screen-shots and XML layouts from every state it has found. It also records the interaction/event that triggered the state transition (*e.g.,* pressing a button).

### 3.2.2 Inputs analyzer

It enables **RIP** to identify user input-related GUI Android components and interact with them accordingly (*e.g.,* check boxes, text boxes, lists, scrollable views). **RIP**

creates random input data for the components based on the input types defined for text fields, and the component nature (*e.g.,* a check box can be activated or deactivated). Keyboards that appear on the screen and GUI components meta-data, give us clues about concepts and domain entities that are part of the domain model of the application. We define entities of the domain model as views that have input elements. Each entity has a set of attributes that correspond to input-related components in the view, and the attributes type is inferred from the type of input allowed (*e.g.,* numbers, special characters, boolean check, lists, *etc.*).

### 3.2.3 Sensors analyzer

During the ripping, **RIP** turns off/on sensors, randomly. The sensors analyzer identifies which sensors the application has access to from the app's manifest file. Once the sensors analyzer has this information, it detects the states where sensors are on and off using specific ADB commands.

### 3.2.4 Connectivity analyzer

This component identifies connectivity status of the app during all the ripping execution. Once the GUI has detected an initial set of states, it sends requests to the device in order to control Bluetooth, Wi-Fi, celullar networks and airplane mode. **RIP** triggers connectivity changes in every view of the application. If an error occurs or the GUI changes after the contextual change, then a new state is discovered and added to the model.

### 3.2.5 Static analyzer and GATOR

Different from the other components, it builds a flow graph of the app based strictly on static analysis, by relying on the GATOR tool [26]. GATOR creates a window transition graph with activities, dialogs and menus. The transitions of the graphs include events such as button pressings and window stack operations (push and pop). This flow graph is used to augment the model collected dynamically by **RIP**, in particular to have a more comprehensive list of transitions and states.

### 3.2.6 RIP GUI

It interacts directly with ADB to explore the applications dynamically and coordinates the execution of the other components. The **RIP** GUI combines the ripping, interactive data collection, and static analysis, to generate individual models and a

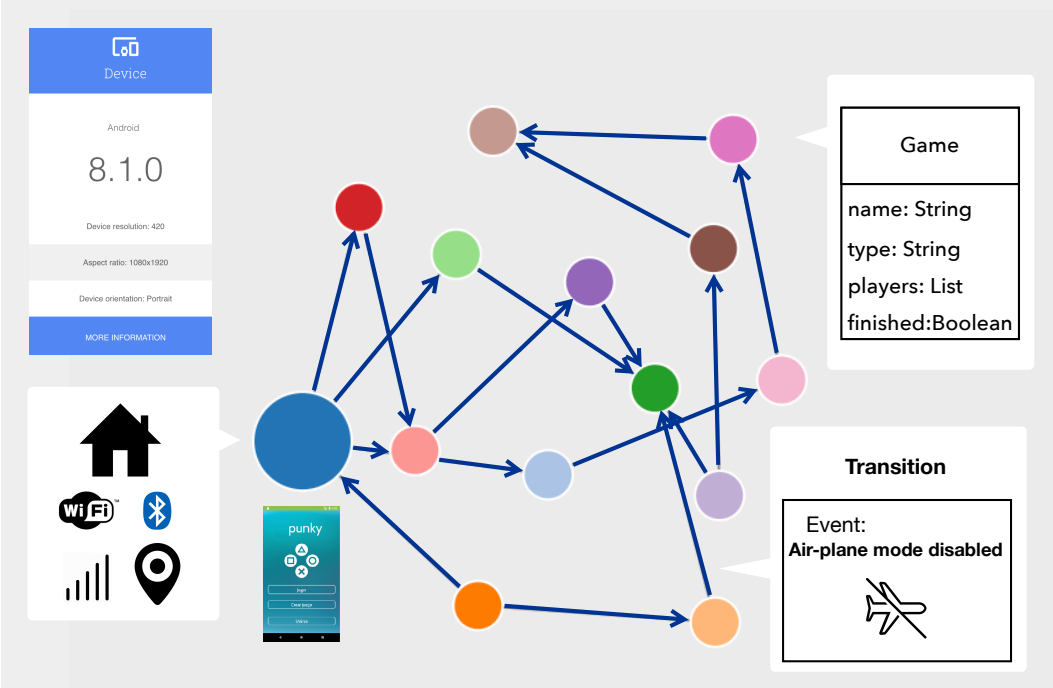


**Fig. 3.3:** Example of multi-model created by RIP.

multi-model like the one presented in (Fig. 3.3, Page 14); the **RIP** GUI also generates the model as a JSON file that can be analyzed by any other tool.

## 3.3 Ripping hybrid applications

Hybrid apps are those in which developers write significant portions of their application in cross-platform web technologies, while maintaining direct access to native APIs when required [**ibmHybrid**]. This way of developing applications is gaining popularity among the mobile developers community because:

- Reduces the time to develop cross platform apps
- It is based on Web technologies, which allows developers to recycle modules and components from existing Web developments
- Browser engines are becoming faster and mobile devices more powerful
- Performance differences between native and web technologies are becoming imperceptible

Currently, Apache Cordova is the component that enables the access to native iOS and Android APIs from the JS code. This piece of software gives access to Local Storage,


**Fig. 3.4:** Example of multi-model generated from an Android app. The states and transitions are a subset of the complete multi-model for the analyzed app.

Camera, GPS, and all the available sensors of each platform. Apache Cordova is the core of many popular hybrid frameworks such as Ionic, Adobe Phonegap, Monaca or Visual Studio.

### 3.3.1 Obstacles ripping hybrid apps

# 4 Empirical study

The *goal* of our empirical study is to evaluate the proposal of combining multiple models to improve accuracy of testing processes and to evaluate **RIP** as a tool to automatically crawl Android applications. We evaluate our proposal in terms of (i) the importance of combining orthogonal information from different models and (ii) the suitability of a multi-model testing tool, in the process of developing mobile applications. At the same time, we evaluate **RIP** in terms of (iii) RIP's ability to crawl native and hybrid Android apps, (iv) RIP's ability to detect hybrid apps and (v) RIP's ability to detect bugs and generate crashes. The *context* of this study consists of (i) a set of 20 open source native APKs , (ii) a set of 30 hybrid APKs from the Google Play Store, (iii) three additional approaches to extract models and crawl Android applications: DroidBot [**droidBot**], Firebase Test Lab Robo [**testLabRobo**] and Monkey [11].

The *quality focus* of this study is the effectiveness of **RIP** to detect the maximum number of states on both native and hybrid apps, reporting crashes and generating multi-models from these applications. To aid in achieving the goals of the study, the following research questions were formulated:

- ***RQ***$_1$: *Is the combination of multiple models useful to gather more information of an app under test?*
- ***RQ***$_2$: *Would actual mobile developers consider using **RIP** as part of their workflow?*
- ***RQ***$_3$: *How accurate is the 'state discovery algorithm' implemented in RIP?*
- ***RQ***$_4$: *Are multi-model discovery strategies suitable to detect crashes and bugs in mobile apps?*

It should be mentioned that in answering $RQ_3$ and $RQ_4$ we take into account hybrid and native apps. These two development strategies could be indistinguishable for final users of the apps, yet their are very different to crawl, analyze and build.

Tab. 4.1: General results obtained from multi-model extraction. **Abbreviations for column headings**. CR = Car Report, YLC = Your Local Weather, SC = Simple Calendar

| | CR [15] | Punky | YLC [24] | Tasks [4] | SC [25] |
|---|---|---|---|---|---|
| **Execution time (mins.)** | 4.6 | 4 | 6 | 5.3 | 4.8 |
| **Total number of states discovered** | 51 | 30 | 31 | 37 | 36 |
| **States discovered due to contextual changes** | 0 | 8 | 5 | 4 | 4 |
| **Domain entities extracted** | 21 | 4 | 23 | 17 | 20 |
| **Attributes extracted** | 63 | 96 | 27 | 60 | 44 |

## 4.1 $RQ_1$ Combining multiple models to improve accuracy of testing processes

to answer $RQ_1$, we conducted a case study to show that combining the different models in an augmented model could improve the accuracy of testing processes. It means, we wanted to understand whether combining the models is useful to gather more knowledge of an app under test, and whether that knowledge could be used to generate more robust test cases. To this, we (i) extracted multi-models from 5 different Android apps which are listed in (siehe Tab. 4.1), and (ii) collected the following information using RIP:

- Execution time required to explore the applications until no more states were discovered,
- Total number of states discovered by triggering simulated user interactions and context events (ripping),
- States that were discovered due to contextual changes (ripping + context),
- Domain entities extracted from the apps,
- Attributes extracted from each domain entity

In the study we executed **RIP** in two modes: (i) ripping only mode, and (ii) ripping + contextual model execution. As reported in (siehe Tab. 4.1), when we added the contextual model, we found new states in 4 out of 5 applications tested. This suggest that generating augmented models can improve software comprehension and testing tasks for mobile apps, because different states are activated by combining individual models.

The first row of (siehe Tab. 4.1)shows the execution time required to explore the applications until no more states were discovered. Second row includes total number of states discovered by triggering simulated user interactions and context events.

Next row shows only the states that were discovered due to contextual changes. Based on the information of these two rows is easy to conclude that Car Report is an application that relays much less to contextual changes in comparison with Your local weather or Simple Calendar. In the case of Punky, changes due to context were triggered by Wi-Fi, Bluetooth and accelerometer interactions.

The 'Domain entities extracted' row refers to entities discovered. This number is directly correlated to the number of application views that contain text inputs, selectors and checkboxes. Each entity contains a set of attributes, whose sum correspond to the final row.

Table 4.1 shows that information from each model is complementary and orthogonal. For instance, applications that are highly dependent on sensors and networking connections have states that could not be discovered if contextual events are not considered.

## 4.2 **RQ**$_3$ Automated exploration of hybrid and native apps

# 5 Conclusion

The results from our preliminary study suggest that automatically extracting augmented models from Android apps enables better understanding of the apps. For modern mobile applications, ripping apps — but based only on GUI exploration— is not enough because they are context-aware. To that end, context, GUI, usage and domain models should be extracted and combined together to build more useful and comprehensive augmented models.

In this paper, we propose to take advantage of the augmented models and implement multi-model-based testing. Augmented models contain much more information than traditional state diagrams of GUIs. All things considered, multi-models have richer information that will enable generation of more effective test suites.

The proposed multi-model could be used to generate test cases, first, using the context variables to define the environmental conditions of the test cases; secondly, generating inputs in test cases by relying on domain entities and attributes; finally, using the GUI and usage information to provide test cases based on developer requirements, such as coverage or specific functionalities.

Further studies will be conducted to evaluate and improve automated exploration of the apps, comparing states discovered by our approach against states discovered by users' interactions. In addition to our efforts for improving the multi-model extraction technique, model-based testing from augmented models will be our next step.

# Bibliography


[1] Christoffer Quist Adamsen, Gianluca Mezzetti, and Anders Møller. „Systematic execution of Android test suites in adverse conditions“. In: *ISSTA'15* (2015) (cit. on p. 7).

[2] Domenico Amalfitano, Anna Rita Fasolino, Porfirio Tramontana, Bryan Dzung Ta, and Atif M Memon. „MobiGUITAR: Automated Model-Based Testing of Mobile Apps“. In: *IEEE Software* 32.5 (2015), pp. 53–59 (cit. on pp. 6, 7).

[3] Domenico Amalfitano, Anna Rita Fasolino, Porfirio Tramontana, Salvatore De Carmine, and Atif M. Memon. „Using GUI ripping for automated testing of Android applications“. In: *ASE'12* (2012) (cit. on pp. 6, 9).

[4] Alex Baker. *Tasks* (cit. on p. 18).

[5] M. Casillo, F. Colace, F. Pascale, S. Lemma, and M. Lombardi. „Context-aware computing for improving the touristic experience: A pervasive app for the Amalfi coast“. In: *M&N'17*. 2017, pp. 1–6 (cit. on p. 1).

[6] Google. *Android Debug Bridge (adb) | Android Developers*. URL: `https://developer.android.com/studio/command-line/adb` (cit. on p. 11).

[7] Google. *Connectivity - Android Developers* (cit. on p. 1).

[8] Google. *Firebase Test Lab*. URL: `https://firebase.google.com/docs/test-lab/android/overview` (cit. on p. 2).

[9] Google. *Intent - Android Developers*. URL: `https://developer.android.com/reference/android/content/Intent` (cit. on p. 11).

[10] Google. *Sensors - Android Developers* (cit. on p. 1).

[11] Google. *UI/Application Exerciser Monkey*. URL: `https://developer.android.com/studio/test/monkey` (cit. on pp. 2, 17).

[12] T. Gu, C. Cao, T. Liu, et al. „AimDroid: Activity-Insulated Multi-level Automated Testing for Android Applications“. In: *ICSME'17*. 2017, pp. 103–114 (cit. on pp. 6, 7).

[13] M. E. Joorabchi, A. Mesbah, and P. Kruchten. „Real Challenges in Mobile App Development“. In: *ESEM'13*. 2013, pp. 15–24 (cit. on p. 2).

[14] Pavneet Singh Kochhar, Ferdian Thung, Nachiappan Nagappan, Thomas Zimmermann, and David Lo. „Understanding the Test Automation Culture of App Developers“. In: *ICST'15* (2015) (cit. on p. 2).

[15] Jan Kühle. *Car Report* (cit. on p. 18).

[16] M. Langhammer, A. Shahbazian, N. Medvidovic, and R. H. Reussner. „Automated Extraction of Rich Software Models from Limited System Information“. In: *WICSA'16*. 2016, pp. 99–108 (cit. on p. 6).

[17] Chieh-Jan Mike Liang, Ranveer Chandra, Feng Zhao, et al. „Caiipa: Automated large-scale mobile app testing through contextual fuzzing“. In: *MobiCom'14* (2014) (cit. on p. 7).

[18] Mario Linares-Vasquez, Kevin Moran, and Denys Poshyvanyk. „Continuous, Evolutionary and Large-Scale: A New Perspective for Automated Mobile App Testing“. In: *ICSME'17* (2017) (cit. on pp. 2, 5, 6).

[19] Mario Linares-Vasquez, Carlos Bernal-Cardenas, Kevin Moran, and Denys Poshyvanyk. „How do Developers Test Android Applications?“ In: *ICSME'17* (2017) (cit. on p. 2).

[20] Mario Linares-Vasquez, Martin White, Carlos Bernal-Cardenas, Kevin Moran, and Denys Poshyvanyk. „Mining Android App Usages for Generating Actionable GUI-Based Execution Scenarios“. In: *MSR'15* (2015) (cit. on pp. 6, 7, 9).

[21] Ke Mao, Mark Harman, and Yue Jia. „Sapienz: multi-objective automated testing for Android applications“. In: *ISSTA'16* (2016) (cit. on pp. 6, 7).

[22] Kevin Moran, Mario Linares-Vasquez, Carlos Bernal-Cardenas, Christopher Vendome, and Denys Poshyvanyk. „Automatically Discovering, Reporting and Reproducing Android Application Crashes“. In: *ICST'16* (2016) (cit. on pp. 6, 7, 9).

[23] R. Sharma, P. Soni, K. Shah, and B. Panchal. „Health care application for Android smartphones using Internet of Things (IoT)“. In: *INDIACom'16*. 2016, pp. 1430–1433 (cit. on p. 1).

[24] thuryn1@gmail.com. *Your local weather* (cit. on p. 18).

[25] Simple Mobile Tools. *Simple Calendar* (cit. on p. 18).

[26] S. Yang, H. Zhang, H. Wu, et al. „Static Window Transition Graphs for Android (T)“. In: *ASE'15*. 2015, pp. 658–668 (cit. on pp. 12, 13).

[27] Shengqian Yang, Dacong Yan, Haowei Wu, Yan Wang, and Atanas Rountev. „Static Control-Flow Analysis of User-Driven Callbacks in Android Applications“. In: *ICSE'15* (2015) (cit. on p. 12).

[28] Samer Zein, Norsaremah Salleh, and John Grundy. „A systematic mapping study of mobile application testing techniques“. In: *Journal of Systems and Software* 117 (2016), pp. 334–356 (cit. on p. 5).

# List of Figures

# List of Tables

# Colophon

This thesis was typeset with LaTeX 2$_\varepsilon$. It uses the *Clean Thesis* style developed by Ricardo Langner. The design of the *Clean Thesis* style is inspired by user guide documents from Apple Inc.

Download the *Clean Thesis* style at `http://cleanthesis.der-ric.de/`.