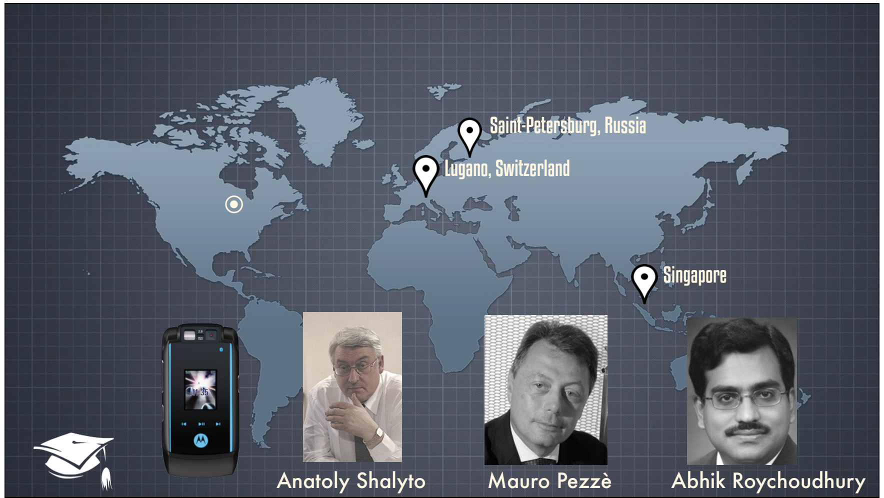
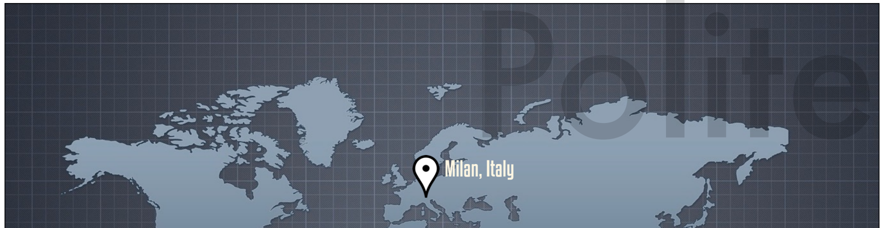
POLITECNICO
MILANO 1863
DIPARTIMENTO DI ELETTRONICA
INFORMAZIONE E BIOINGEGNERIA

Konstantin Rubinov

Politecnico di Milano, Italy



1886
MOTOROLA
ITMO
Software Testing and Analysis Research group
NUS National University of Singapore
School of Computing
Leading The World With Asia's Best



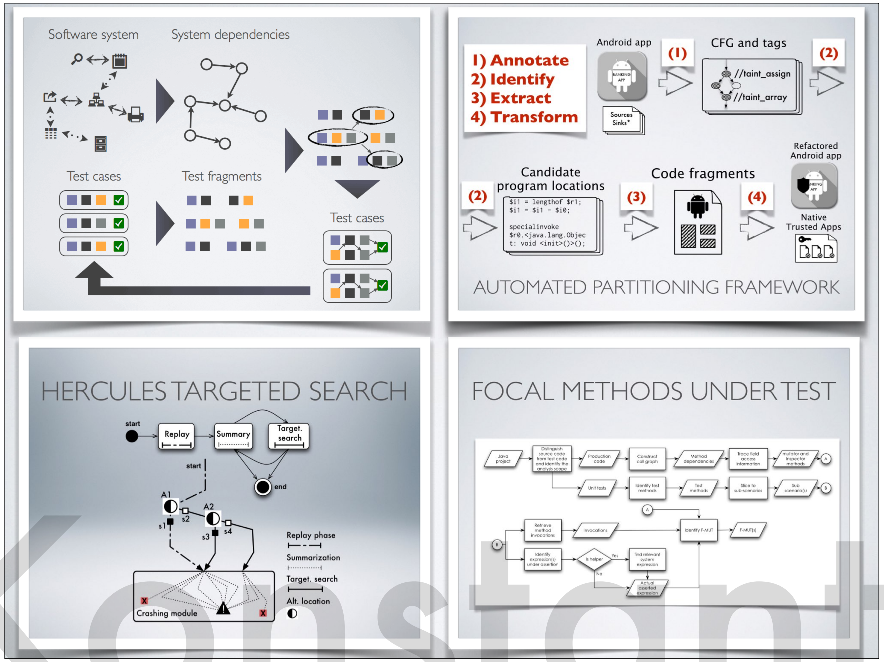
Current focus:
research and teaching in program analysis, automated software testing, security, and quality assurance for mobile applications (Android)

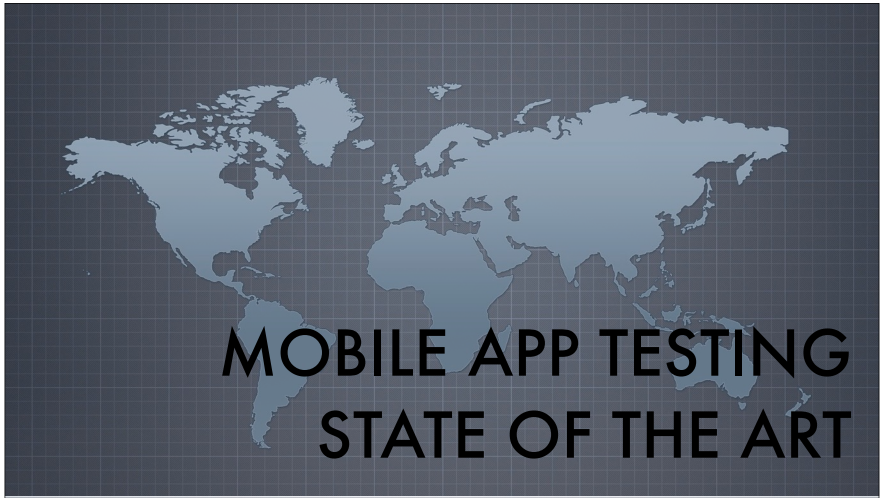
POLITECNICO
MILANO 1863
DIPARTIMENTO DI ELETTRONICA
INFORMAZIONE E BIOINGEGNERIA

MIP
POLITECNICO DI MILANO
GRADUATE SCHOOL
OF BUSINESS

RESEARCH BACKGROUND

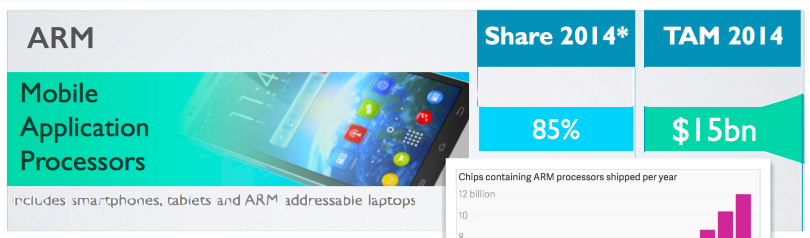
- Automated test case generation, Design for testability (with application in software recommendations systems)
- Analysis of software binaries, symbolic execution (application in vulnerability detection and crash reproduction)
- Analysis and testing of Android applications (application in automated partitioning of Android apps for enhanced security)

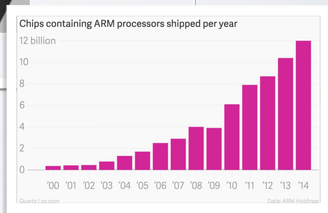




Period	Android	iOS	Windows Phone	BlackBerry OS	Others
2015Q2	82.8%	13.9%	2.6%	0.3%	0.4%
2014Q2	84.8%	11.6%	2.5%	0.5%	0.7%
2013Q2	79.8%	12.9%	3.4%	2.8%	1.2%
2012Q2	69.3%	16.6%	3.1%	4.9%	6.1%

Source: IDC, Aug 2015





Android and ARM TrustZone

ANDROID

Mobile apps are designed for a portable device with **limited resources**;

are instances of **reactive non-terminating software** with **dynamic/adaptive user interfaces,** and

context-aware and react to changes in environment and physical factors;

Apps are often **hybrid** -- presenting both mobile and web content

Mobile app vs. Web app

== 'pages' are alike with 'activities'

== DOM is alike with XML view hierarchy

Mobile app vs. Embedded software

== limited resources; sensors

!= mobile apps can interact with other apps

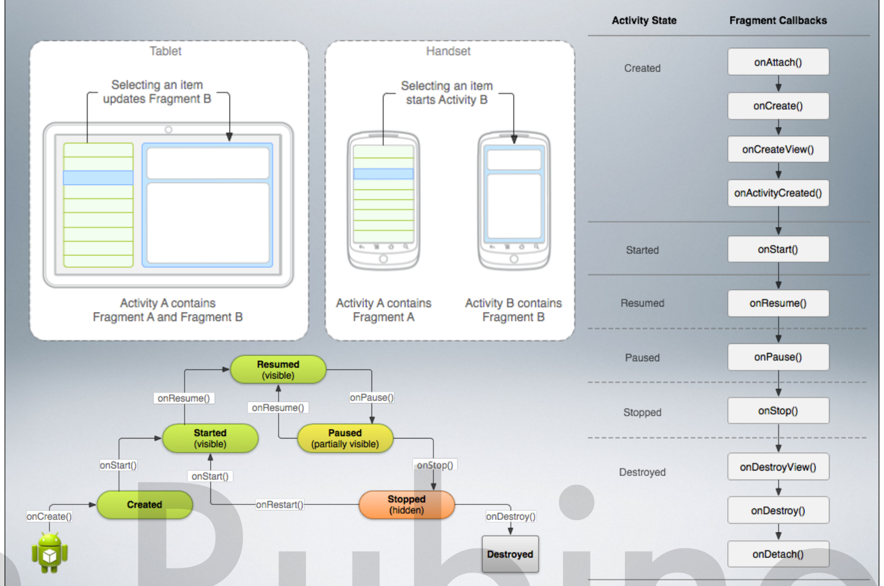
Mobile app vs. Desktop application

!= app manages it's lifecycle

!= complex input gestures (swipes, multi-touch)

!= contextual input (location, acceleration, gyroscope, etc.)

IS IT DIFFERENT?

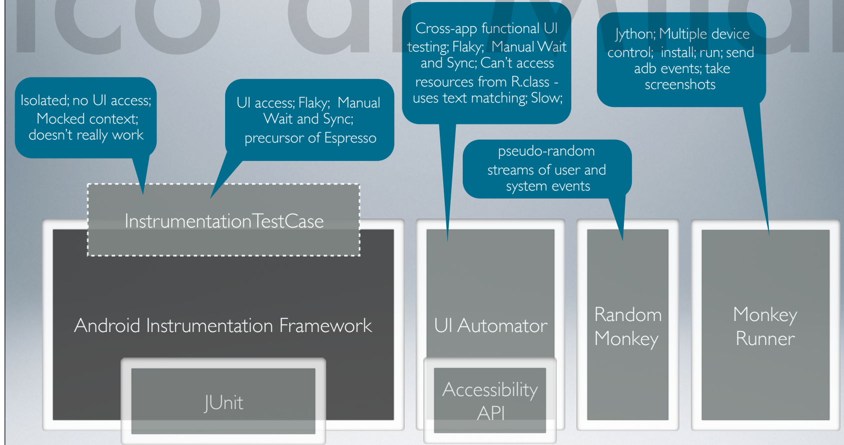


[P. Kochhar, F. Thung, N. Nagappan, T. Zimmermann, and D. Lo. ICST 2015]

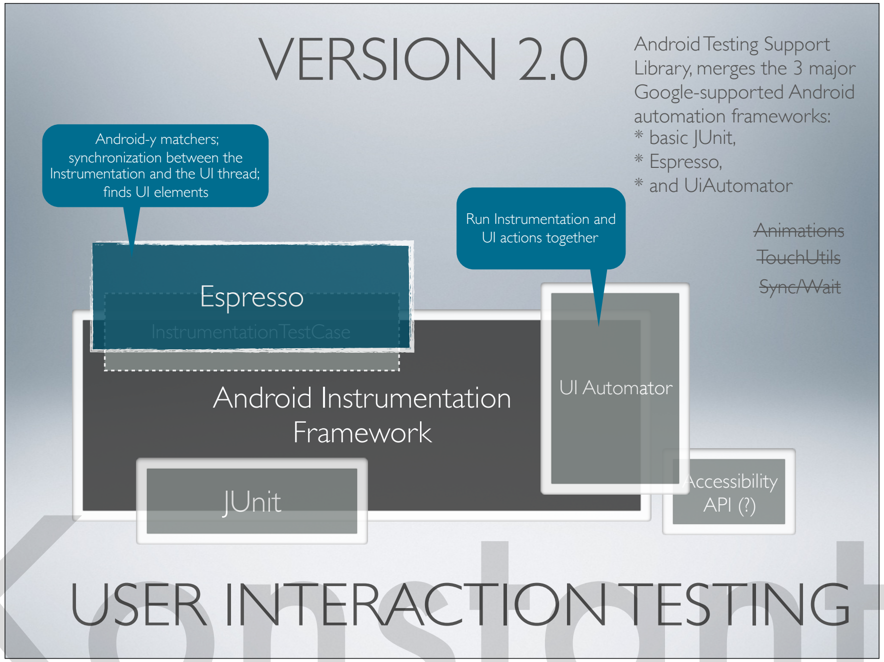
"Only about 14% of the [600 open source] apps contain test cases and only about 9% of the apps that have executable test cases have coverage above 40%."

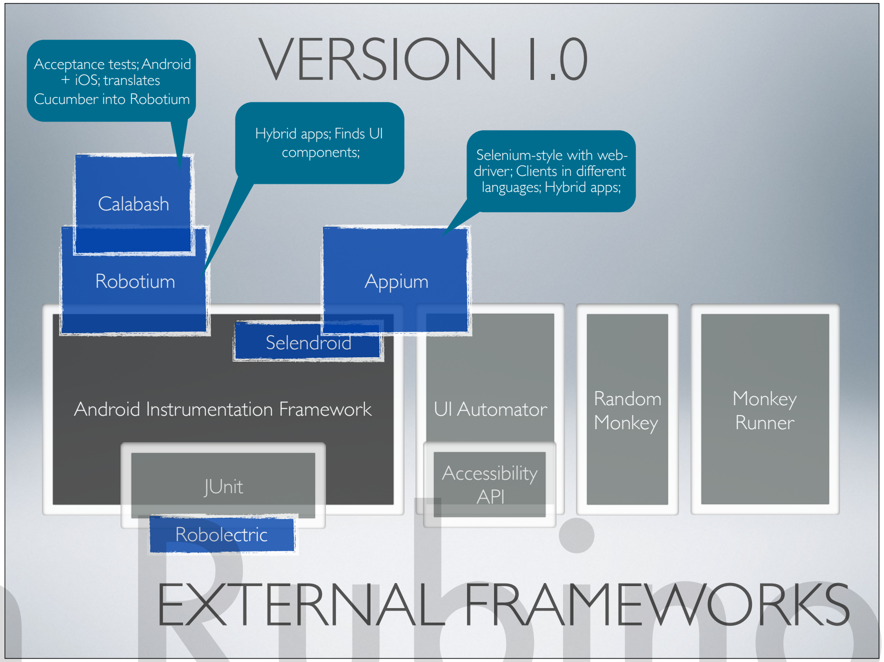
OBSERVATIONS

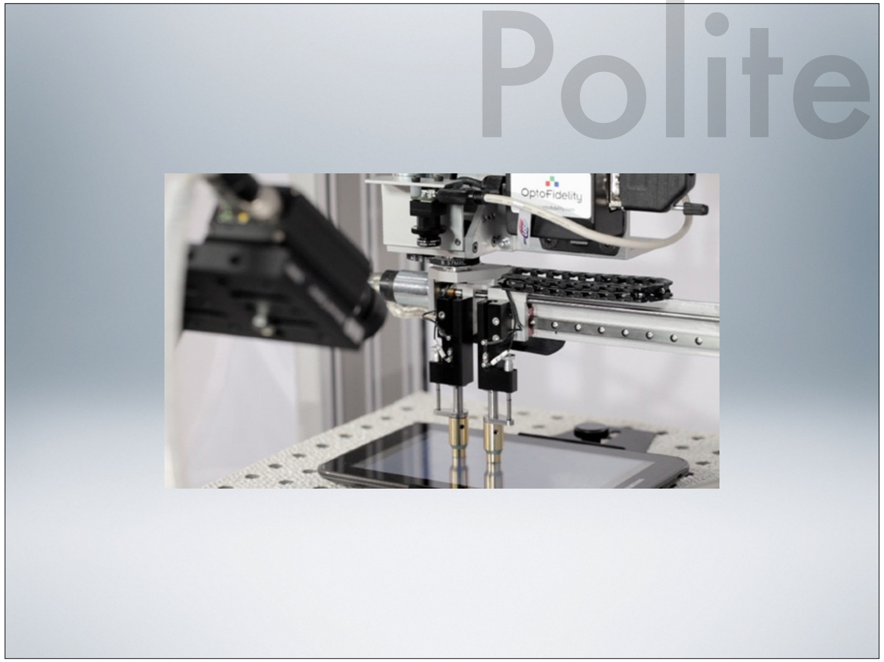
VERSION 1.0

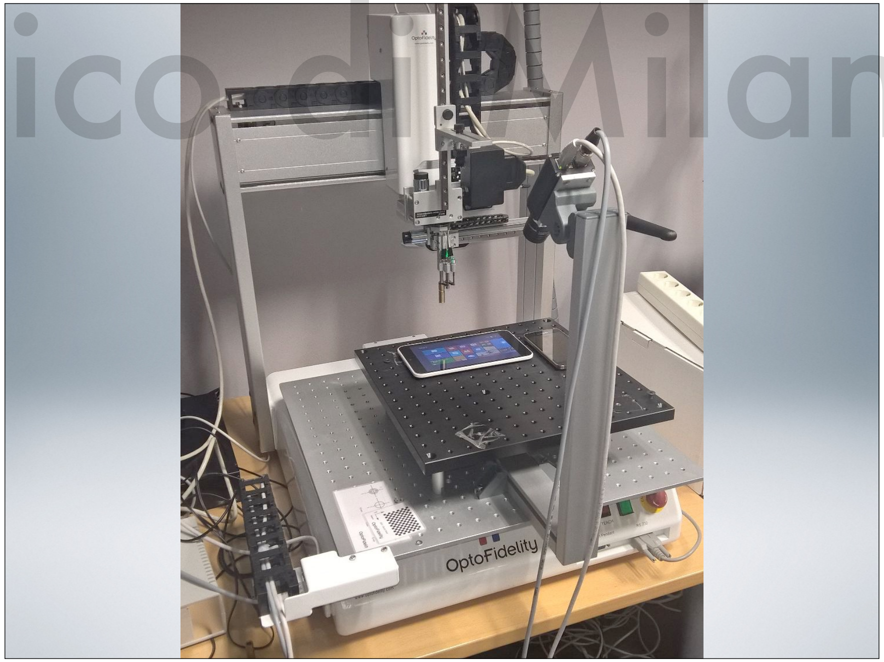


TESTING INFRASTRUCTURE


VERSION 2.0
Android Testing Support Library, merges the 3 major Google-supported Android automation frameworks:
* basic JUnit,
* Espresso,
* and UiAutomator
~~Animations~~
~~TouchUtils~~
~~Sync/Wait~~
Android-y matchers; synchronization between the Instrumentation and the UI thread; finds UI elements
Run Instrumentation and UI actions together
Espresso
InstrumentationTestCase
Android Instrumentation Framework
UI Automator
JUnit
Accessibility API (?)
USER INTERACTION TESTING


VERSION 1.0
Acceptance tests; Android + iOS; translates Cucumber into Robotium
Hybrid apps; Finds UI components;
Selenium-style with web-driver; Clients in different languages; Hybrid apps;
Calabash
Robotium
Appium
Selendroid
Android Instrumentation Framework
UI Automator
Random Monkey
Monkey Runner
JUnit
Accessibility API
Robolectric
EXTERNAL FRAMEWORKS


OptoFidelity


OptoFidelity

STATE OF THE AFFAIRS

- Overlapping functionality between different frameworks
- Poor and conflicting testing documentation (esp. Google)
- Game app testing is weak/missing. Game apps bypass Android Views to draw and thus cannot be tested as normal view resources (Testdroid solution has approached it through image recognition)
- Activity testing is slow, requires mocking, and has to run on Emulator/Device (addressed by http://robolectric.org - mimic how Android creates Activities and drives them through their lifecycle)
- Active improvements in 2014-2015
- Automated test execution, but little or no automated test case generation

ANDROID INFRASTRUCTURE

[Choudhary, Gorla and Orso ASE 2015]

[P. Kochhar, F. Thung, N. Nagappan, T. Zimmermann, and D. Lo. ICST 2015]

- Android emulators are slow and unstable
- "Flaky test" issue
- Input generation (Input data like user account is impossible to generate automatically)
- Supporting a wide range of devices, platforms and versions
- GUI models are limited ("Some events may change the internal state of the app without affecting the GUI"), yet allow to cover large parts of app
- Isolating app behavior yet testing platform specific functionality
- State-sensitivity and state explosion

CHALLENGES

RESEARCH SAMPLES

Dynodroid: An Input Generation System for Android Apps

Aravind Machiry, Rohan Tahiliani, Mayur Naik
Georgia Institute of Technology, USA
{amachiry, rohan_tahil, naik}@gatech.edu

ABSTRACT

We present a system Dynodroid for generating relevant inputs to unmodified Android apps. Dynodroid views an app as an event-driven program that interacts with its environment by means of a sequence of events through the Android framework. By instrumenting the framework once and for all, Dynodroid monitors the reaction of an app upon each event in a lightweight manner, using it to guide the generation of the next event to the app. Dynodroid also allows interleaving events from machines, which are better at generating a large number of simple inputs, with events from humans, who are better at providing intelligent inputs.

We evaluated Dynodroid on 50 open-source Android apps, and compared it with two prevalent approaches: users manually exercising apps, and Monkey, a popular fuzzing tool. Dynodroid, humans, and Monkey covered 55%, 60%, and 53%, respectively, of each app's Java source code on average. Monkey took 20X more events on average than Dynodroid. Dynodroid also found 9 bugs in 7 of the 50 apps, and 6 bugs in 5 of the top 1,000 free apps on Google Play.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging

General Terms

Reliability, Experimentation

Keywords

GUI testing, testing event-driven programs, Android

1. INTRODUCTION

Mobile apps—programs that run on advanced mobile devices such as smartphones and tablets—are becoming increasingly prevalent. Unlike traditional enterprise software, mobile apps serve a wide range of users in heterogeneous and demanding conditions. As a result, mobile app developers, testers, marketplace auditors, and ultimately end users can benefit greatly from what-if analyses of mobile apps.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ESEC/FSE '13, August 18-26, 2013, Saint Petersburg, Russia
Copyright 2013 ACM 978-1-4503-2237-9/13/08 ...$15.00.

What-if analyses of programs are broadly classified into static and dynamic. Static analyses are hindered by features commonly used by mobile apps such as code obfuscation, native libraries, and a complex SDK framework. As a result, there is growing interest in dynamic analyses of mobile apps (e.g., [1, 12, 13, 25]). A key challenge to applying dynamic analysis ahead-of-time, however, is obtaining program inputs that adequately exercise the program's functionality.

We set out to build a system for generating inputs to mobile apps on Android, the dominant mobile app platform, and identified five key criteria that we felt such a system must satisfy in order to be useful:

- *Robust*: Does the system handle real-world apps?
- *Black-box*: Does the system forgo the need for app sources and the ability to decompile app binaries?
- *Versatile*: Is the system capable of exercising important app functionality?
- *Automated*: Does the system reduce manual effort?
- *Efficient*: Does the system generate concise inputs, i.e., avoid generating redundant inputs?

This paper presents a system Dynodroid that satisfies the above criteria. Dynodroid views a mobile app as an event-driven program that interacts with its environment by means of a sequence of events. The main principle underlying Dynodroid is an *observe-select-execute* cycle, in which it first *observes* which events are relevant to the app in the current state, then *selects* one of those events, and finally *executes* the selected event to yield a new state in which it repeats this process. This cycle is relatively straightforward for *UI events*—inputs delivered via the program's user interface (UI) such as a tap or a gesture on the device's touchscreen. In the *observer* stage, Dynodroid determines the layout of widgets on the current screen and what kind of input each widget expects. In the *selector* stage, Dynodroid uses a novel randomized algorithm to select a widget in a manner that penalizes frequently selected widgets without starving any widget indefinitely. Finally, in the *executor* stage, Dynodroid exercises the selected widget.

In practice, human intelligence may be needed for exercising certain app functionality, in terms of generating both individual events (e.g., inputs to text boxes that expect valid passwords) and sequences of events (e.g., a strategy for winning a game). For this reason, Dynodroid allows a user to observe an app reacting to events as it generates them, and lets the user pause the system's event generation, manually generate arbitrary events, and resume the system's event

[A. Machiry, R. Tahiliani, and M. Naik. FSE 2013]

Vision: "synergistically combine human and machine for testing"

DYNODROID

Overview

- Finds *relevant* system events: what app can react to at each moment in execution
- Considers both UI (leaf/visible nodes on View Hierarchy) and System events (broadcast and system services)
- Randomized exploration (select a widget by penalize frequently selected ones)
- The approach is black box, it works iteratively and finds registered listeners dynamically
- Allows intermediate manual input

DYNODROID

Evaluation

- Dynodroid exclusively covers 0-26% of code; an average of 4%
- Dynodroid + Manual input covers 4-91% of code per app; an average of 51%

DYNODROID

Systematic Execution of Android Test Suites in Adverse Conditions

Christoffer Quist Adamsen, Aarhus University, Denmark, quist@cs.au.dk
Gianluca Mezzetti, Aarhus University, Denmark, mezzetti@cs.au.dk
Anders Møller, Aarhus University, Denmark, amoeller@cs.au.dk

ABSTRACT

Event-driven applications, such as, mobile apps, are difficult to test thoroughly. The application programmers often put significant effort into writing end-to-end test suites. Even though such tests often have high coverage of the source code, we find that they often focus on the expected behavior, not on occurrences of unusual events. On the other hand, automated testing tools may be capable of exploring the state space more systematically, but this is mostly without knowledge of the intended behavior of the individual applications. As a consequence, many programming errors remain unnoticed until they are encountered by the users.

We propose a new methodology for testing by leveraging existing test suites such that each test case is systematically exposed to adverse conditions where certain unexpected events may interfere with the execution. In this way, we explore the interesting execution paths and take advantage of the assertions in the manually written test suite, while ensuring that the injected events do not affect the expected outcome. The main challenge that we address is how to accomplish this systematically and efficiently.

We have evaluated the approach by implementing a tool, THOR, working on Android. The results on four real-world apps with existing test suites demonstrate that apps are often fragile with respect to certain unexpected events and that our methodology effectively increases the testing quality: Of 507 individual tests, 429 fail when exposed to adverse conditions, which reveals 66 distinct problems that are not detected by ordinary execution of the tests.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging

Keywords

UI testing; automated testing; mobile apps; Android

This is the author's version of the work. It is posted here for your personal use. Not for redistribution. The definitive version was published in the following publication:

ISSTA'15, July 12–17, 2015, Baltimore, MD, USA
© 2015 ACM. 978-1-4503-3620-8/15/07...
http://dx.doi.org/10.1145/2771783.2771786

1. INTRODUCTION

As of May 2015 more than 1.5 million Android apps have been published in the Google Play Store.[1] Execution of such apps is driven by events, such as, user events caused by physical interaction with the device. One of the primary techniques developers apply for detecting programming errors is to create end-to-end test suites (also called UI tests) that explore the UI programmatically, mimicking user behavior while checking for problems. Testing frameworks, such as, Robotium,[2] Calabash,[3] and Espresso[4] are highly popular among Android app developers. As a significant amount of the software development time is often devoted to testing [19], it is not unusual that test suites have high coverage of the source code and incorporate a deep knowledge of the app UI and logic. Furthermore, the result of each single test can be of critical importance to sanction the success of the entire development process, as tests may be used for verifying scenarios in the business requirements.

Nevertheless, due to the event-driven model, only a tiny fraction of the possible inputs is typically explored by such test suites. As the test cases are written manually, they tend to concentrate on the expected event sequences, not on the unusual ones that may occur in real use environments. In other words, although the purpose of writing test suites is to detect errors, the tests are traditionally run in "good weather" conditions where no surprises occur.

Our goal is to improve testing of apps also under adverse conditions. Such conditions may arise from events that can occur at any time, comprising notifications from the operating system due to sensor status changes (e.g. GPS location change), operating system interference (e.g. low memory), or interference by another app that concurrently accesses the same resource (e.g. audio). It is well known that Android apps can be difficult to program when such events may occur at any time and change the app state [13, 14, 22, 29]. A typical example of bad behavior is that the value of a form field is lost when the screen is rotated.

As a supplement or alternative to manually written test suites, many automated testing techniques have been created aiming to find bugs with little or no help from the developer [3, 4, 6, 7, 12, 13, 14, 22, 23, 25, 27, 29]. The primary advantage of such techniques is that they can, in principle, explore the state space more extensively, including the unusual event sequences. However, these techniques generally cannot

[1] http://www.appbrain.com/stats/number-of-android-apps
[2] http://code.google.com/p/robotium
[3] http://calaba.sh/
[4] http://code.google.com/p/android-test-kit

83

[C. Q. Adamsen, G. Mezzetti, and A. Møller. ISSTA 2015]

Extend existing test cases with neutral system events

http://brics.dk/thor

THOR

Approach

- Test adverse conditions, yet injecting expected events
- Injecting neutral system events (An event sequence *n* is neutral if injecting *n* during a test *t* is not expected to affect the outcome of *t*)
- Examples: Pause → Resume; Pause → Stop → Restart; Audio focus loss → Audio focus gain;
- Orig. test cases are redundant. Optimization: omit injecting *n* in abstract state *s* after event *e*, if *(n, s, e)* already appears in the cache (uses View Hierarchy)

THOR

Overview

- Systematic system event fuzzing based on existing test cases with the focus on activity lifecycle changes
- Finds suitable locations for injecting events in TCs
- Localizes faults (a variant of delta debugging for failing TCs)
- Minimizes rerunning (ignores injections that are redundant)
- Provides fault classification and criticality (Element disappears; Not persisted; User setting lost; Crash; Silent fail; Unexpected screen; etc.)

THOR

Evaluation

- Works for Robotium (and Espresso) test suites
- 4 open-source Android apps (with a total of 507 tests)
- 429 tests of a total of 507 fail in adverse conditions
- revealed 66 distinct problems
- 18 of the 22 critical bugs found by Thor are not crashes

THOR

WHAT'S NEXT?

ANDROID

- Activities in isolation - business logic, unit testing
- Message passing between activities (Intents), integration testing
- Explore application GUI; guided/random exploration;
- Activity/Fragment/app Life-cycle changes/related interactions
- Interaction with OS, sensors
- Interactions with other apps and services, web info
- Security/Privacy/Energy testing (not covered here)
- Distributed testing (run on multitude of read devices and simulators)

Business logic

User/sensor interactions

OS interactions

WHAT TO TEST AND HOW?

[Choudhary, Gorla and Orso ASE 2015]

[P. Kochhar, F. Thung, N. Nagappan, T. Zimmermann, and D. Lo. ICST 2015]

- Android emulators are slow and unstable
- "Flaky
- Input
- Supp
- GUI GUI"), yet al
- Isolat
- State-sensitivity and state explosion

Developer is not provided with tools or models to manage the state configuration during design time, while testing approaches seek to explore the visible state and basic system interactions

CHALLENGES

- Activity state (Paused, Resumed, etc.)
- Activity's
- Shown V e)
- Model (b
- BackStac Task)
- Backgrou
- Sensors, screen orientation

Business
ogic

/sensor
actions

OS
interactions

Aggregate state is extremely large

App models are needed to reason about these states
and
automatically generate test cases to cover state combinations

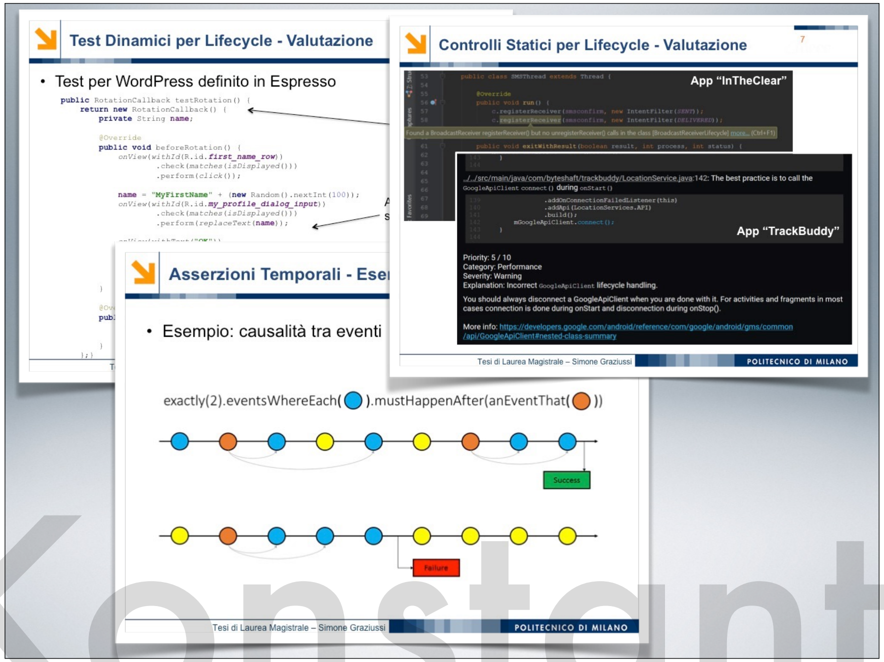
ACTIVITY STATE

CURRENT WORK @ POLITECNICO

COMPREHENSIVE EVENT-BASED TESTING

- Static analysis for: resource release, best practices, double instantiation (e.g., location acquire/release)
- Framework for lifecycle testing, works with Espresso/Robotium (e.g., test app while after Activity.onCreate())
- Temporal assertion language for event-based testing works with Espresso/Robotium built on RxJava (express causality and order)

https://github.com/Simone3/Thesis by Simone Graziussi

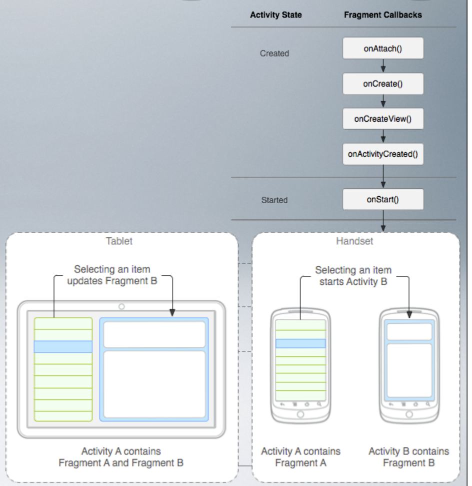


RESEARCH PROPOSALS

Static analysis: integrating with automated program repair; novel dynamic/ adaptive interface checks

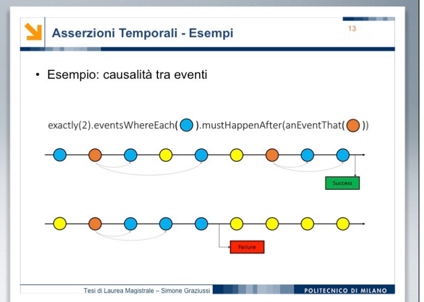
RESEARCH PROPOSALS

Lifecycle testing: automatic test case generation; novel mechanisms for dynamic testing of Fragments



RESEARCH PROPOSALS

Temporal assertion generation: automated assertion placement; automated collection of oracles for temporal assertions; automated test case generation




RESEARCH PROPOSALS
Layout/fragmentation issues:
• automated testing dynamic/ adaptive interfaces;
• automated generation of layout oracles and constraints;
• optimal device selection for dynamic interface testing
Stats
Insights
Wednesday
9:00 PM


find me:
http://futurezoom.in
email: konstantin.rubinov@polimi.it