

TODO
2015-2016
TODO

836897 Simone Graziussi

TODO

Abstract

TODO

Contents

I	Introduction	3
1	Abstract	3
2	Testing	3
3	Android	4
4	Android App Components	4
5	Android Testing State of the Art	5
6	Assertions in Android	7
7	Introduction to Events	8
8	Background/Motivation	8
II	Lifecycle Testing	8
9	Lifecycle	8
9.1	Component Lifecycle	8
9.2	Activity Lifecycle	8
9.3	Fragment Lifecycle	9
9.4	Managing the Lifecycle	10
9.5	Existing Lifecycle Testing	10
10	Static Analysis	11
10.1	Static Program Analysis	11
10.2	Android Lint	12
10.3	Custom Android Lint Checks [TODO move just before im- plementation?	13
10.4	Static Lifecycle Checks	16
10.4.1	Target Components	16
10.4.2	Design	16
10.4.3	Implementation	16
10.4.4	Evaluation	16
11	Dynamic Analysis	16
11.1	Design	16
11.2	Implementation	16
11.3	Evaluation	16

III	Event-based Testing	16
12	Event-based Systems	16
12.1	Android as an Event-Driven Architecture	16
12.2	Event Concurrency Errors	17
12.3	Existing Event Testing	18
13	Temporal Assertions Language	19
13.1	Consistency Checks	19
13.2	Checks on Single Events	19
13.2.1	Can Happen Only After	19
13.3	Checks on Sets of Events	21
13.3.1	Must Happen After	21
13.4	Checks on the Whole Stream	23
13.5	Connectives between Checks	23
14	Reactive Programming	23
14.1	ReactiveX	23
14.2	Components	23
14.3	RxJava and RxAndroid	23
14.4	Events Observable in Android	23
15	Design	23
16	Implementation	23
17	Evaluation	23
IV	Conclusion	23
18	Recap	23
19	Future Work	23

Part I

Introduction

1 Abstract

2 Testing

Testing is the process used in software development to assess the validity of functional and non-functional requirements of an application. Although this verification is able to guarantee the correctness of the tested components within the specific conditions described by the test cases, testing cannot assess the validity of the whole application in every situation, because this would require an unfeasible amount of detail. For this reason, testing is mainly used to discover software bugs in particular situations and to reach an acceptable confidence that the program works as expected under the most common conditions.

Test cases require a mechanism to actually determine the test outcome, i.e. to tell if the application behaves as expected during the validation process. This mechanism, called oracle, should ideally be complete but avoiding over-specification, while also being efficiently checkable [5]. Oracles can assume many forms, for example the behavior of the application is compared with the technical specifications (e.g. documentation), it is automatically checked by the system thanks to some constructs that allow the developer to specify the test conditions or it can even be manually evaluated by a human being.

Test cases can be designed from different points of view. In particular, we can have:

- White-box testing: the focus is on the internal structure of the application, i.e. tests are defined at the source code level (*how* the software behaves).
- Black-box testing: examines the external behavior of the application without considering the actual implementation, i.e. tests are defined at the user level (*what* the software does).
- Grey-box testing: combination of white-box and black-box testing. Tests are defined with a partial knowledge of the internal structure of the application (i.e. how the main components interact and the general algorithms used).

There are usually four levels on which test cases can be defined:

- Unit Testing: focuses on a specific unit of the program, for example a single function/method/class used in the source code. Usually, it follows a white-box testing approach and it is performed during development to build a program using units guaranteed to work.
- Integration Testing: tests interactions between components of the application, i.e. it usually puts together the units tested in the previous step to see if they work well together.
- System Testing: it considers the program as a whole to see if it meets all requirements and quality standards.
- Acceptance Testing: final step that decides if the application is complete and ready to be deployed (e.g. released to the public)

3 Android

Android is a operating system (OS) developed by Google. It is designed primarily for touchscreen mobile devices like smartphones and tablets, but recently it was extended to televisions (Android TV), cars (Android Auto) and smartwatches (Android Wear).

The OS is a multi-user Linux system in which each app is a different user. It is characterized by the so called sandboxing mechanism: each process runs in its own virtual machine (VM) and so every app runs in isolation from the other applications. This means that, by default, an application can access only a limited set of components and cannot access parts of the system for which it does not have permissions.

Android applications are developed in a Java language environment. The Android Software Development Kit (SDK) compiles code, data and resources into a package called APK, which is used by the devices to install the application.

4 Android App Components

Android applications are built by five main components, each with its specific purpose:

- Activity: an activity represents a single “action” that the user can take and, since almost all activities interact with the user, they provide a screen with a user interface. Each activity in the application is independent from the others, but it is of course possible to start an activity (for example when the user clicks on a button) from another to build the application flow.

- **Fragment:** this component was introduced in Android 3.0 (API level 11) to support dynamic and flexible UI on large screens, for example on tablets. A Fragment represents a “portion” of an Activity, with its own state and UI. Each Activity can contain multiple Fragments at a time, Fragments can be added/removed at runtime and each Fragment can be reused in more than one Activity. A Fragment can only be instantiated inside an Activity.
- **Service:** a service is a component that is executed in background and, as such, it provides no user interface. It is used to perform complex computations or to interact with an external API (e.g. via the network). The advantage of this approach is that another component (e.g. an Activity) can start and interact with a Service in order to avoid blocking its UI with computationally intensive operations.
- **Content Provider:** a content provider allows to store and retrieve data in some persistent storage location, for example a local SQL database or a remote repository. The provided data can be shared among different applications or private to a specific one.
- **Broadcast Receiver:** a broadcast receiver responds to global messages, i.e. messages received by all applications on the device. These events may be fired by the system (e.g. the device has just rebooted) or by a single application (e.g. some data is available), and then intercepted by the applications interested to them.

Activities, services and Broadcast Receivers are started asynchronously by messages called Intents. This allows not only an application to start its own components, but also to call on other applications. For example, a game may start its internal GameService to manage the game loop, but also send an intent to a social network application to share the game progress.

5 Android Testing State of the Art

Android tests are mainly based on JUnit [1], a testing framework for Java. It allows to create classes called test cases that contain methods annotated with `@Test`, each representing a test.

White-box unit testing in Android can be

- **Local:** it runs on the local development machine (i.e. the computer where the application is coded). It has the advantage of being fast (avoids the overhead to load the application in a device/emulator), but can be exploited only if the tested unit has no dependencies or simple dependencies. This means that the test case should not use

any device-specific features (e.g. expect a sensor input) or, if it does so, they should be minimal since they need to be mocked using for example tools like Mockito [3].

- Instrumented: it is executed in a physical device or on an emulator and so has access to instrumentation information, such as the Context (information about the application environment). It is slower than the previous case but it's more convenient if the unit dependencies are too complex to mock.
- Hybrid: the external library Robolectric [4] tries to take the advantages of the previous two approaches, i.e. it runs “instrumented” tests on the local machine, without mocking. As reported on the website, Robolectric allows a test style that is closer to black box testing, making the tests more effective for refactoring and allowing the tests to focus on the behavior of the application instead of the implementation of Android.

Android also provides a way to test User Interface (UI) to see if it behaves as expected. This type of testing can be defined as a grey-box approach: the application is tested at the user level without considering the actual implementation of the UI, but the definition of the test cases may require to know some information on the internal structure, for example the IDs of the buttons to be clicked.

- UI testing on a single app: the Espresso library provides APIs for writing UI tests to simulate user interactions. In general, defining of a test case means building a series of `onView(Matcher).perform(ViewAction).check(ViewAssertion)` instructions, i.e. select a particular View that matches some description (e.g. a button with “Start” text), perform one or more actions on it (e.g. click) and check if some conditions are true (e.g. if after the click the button text changes). The main advantages of Espresso are that the test cases are easily readable and understood, and that it has automatic synchronization (before performing an action it waits for the previous ones to be completed, i.e. for the main thread to be idle).
- UI testing on multiple apps: the UI Automator library allows to test if the developed application interacts correctly with the system or other apps (e.g. the application may request an image, the Camera app is opened, the picture is taken and then the control goes back to the original application).

In addition to this, Android also offers some tools to run and test an application without accessing the source code (black-box testing):

- **UI Exerciser Monkey**: allows to run an application on a physical device or emulator generating a pseudo-random (but repeatable) streams of user events (e.g. clicks) and system-level events (e.g. start call). The developer can set several options like target package, probability of certain events, etc.
- **monkeyrunner**: controls a device or emulator from a workstation by sending specific commands and events defined as a Python program. It also allows to take screenshot during the test execution and store them on the workstation.

6 Assertions in Android

An assertion is a statement at a specific point in a program that enables to check an assumption. It is expected to be true, but if a bug is present the assertion will fail and the system will throw an error. Assertions are test oracles that specify what the application does rather than how.

In Java, the assertion statement is

```
assert Expression1 : Expression2;
```

where **Expression1** is the boolean expression to be checked and **Expression2** is an expression that has a value to better describe the error. If **Expression1** is false, an **AssertionError** exception is thrown.

Many frameworks, including JUnit itself, offer several utility methods to easily write more complex assertions without using the **assert** keyword. For example, the method **assertEquals(String message, Object expected, Object actual)** checks that the two given objects are equal and, if not, throws an **AssertionError** with the given message.

Since Android has its own virtual machine ART, which is not compatible with JVM from Oracle, the **assert** keyword is not supported by default¹. For this reason, in an Android application assertions are usually created exploiting JUnit-like methods that directly throw an **AssertionError** if the check fails. This means for example using the assertions provided by Android (via **Assert**, **MoreAsserts** and **ViewAssert** classes), the JUnit framework itself or other external libraries.

¹It is however possible to manually enable assertions on a given device using for example the Android Debug Bridge command **adb shell setprop debug.assert 1** (assertions will be enabled for all applications until the device is rebooted), but the Android developers themselves discourage their use

7 Introduction to Events

8 Background/Motivation

Part II

Lifecycle Testing

9 Lifecycle

9.1 Component Lifecycle

App components in Android such as Activities, Fragments and Services are characterized by their lifecycle, i.e. the current runtime state. In general each component is started and then destroyed, but some can also be paused and resumed, and go through several other states. When a lifecycle state transition happens, the Android system allows the developers to implement some callbacks in the component implementation to manage their behavior in those particular situations.

9.2 Activity Lifecycle

An Activity can be in three static states:

- Resumed: the Activity is visible and can receive user input
- Paused: the Activity is *partially* hidden by another visual component, for example a notification dialog, and has lost the focus. When the activity is paused it cannot receive any user input or execute code.
- Stopped: the Activity is completely hidden to the user, i.e. it is in the background. Like in the previous case, the activity cannot receive inputs or run code. In this state the Activity is still “alive”: the state (e.g. member variables) is retained and the Activity can be later restarted.

An Activity can also be in two transient states:

- Created: the Activity has been instantiated and will soon become Started
- Started: the Activity has been initialized and will soon become Resumed

Finally, an Activity can be Destroyed. In this “state” the Activity instance is dead. An Activity can for example be destroyed by the OS when it is

stopped and the device needs to free resources.

The Activity class provides several callbacks for lifecycle changes. These methods can be implemented by the developer to manage the state of the Activity: for example `onCreate()` can be used to initialize the UI components, `onStop()` to free resources for the other applications. Note that not all callbacks need to be explicitly implemented for each activity: actually simple Activities usually implement only `onCreate()`.

This image shows a simplified scheme of the Activity lifecycle transitions with the corresponding callbacks.

[TODO extend with more detailed explanations of callbacks, save instance, etc.?] [TODO extend with more detailed diagram?]

9.3 Fragment Lifecycle

The lifecycle of a Fragment is closely related to the lifecycle of the Activity that contains it: for example, when an activity is paused, all the contained Fragments are paused too. In addition to this, however, Fragments can go through lifecycle changes independently of their host Activity: in particular, since fragments can be dynamically added and removed at runtime, they can be created and destroyed while the Activity is running. Moreover, the developer also has the option to store removed fragments in the so-called backstack and be able to restore them later for example when the user presses the “back” button.

Like Activities, Fragments are characterized by three static states:

- Resumed: the Fragment is visible and can receive user input
- Paused: the Activity that contains this Fragment is *partially* hidden by another visual component and has lost the focus.
- Stopped: the Fragment is completely hidden to the user. This can happen if the host Activity is also not visible (i.e. in background) or if the Fragment has been stored in the backstack. Like Activities, stopped Fragments are still “alive” and their state is retained.

Lifecycle management for Fragments is very similar to the one for Activities, since all callbacks are the same. Fragments provide however some additional methods to manage the interaction with the host Activity: for example `onAttach()` is called when the Fragment is linked to an Activity, `onCreateView()` when the Fragment is ready to build its UI, etc.

9.4 Managing the Lifecycle

Handling the components lifecycle is a critical aspect in developing an Android application and it is often source of bugs or unexpected behaviors. For example, properly implementing Activity/Fragment lifecycle methods ensures that the app

- does not waste system resources (e.g. device sensors) while the user is not interacting with it
- stops its execution when the user leaves the application (for example a game should pause if the user opens another application or a music player should pause when the user receives a phone call)
- retains its state if the user leaves and then returns to the application (e.g. a messaging app must keep a partially written message even if the user puts the app in background for a moment)
- does not crash or loses user progress when lifecycle changes occur (e.g. an app that does not correctly manage lifecycle may crash with a `NullPointerException` if some internal component was destroyed during `onStop()` but not restored during `onStart()`)
- adapts to configuration changes (like a device rotation between landscape and portrait modes) without losing data or crashing

In general, for Activities and Fragments the developer should make sure to

- during `onCreate()`
 - ... [release all remaining resources in `onDestroy()`]

9.5 Existing Lifecycle Testing

Due to the importance of lifecycle handling, some approaches to test applications focusing on this aspect have been developed.

One example is THOR [?]. The idea of the tool is to run pre-existing UI test cases (defined in Robotium or Espresso) in adverse conditions to test their robustness. These adverse conditions are not however unusual events, but common expected behaviors of the application. In particular, THOR injects in the tests several neutral system events, i.e. events that are not expected to change the outcome of the test. These neutral events are mainly related to Activity lifecycle: for example `Pause` \rightarrow `Resume`; `Pause` \rightarrow `Stop` \rightarrow `Restart`; `Audio focus loss` \rightarrow `Audio focus gain`. Neutral events injected in an application that does not manage the lifecycle correctly can lead to the discovery of bugs, which are not necessarily crashes but also unexpected behaviors for the specific application.

The tool provides several interesting features like

- Neutral events are injected in suitable locations to avoid conflicts with the test case: in particular they are triggered when the event queue becomes empty and the execution of the remaining test is delayed.
- Multiple errors for each test: if a test fails after some neutral event injections, the test is rerun but injections are only performed after the previous failure point to maximize error detection
- Faults Localization: if a test fails, the tool tries to identify the exact causes (i.e. the neutral events responsible for the unexpected behavior) using a variant of delta debugging (scientific approach of hypothesis-trial-result loop), then displays this information to the user to allow further investigation
- Faults Classification: the errors that make the test cases fail are classified by importance and criticality
- Customization: the developer can select the set of tests to run, the set of neutral event sequences to take into account, and several other different variations

While very interesting and useful for bug detection, THOR is not a very user-friendly tool. First of all the tests are run via an external program (and so the developer is not able to simply run the tests via an IDE like Android Studio) that is only available for Linux and its installation is not immediate. Moreover, THOR only executes the test cases on an emulator running Android KitKat 4.4.3, which does not leave any choice to the developer on which version of Android to test.

10 Static Analysis

10.1 Static Program Analysis

As opposed to dynamic analysis that requires to actually run an application, static analysis only inspects the source code. One of the techniques for static analysis is to build the Abstract Syntax Tree (AST) of the application, i.e. the tree representation of the code structure, and visit it starting from the root. In the AST each node represents a syntax construct: for example in Java we may have class declaration nodes, method invocation nodes, branch nodes, etc.

An example of static analysis tool for Java is the one integrated in IntelliJ IDEA (on which Android Studio is based) that allows for example to

- find probable bugs (e.g. using an object that may be null)
- locate “dead” code (unused/unreachable code that only makes the application heavier)

- detect performance issues (e.g. suggest ways to implement “tail recursion” in a recursive method)
- improve code structure and maintainability (code dependencies)
- conform to coding guidelines, standards and specifications

10.2 Android Lint

Android Lint [2] is a static analysis tool that scans Android projects for potential bugs, performance, security, usability, accessibility and internationalization issues, and more. It is an IDE-independent analyzer, at the moment integrated with Eclipse and Android Studio.

Some of the checked issues are run directly when the user is writing the code and shown via an in-line warning. A more complete analysis can be performed by explicitly running the tool on the whole project, via terminal command or directly from the IDE options. The results are then presented in a list with a description message and a severity level, so that the developer can easily identify the most critical problems and understand their causes. It is also possible to customize the checker for example by specifying the minimum severity level of the detected issues and by suppressing some specific checks if the developer is not interested.

Android Lint provides more than 100 built-in checks. Some examples:

- `MissingPermission` (correctness issue): lint detects that a call to some method (e.g. store a file on the SD card) requires a specific system permission (e.g. access to external storage) that has not been included in the application manifest
- `WrongThread` (correctness issue): checks that the methods that must run on the UI thread (e.g. manipulation of a View component) are actually called there
- `SecureRandom` (security issue): detects random numbers generated by fixed seeds, usually used only during debugging
- `UnusedResources` (performance issue): a resource like an image, a string, etc. is not used in the application and so it can be deleted to free space
- `UselessParent` (performance issue): a layout file contains a View component that is of no use and can be removed for a more efficient layout hierarchy
- `ButtonOrder` (usability issue): makes sure that “cancel” buttons are placed on the left of the UI component, to follow the Android Design Guideline

- `ContentDescription` (accessibility issues): checks that important visual elements like image-buttons have a textual description to allow the system accessibility tools to describe their purpose
- `HardcodedText` (internationalization issue): makes sure that text strings displayed to the user are placed on the appropriate xml files to allow translation and not hard-coded in Java

10.3 Custom Android Lint Checks [TODO move just before implementation?]

Important note: the Lint API is not final and mostly undocumented. This means that what is reported below is based only on a few examples/tutorials found on the Internet, on the built-in checks implementations and on source code inspection, and not on official documentation. Moreover, since it is not final, future releases of the tool may invalidate the following statements and the custom implementations.

The open source Android Lint API allows to build custom rules for the static analyzer.

Implementing a custom Lint rule means building four components:

- **Issue:** a problem detected by the rule, characterized by properties like ID, explanation, category, priority, etc. For example the “Missing-Permission” issue has “9/10” priority, “Error” severity, “Correctness” category and an explanation describing the problem to the developer.
- **Detector:** a detector is in charge of identifying one or more issues (if more than one, they are independent but logically related). For example the built-in `ButtonDetector` identifies the “Order” (cancel button on the left), “Style” (button borders), “Back” (avoid custom back buttons) and “Case” (capitalization of “OK” and “Cancel” labels on buttons) issues.
- **Implementation:** links an issue to a detector and specifies the rule scope.
- **Registry:** it is simply in charge of registering the issues to allow the Lint tool to identify the custom rules.

More in detail, a custom detector extends the `Detector` class and implements one (or more in special cases) of these interfaces:

- `XmlScanner` if it needs to analyze XML files (such as the application manifest or layout descriptions)

- **JavaScanner** if it needs to analyze Java files (source code for classes)
- **ClassScanner** if it needs to analyze Class files (compiled Java files)

The extended class and the interfaces provide some utility methods that can be overridden by the developer to filter applicable files (e.g. name contains a given substring), nodes (e.g. only variable declarations), elements (e.g. an XML scanner only wants to read **TextView** elements), etc. to focus the rule attention only on particular situations and improve performance.

In particular, a detector that implements **JavaScanner** is able to visit the Java files via an Abstract Syntax Tree, represented with the **lombok.ast** API. The developer has two options:

- Use the **Detector** callback methods to visit the nodes using the default AST Visitor. First of all, if the detector is interested only in specific types of nodes in the AST (as it is usually the case) the developer should override the **getApplicableNodeTypes()** method and return a list of types (e.g. **ClassDeclaration**, **MethodInvocation**, etc.). The class also provides several other methods to focus the search, like **getApplicableMethods()** (return list of method names), **applicableSuperClasses()** (return list of super-classes names), etc. Once this is done, the developer can override the detector methods like **visitMethod()** to receive all matching method invocations found in the tree, **checkClass()** to receive the matching class declarations, etc. to implement the rule logic.
- Return in **createJavaVisitor()** method a custom implementation of the AST Visitor (subclass of **AstVisitor**) that implements the rule logic. Inside the AST Visitor the developer can override one or more methods that allow to visit every type of node in the tree such as **visitMethodDeclaration()**, **visitVariableDeclaration()**, **visitAnnotation()**, **visitWhile()**, etc.

If a problem is found during one of the callbacks, the detector can call the **report()** method to pass the issue, the location (i.e. file and line) and a message in order to show the warning to the user.

Note that some detectors can just visit the nodes and immediately recognize and report an error (e.g. if an attribute of a given component is not set), but others may require to do more expensive computation (even across multiple files). If this is the case, the developer can use the detector's **afterProject()** hook that is called when the whole project has been analyzed. In order to decide if a problem is present or not, the developer must of course be able to store the state of the computation: variables like boolean flags can be easily stored in the detector object, but the problem is with code locations. Computing the location of an issue is an expensive

operation and, especially if the probability of error is very low, we may have performance issues (e.g. a detector that finds unused resources cannot store the location of each of them before finding out that they are actually used somewhere). To solve this issue we can store location handles (lightweight representations of locations that can later be fully resolved if the problem is actually present) or request a second pass to the Lint tool (i.e. in the first project analysis one only sets some flags to detect problems, then if that is the case the project is scanned again to gather the actual locations of the issues).

As a side note, we can also mention the `ControlFlowGraph` class available during detection. It allows to build a low-level Control Flow Graph (all paths that may be traversed) containing the *insns* nodes of a method, i.e. the RTL (Register Transfer Language) representation of the code where each *insns* node is a bytecode instruction (e.g. jump). This Control Flow Graph can be useful for example to analyze if some component is always released, e.g. the built-in `WakelockDetector` uses it to see if, when the Wake-Lock (a lock to keep the device awake) is acquired in a method, it is released afterwards in every possible path.

Issues are usually defined as public, final and static fields inside their detector class. Issues are simply objects of the `Issue` class that are instantiated calling `Issue.create()` with some parameters like ID, category, severity, etc.

Implementations are objects of the `Implementation` class, whose constructor requires the detector class and the scope (e.g. `Scope.JAVA_FILE_SCOPE`). The implementation is passed as the last parameter of the `Issue.create()` method to link the issue with the detector.

Finally, registers are sub-classes of `IssueRegistry` that usually only override the `getIssues()` method to return the list of custom issues. The registry must be referenced in the `build.gradle` file (in Android Studio) or in a manifest file (in Eclipse) to allow Lint to find it.

Once every component has been implemented, to actually include the Lint check in the list of rules enforced by the tool one must copy the generated JAR file of the library in the `~/.android/lint/` directory. Using Gradle in Android Studio, one can write a snippet to automatically copy the JAR after each *install* command.

10.4 Static Lifecycle Checks

10.4.1 Target Components

10.4.2 Design

10.4.3 Implementation

10.4.4 Evaluation

11 Dynamic Analysis

11.1 Design

11.2 Implementation

11.3 Evaluation

aaa

Part III

Event-based Testing

12 Event-based Systems

Event-Driven Architecture is a software pattern where the focus is on generation and reaction to events. An event can be defined as a message generated by a producer that represents a change of state or a relevant action performed by some component/actor (e.g. the user). Once generated, events are sent via event channels to all the consumers that are interested to them. Event-Driven Architecture are

- extremely loosely coupled: the producer does not know about the consequences of its events. It just generates them and then it's up to the consumers to manage everything else.
- well distributed: an event can be anything and exist almost anywhere

12.1 Android as an Event-Driven Architecture

Android is implemented as an event-based model. This is because a device must manage several events, like clicks on the touchscreen, sensor data, network requests/responses, etc.

From an application point of view, events can be generated internally (e.g. from a service/thread created by the application itself) or externally (e.g. sensor data). Each application is composed of several threads, a subset of

which, called Looper threads, are in charge of processing events by invoking an appropriate event Handler for each of them.

Events from a single thread are atomic, i.e. they are placed on a FIFO event queue and processes one by one. However, events produced by several threads are processed concurrently and not guaranteed to be ordered or atomic. For example the user may click on a button while the application receives a network response and the device broadcasts some sensor data.

Examples of events that can be observed in an Android application:

- User input events can be observed attaching listeners to View components. An event listener is an interface whose methods are called by the Android framework when the View is triggered by user interaction, for example the `onClick()` callback of a listener attached to a Button will be called when the user clicks on that button. The developer can then implement the callback to perform any action in response to the input event generation, e.g. starting another activity.
- Lifecycle events are triggered when there is a state transition of some component, like an Activity. For example the `onPause()` callback is invoked when an Activity is paused.
- Broadcast events like system messages (e.g. device boot completed, alarm goes off, etc.) or application-specific messages are received by BroadcastReceivers in the `onReceive()` callback.
- Asynchronous messages from services external to the application (e.g. sensor data, network requests, etc.) also have their specific callbacks

And many more.

12.2 Event Concurrency Errors

The concurrency among events inside a single application can lead to unexpected behaviors or crashes.

When the Android developer decides to explicitly create a multi-threaded application we may have the classical problems of concurrency like synchronization, deadlocks and starvation, but problems arise even if he/she does not manually create threads, due to the aforementioned concurrency of events.

The most relevant problem in an Android application is race conditions: two or more events do not happen in the order the programmer intended. A simple example of this issue is when the developer asynchronously starts an `AsyncTask` (an utility class that allows to perform short operations in the background without blocking the UI thread) to download an image from the network. The `AsyncTask`, when the download is completed, expects the

Activity that started it to be still there to receive the result, but maybe the user already left the application and so we encounter for example a `NullPointerException`.

12.3 Existing Event Testing

Due to the relevance of the race conditions problem, several research studies tried to provide a way to detect them.

For example, CAFA [7] is a tool that allows to detect use-free races. The authors note that thousands of events may get executed every second in a mobile system and that, even if they are processed sequentially in one thread, most of them are logically concurrent. These concurrent events could be commutative with respect to each other, i.e. the result is the same even if they are executed in a different order. The tool tries to determine if two events are commutative or not, restricting the focus on use-after-free violations (a reference is used after it has been freed, i.e. it does not point to an object anymore). If two events where one uses and the other frees a reference are logically concurrent, they must be non-commutative. To detect possible racy conditions the tool analyzes the traces of the low-level read and write operations, as well as certain branch instructions. While CAFA was a good starting point of race condition analysis, its main problems are that it's quite slow (it may take hours to analyze an application), only focuses on a very specific type of race condition and the tool is not publicly available.

Android EventRacer [6] is an improvement of CAFA, defined by the authors as the first scalable analysis system for finding harmful data races in real-world Android applications. EventRacer not only analyzes use-free races, but also other types of race conditions like:

- **Data Races Caused by Object Reuse:** list components in Android, like `ListView`, usually reuse rows while the user scrolls to improve performance (no need to create as many rows as the number of data values). If data for each row is loaded asynchronously, it may happen that the wrong data is loaded in a row (the user scrolled the list in the meanwhile and the row has already been reused).
- **Data Races Caused by Invalidation:** these are the races similar to the first example, i.e. the `AsyncTask` completing when the `Activity` no longer exists
- **Callback Races:** different listener callbacks may be invoked in any order. For example the developer may asynchronously create a `GoogleMap` object and request location updates from the device GPS. He/she may expect the `onMapReady()` callback to be executed before the first `onLocationChanged()` callback (and so use the map to place a marker at the received location), but it might not be the case.

Moreover, EventRacer analyzes an application in much less time than CAFA: building the Happens-Before relationship graph has $\mathcal{O}(n^2)$ time complexity instead of $\mathcal{O}(n^3)$. The tool is also publicly available as an online tester or an offline application (only available for Linux systems). One disadvantage of the EventRacer approach is that the implementation requires to modify the Android framework in order to access low-level information and, for this reason, it is only available for Android 4.4, which limits the options of the developer for testing other versions of the OS.

A completely different approach for race detection is provided by DEvA [8]. The idea of the tool is to base its search on static code analysis rather than dynamic analysis (CAFA and EventRacer run the application on emulators, interact with it via “robots” and then retrieve the traces to be analyzed): this approach of course guarantees more code coverage and completeness. DEvA focuses on a specific type of problem called Event Anomalies: processing of two or more events results in accesses to the same memory location and at least one of those is a write access (note that the use-free races analyzed by CAFA are a subclass of this type of issue). The idea of the tool is to identify variables that may be modified as a result of receiving an event (i.e. a potential Event Anomaly) using the Control Flow Graph of the application (i.e. all paths that may be traversed in the application during its execution). The tool receives as input from the developer

- the list of all methods used as event handlers (callbacks that use the events)
- the base class used to implement events in the system
- the set of methods used as consumed event revealing statements (i.e. methods that retrieve information stored in an event without modifying the event’s attributes), used to tell apart events when a general parameter is passed to a callback

DEvA allows a very fast analysis (usually 1 or 2 minutes) and guarantees complete code coverage, but static analysis may report false positives or be unable to detect some anomalies.

13 Temporal Assertions Language

13.1 Consistency Checks

13.2 Checks on Single Events

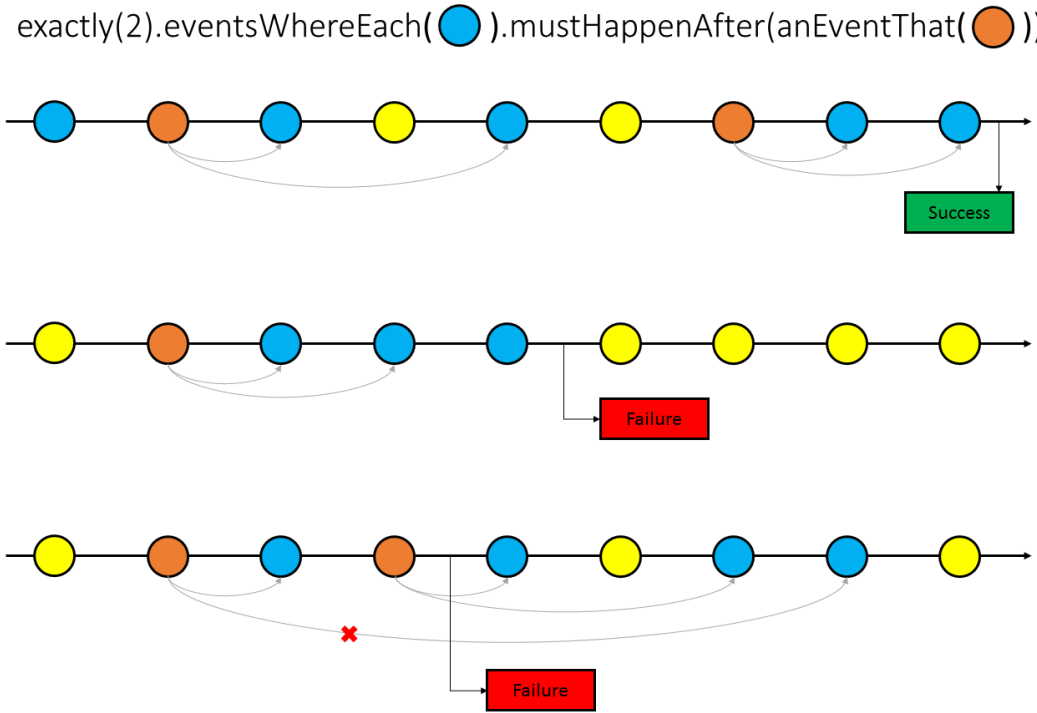
13.2.1 Can Happen Only After

Structure	<code>anEventThat(m1).canHappenOnlyAfter(anEventThat(m2))</code>
Description	Checks that the events that match <code>m1</code> happen only after any event that matches <code>m2</code> , i.e. there cannot be an event that matches <code>m1</code> before the first event that matches <code>m2</code> .
FOL	$\forall e1 \left(match(e1, m1) \Rightarrow \exists e2 \left(match(e2, m2) \wedge before(e2, e1) \right) \right)$
Visual	<p><code>anEventThat(●).canHappenOnlyAfter(anEventThat(●))</code></p> <p>Note in particular that in the second case the checks succeeds even if no blue event occurs: we are just saying that blue events <i>can</i> happen after an orange one.</p>
Code Example	<pre> anEventThat(isMarkerPlacement()) .canHappenOnlyAfter(anEventThat(isMapReady())) </pre>

13.3 Checks on Sets of Events

13.3.1 Must Happen After

Structure	<code>exactly(n).eventsWhereEach(m1).mustHappenAfter(anEventThat(m2))</code>
Description	Checks that exactly n events that match m1 happen exclusively after every event that matches m2 . “Exclusively” means that there cannot be another event that matches m2 before the sequence of n events is completed.
FOL	$\forall e2 \left(match(e2, m2) \Rightarrow \exists_n e1 \left(match(e1, m1) \wedge before(e2, e1) \right. \right.$ $\left. \left. \wedge \neg \exists e2' \left(match(e2', m2) \wedge between(e2, e2', e1) \right) \right) \right)$

Visual	<p> <code>exactly(2).eventsWhereEach(●).mustHappenAfter(anEventThat(●))</code> </p>  <p>Note in particular that the third case shows the “exclusively” constraint mentioned before: the check fails because we do not have two blue events after the first orange but before the second one (i.e. the first orange event does <i>not</i> match one of the three blue events that follow the second orange event)</p>
Code Example	<pre> exactly(1).eventsWhereEach(isTextChangeFrom(locationTextView)) .mustHappenAfter(anEventThat(isLocationChange())) </pre>

13.4 Checks on the Whole Stream

13.5 Connectives between Checks

14 Reactive Programming

14.1 ReactiveX

14.2 Components

14.3 RxJava and RxAndroid

14.4 Events Observable in Android

15 Design

16 Implementation

17 Evaluation

Part IV

Conclusion

18 Recap

19 Future Work

References

- [1] JUnit. <http://junit.org>.
- [2] Lint. <http://tools.android.com/tips/lint>.
- [3] Mockito. <http://mockito.org/>.
- [4] Robolectric. <http://robolectric.org/>.
- [5] Luciano Baresi and Michal Young. Test oracles. Technical Report CIS-TR-01-02, University of Oregon, Dept. of Computer and Information Science, Eugene, Oregon, U.S.A., August 2001. <http://www.cs.uoregon.edu/~michal/pubs/oracles.html>.
- [6] Pavol Bielik, Veselin Raychev, and Martin Vechev. Scalable race detection for android applications. *SIGPLAN Not.*, 50(10):332–348, October 2015.

- [7] Chun-Hung Hsiao, Jie Yu, Satish Narayanasamy, Ziyun Kong, Cristiano L. Pereira, Gilles A. Pokam, Peter M. Chen, and Jason Flinn. Race detection for event-driven mobile applications. SIGPLAN Not., 49(6):326–336, June 2014.
- [8] Gholamreza Safi, Arman Shahbazian, William G. J. Halfond, and Nenad Medvidovic. Detecting event anomalies in event-based systems. In Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, pages 25–37, New York, NY, USA, 2015. ACM.