Politecnico di Milano
Computer Science and Engineering

Scuola di Ingegneria Industriale e dell'Informazione
Computer Science and Engineering

Lifecycle and Event-Based Testing for Android Applications

Advisor: Luciano BARESI
Co-Advisor: Konstantin RUBINOV

Master thesis by:
Simone GRAZIUSSI matr. 836897

Academic Year 2015-2016

Abstract

Mobile market is continuously expanding, and mobile applications require higher and higher quality standards. The most cost-effective way a developer can obtain confidence of correctness and usability in a set of specific circumstances is testing.

This thesis analyzes two particular challenges of mobile testing that have been largely overlooked to date, lifecycle management and event concurrency, focusing its attention on the Android operating system.

Application components are characterized by their lifecycle, i.e. by several working states such as running, paused, destroyed, etc. This mechanism needs to be carefully handled by the developer to avoid crashes or unexpected behaviors, but the available means to test the applications with regard to this aspect are lacking. The proposed work addresses this issue first with a static code analysis approach to recognize possible misuses of components, and then with a dynamic technique that allows the developer to drive the application lifecycle to test its robustness.

In addition to lifecycle transitions, mobile applications are characterized by a great number of other events, hundreds of which may be processed every minute, such as sensor data, connectivity changes, user input, network responses, etc. These events can happen in many different orders and frequencies, and, if the developer does not handle them correctly, issues may arise. The second part of this thesis describes an event-based testing approach that allows to observe several events during the application execution and to express consistency checks on their stream by means of a proposed temporal assertions language.

Together, the thesis forms a comprehensive framework to automating testing of events in Android applications, allowing to express existence, ordering, causality and quantification conditions on them, as well as to better control the critical lifecycle transitions.

Estratto

Il mercato dei dispositivi mobili è in continua espansione, e le applicazioni richiedono standard di qualità sempre più alti. Il modo migliore per ottenere una sufficiente confidenza che l'applicazione si comporti nel modo desiderato in determinate situazioni è il testing.

Questa tesi analizza due aspetti critici del testing per dispositivi mobili, la gestione del lifecycle e la concorrenza tra eventi, prendendo il sistema operativo Android come esempio.

I componenti delle applicazioni sono caratterizzati dal loro lifecycle (ciclo di vita), cioè diversi stadi come in esecuzione, in pausa, ecc. Questo meccanismo deve essere gestito con attenzione per evitare crash o comportamenti inaspettati, ma le tecniche disponibili per testare questo aspetto sono carenti. Questo documento si occupa di tale problema prima attraverso un'analisi statica del codice per riconoscere possibili criticità, e successivamente tramite una tecnica dinamica che permette allo sviluppatore di controllare il lifecycle dell'applicazione per testarne la robustezza.

In aggiunta alle transizioni del ciclo di vita, le applicazioni per dispositivi mobili sono caratterizzate da un gran numero di altri eventi, che possono essere elaborati a centinaia ogni minuto, come informazioni dai sensori, cambi di connettività, attività da parte dell'utente, ecc. Questi eventi possono essere registrati in molti ordini differenti e, se lo sviluppatore non li gestisce in modo corretto, possono insorgere dei problemi. La seconda parte di questa tesi descrive una tecnica per il testing che permette di rilevare diversi eventi durante un'esecuzione dell'applicazione e di esprimere delle condizioni di consistenza su di essi tramite un linguaggio di asserzioni temporali.

Nell'insieme, la tesi costituisce una soluzione comprensiva per il testing di eventi in applicazioni Android, offrendo la possibilità di esprimere condizioni di esistenza, ordinamento, causalità e quantificazione su di essi, così come di controllare meglio le transizioni critiche del ciclo di vita.

Contents

I Introduction **11**

1 Context **11**

2 Objectives **11**

3 Achieved Results **12**

4 Outline of the Thesis **12**

II Background **14**

5 Introduction **14**

6 Testing **14**

7 Android **15**
7.1 Operating System 15
7.2 Application Components 16
7.3 Component Lifecycle 17
7.3.1 Activity Lifecycle 17
7.3.2 Fragment Lifecycle 18
7.3.3 Managing the Lifecycle 19

8 Event-Based Systems **22**
8.1 Event-Driven Architecture 22
8.2 Android as an Event-Driven Architecture 22
8.3 Event Concurrency Errors 24

III Android Testing State of the Art **25**

9 Unit Testing **25**

10 UI Testing **26**

11 Runner Tools **27**

12 Lifecycle Testing **27**
12.1 Testing Frameworks Support 27
12.2 THOR 28

13 Race Conditions Testing **29**

13.1 CAFA . . . 29
13.2 EventRacer . . . 29
13.3 ERVA . . . 30
13.4 DEvA . . . 31

14 Addressed Limitations **32**

IV Lifecycle Testing **33**

15 Introduction **33**

16 Static Analysis **33**
16.1 Introduction . . . 33
16.2 Static Program Analysis . . . 33
16.3 Static Lifecycle Checks . . . 34
16.4 Target Components . . . 35
16.5 Design . . . 39
16.6 Implementation . . . 40
16.6.1 Introduction . . . 40
16.6.2 Android Lint . . . 40
16.6.3 Custom Android Lint Checks . . . 41
16.6.4 Code Structure . . . 44
16.7 Evaluation . . . 45
16.7.1 Real-World Case Study . . . 45
16.7.2 Discussion . . . 46

17 Dynamic Analysis **46**
17.1 Lifecycle Test Cases . . . 46
17.2 Defined Tests . . . 47
17.3 Design . . . 49
17.4 Implementation . . . 50
17.5 Evaluation . . . 52
17.5.1 Real-World Case Study . . . 52
17.5.2 Discussion . . . 54

V Event-Based Testing **56**

18 Introduction **56**

19 Temporal Assertions Language **57**
19.1 Consistency Checks . . . 57
19.2 Checks on Single Events . . . 58
19.2.1 Can Happen Only After . . . 58

19.2.2 Can Happen Only Before . . . 59
19.2.3 Can Happen Only Between . . . 60
19.3 Checks on Sets of Events . . . 62
19.3.1 Must Happen After . . . 62
19.3.2 Must Happen Before . . . 63
19.3.3 Must Happen Between . . . 65
19.4 Checks on the Whole Stream . . . 66
19.4.1 Match In Order . . . 66
19.4.2 Are Ordered . . . 68
19.5 Existential Checks . . . 69
19.5.1 Exists An Event . . . 69
19.5.2 Exist Events . . . 70
19.5.3 Exist Events After . . . 71
19.5.4 Exist Events Before . . . 72
19.5.5 Exist Events Between . . . 73
19.6 Connectives between Checks . . . 75
19.6.1 And . . . 75
19.6.2 Or . . . 76
19.6.3 Not . . . 77
19.6.4 Single Implication . . . 77
19.6.5 Double Implication . . . 78

20 Design **79**

21 Implementation **80**
21.1 ReactiveX . . . 82
21.2 RxJava and RxAndroid . . . 83
21.3 Events Observable in Android . . . 84
21.4 The System . . . 85
21.5 Event Monitor . . . 85
21.6 Event Generators . . . 88
21.7 Checks . . . 88
21.8 Descriptors . . . 88

22 Evaluation **89**
22.1 Real-World Case Study . . . 89
22.1.1 Structure . . . 90
22.1.2 Events . . . 90
22.1.3 Checks . . . 91
22.1.4 Check Results . . . 92
22.1.5 Performance . . . 93
22.2 Discussion . . . 94
22.2.1 Applicability . . . 94
22.2.2 Effectiveness . . . 94

22.2.3 Usability 95
22.2.4 Performance 96
22.2.5 Extensibility 96
22.2.6 Comparison with Testing Frameworks 97
22.2.7 Comparison with Race Detection Tools 97

VI Conclusions and Future Work **100**

23 Conclusions **100**

24 Future Work **101**

Appendices **103**

A Code Listings **103**

List of Figures **118**

References **121**

Part I

Introduction

1 Context

Mobile market is continuously expanding, in 2016 2 billion people worldwide own a smartphone and stores like Google Play and Apple's App Store host more than 2 million applications [1].

Due to this widespread, mobile applications require high quality standards and the most cost-effective way a developer can obtain confidence of correctness and usability in a set of specific circumstances is testing.

Mobile testing, however, presents several challenges. The main problem is fragmentation: there are hundreds of different devices that may run an application, each with different hardware capabilities, screen sizes, available sensors and input methods, operating systems or versions of the same operating system. Other issues are for example network availability, internationalization, scripting (e.g. emulate touchscreen gestures during a test) and usability.

This thesis focuses on two particular challenges of mobile development that have been largely overlooked to date:

- Events: mobile applications are characterized by a highly dynamic environment, where hundreds of events may be processed every minute, such as sensor data, connectivity changes, user input, network responses, etc. These events can happen in many different orders and frequencies, and, if the developer does not handle them correctly, this may lead to unexpected behaviors or crashes.
- Lifecycle: application components are characterized by their lifecycle, i.e. by several working states such as running, paused, destroyed, etc. This mechanism needs to be carefully handled by the developer: for example, an application should stop requesting sensor data while paused to avoid wasting resources or to commit unsaved data before it is closed.

2 Objectives

Given the critical aspects of event concurrency and lifecycle handling, the aim of this thesis is to describe an event-based testing approach for An-

droid applications. The Android operating system was chosen because of its widespread and openness, but the main concepts are valid also for other mobile environments.

The focus of the first part of the thesis is on lifecycle changes events. In particular, the problem of lifecycle handling is addressed first with a static code analysis approach to recognize possible misuses of components, and then with a dynamic technique that allows the developer to drive the application lifecycle to test its robustness.

The second part of the thesis focuses instead on generic events, providing to the developer a tool to test their behavior in dynamic conditions. More in detail, a temporal logic language is defined to specify consistency checks on the stream of events, implemented in Android exploiting the ReactiveX library, an innovative system of industrial interest.

3 Achieved Results

The original contributions of this thesis are:

- A survey of the approaches to test Android applications, with particular regard to lifecycle handling and concurrency of events.
- A collection of examples of best practices in handling components according to the application lifecycle.
- Implementation of static checks to help the developer recognize possible misuses of components with regard to lifecycle.
- Implementation of a testing framework that allows to verify the behavior of an application during common lifecycle changes.
- Definition of a temporal logic language to describe consistency checks among events and its implementation in a testing tool.
- Empirical evaluation of the proposed tools on real-world applications, to discuss their effectiveness.

4 Outline of the Thesis

The thesis is organized as follows:

- Part II provides background information on testing and Android in general, as a reference to the main concepts used in the document.

- Part III analyzes the main technologies available to test Android applications and some examples of research related to the objectives of the thesis.
- Part IV focuses on lifecycle events and especially on the static and dynamic approaches to test them.
- Part V explains in detail the concept of event-based testing proposed in the thesis.
- Part VI summarizes the main concepts highlighting applicability, limitations and possible improvements.

Part II

Background

5 Introduction

This part presents more in depth the main concepts that are used during the thesis. In particular, it starts with a short description of testing in general, with some terminology that is employed throughout the thesis. It then provides some information on Android and its development concepts, with particular attention to component lifecycle, which is the focus of the first half of the thesis. Finally, it describes general event-driven systems, introducing some of the issues that are addressed by the second half of the thesis.

6 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 certain situations and to reach an acceptable confidence that the program works as expected under the most common conditions. The topic of assessing the quality of a product, as well as issues like when to start or end the evaluation process, has been widely analyzed by research papers and books, such as Software Testing and Analysis [2].

Test cases require a mechanism to 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 [3]. 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 enables the developer to specify the test conditions or it can even be manually evaluated by a human being.

This thesis mostly focuses on assertions as testing oracles. An assertion is a statement placed either at a specific point in a program or inside a test case

that enables to check a condition. It is expected to be verified, but if a bug is present the assertion fails and the system throws an error. Assertions are test oracles that specify what the application does rather than how.

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 at the user level but 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).

7 Android

7.1 Operating System

Android [4] is an operating system (OS) developed by Google. It is designed primarily for touchscreen mobile devices like smartphones and tablets, but it was recently extended to televisions (Android TV), cars (Android Auto) and smartwatches (Android Wear).

The OS works on top of a Linux kernel, but rather than running typical Linux applications it uses a virtual environment (Dalvik or, starting from Android 5.0, Android Runtime) to execute Android-specific apps. The system is characterized by the so called sandboxing mechanism: each process runs in its own virtual machine 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, i.e. only the parts of the system for which it has specific 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.

7.2 Application 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 an 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 from a 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 events, 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.

7.3 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 a number of other states.

When a lifecycle state transition happens, the Android system gives the opportunity to implement several callbacks in the component implementation to manage their behavior in those situations. In particular, the Java classes that define a component (such as `Activity`), which are subclassed by the developer, allow to override one or more methods to handle the callbacks (such as `onCreate()` or `onPause()`).

7.3.1 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.

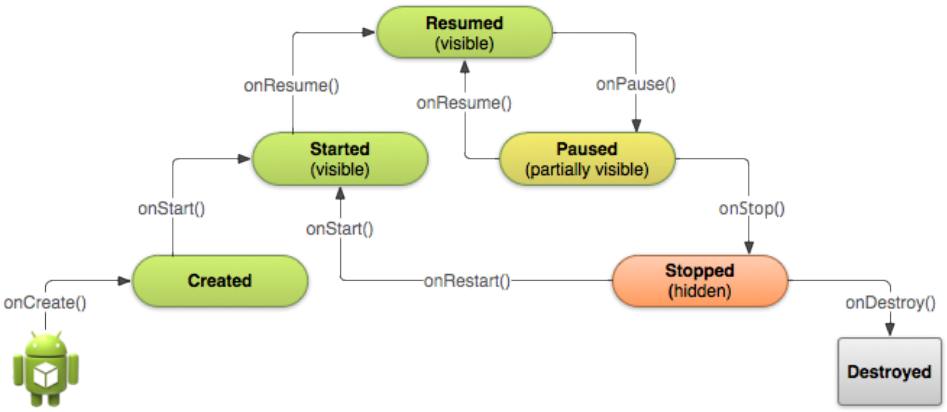


Figure 1: Simplified scheme of the Activity lifecycle. Source: `https://developer.android.com/training/basics/activity-lifecycle/starting.html#lifecycle-states`

- 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 when the user presses the back button or by the OS when it is stopped and the device needs to free resources. In the latter case, if the application is then reopened by the user, the operating system provides a way to restore the lost information: in particular, the developer can override the callbacks `onSaveInstanceState()` and `onCreate()` to pass data between the old and the new instance.

Figure 1 shows a simplified scheme of the Activity lifecycle, with the corresponding Java callbacks.

7.3.2 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: 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, 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. Figure 2 shows the main Fragment lifecycle callbacks.

Another characteristic of Fragments is that their instance can be retained, if the developer chooses to do so. This means that the Fragment instance is kept even if the Activity is recreated (e.g. the device screen is rotated), allowing to skip time-consuming initializations.

7.3.3 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 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 an 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.

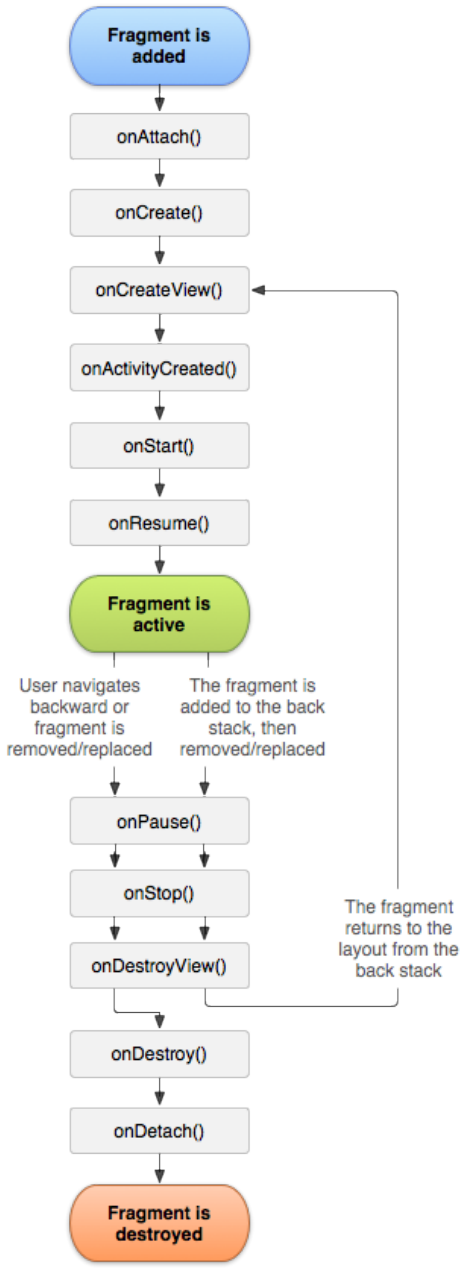


Figure 2: Main Fragment lifecycle callbacks. Source: `https://developer.android.com/guide/components/fragments.html#Creating`

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

- During `onCreate()`:
  - Set the layout resource defining the UI.
  - Initialize View components (e.g. add click listeners to buttons).
  - Initialize the component logic (e.g. class-scope variables).
  - Restore the previous state (if any) saved during `onSaveInstanceState()` [this can be alternatively performed during `onRestoreInstanceState()`].
- During `onRestart()`:
  - Requery raw Cursor objects previously deactivated during `onStop()`.
- During `onStart()`:
  - Acquire "secondary" resources (e.g. Broadcast Receiver).
  - Verify system features (e.g. GPS enabled), because they may change when the application is in background.
- During `onResume()`:
  - Start animations and similar CPU-intensive operations.
  - Acquire "critical" resources (e.g. camera, sensors).
  - Should *not* restart on-going operations that require user visibility (e.g. games, videos), but let the user decide when to do so.
- During `onPause()`:
  - Commit unsaved changes made by the user (e.g. save a draft for an unfinished email), if it does not require too much time.
  - Stop animations and other operations that may be consuming CPU.
  - Stop on-going operations that require user visibility (e.g. games, videos).
  - Release "critical" resources (e.g. camera, sensors).
  - Should *not* perform any long running operation, the time complexity of this method should be as low as possible.
- During `onStop()`:
  - Release "secondary" resources (e.g. Broadcast Receiver).

  - Perform CPU-intensive shutdown operations (e.g. write unsaved data to database).

- During `onDestroy()`:
  - Stop background threads created during `onCreate()`.
  - Release long-running resources that could create memory leaks.
  - Should *not* save data, because the method may not be called in all situations.

8 Event-Based Systems

8.1 Event-Driven Architecture

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 a 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 Architectures are:

- Extremely loosely coupled: the producer does not know anything 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.

8.2 Android as an Event-Driven Architecture

Android is implemented as an event-based system. This is because a mobile 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 in a FIFO event queue and processes one by one. However, events produced by different threads are processed concurrently and not guaranteed to be ordered

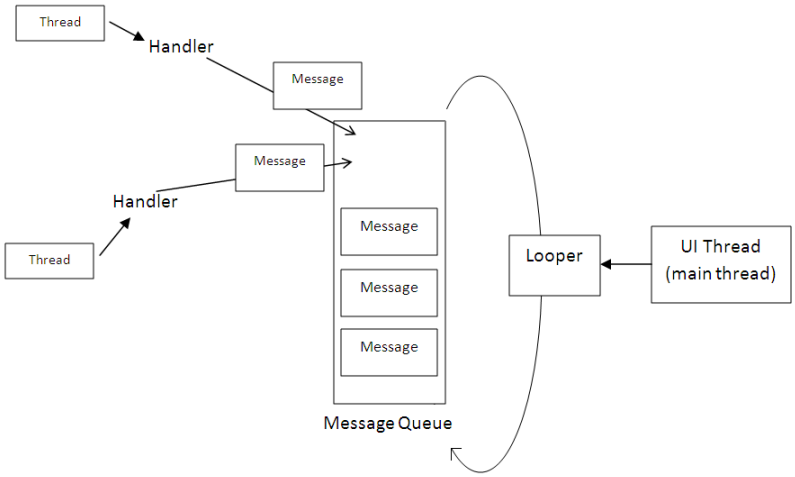


Figure 3: Simplified scheme of Android event handling. Source: `https://guides.codepath.com/android/Repeating-Periodic-Tasks`

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 are:

- User input events can be observed attaching listeners to View components. An event listener is a Java 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 Broadcast Receivers in the `onReceive()` callback.
- Asynchronous messages from services external to the application (e.g. sensor data, network requests, etc.) also have their specific callbacks.

8.3 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.

In this context, the most relevant challenge in an Android application is race conditions: two or more events do not happen in the order the programmer intended. As an example, reported in listing A.2, a race condition may happen when the developer starts an AsyncTask (a utility class that allows to perform short operations in the background without blocking the UI thread) to asynchronously perform some operation. If the user, for example, rotates the device while the AsyncTask is running, the application crashes with an `IllegalStateException` because the reference to the old View is lost.

Part III

Android Testing State of the Art

Android test cases adhere to the JUnit [5] format, a testing framework for Java. It gives the possibility to create classes called test cases that contain methods annotated with `@Test`, each representing a test.

In addition to the JUnit environment to run the tests, in Android testing JUnit-like assertions are usually employed. These assertions allow to specify a condition on one or more Java objects and throw an `AssertionError` exception if the check fails. For example, the method `assertEquals(String message, Object expected, Object actual)` checks that the two given objects are equal and, if not, throws an error with the given message.

9 Unit Testing

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 [6].
- Instrumented: it is executed on a physical device or on an emulator and so has access to the full instrumentation 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 [7] tries to take the advantages of the two previous 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 them to focus on the behavior of the application instead of the implementation of Android.

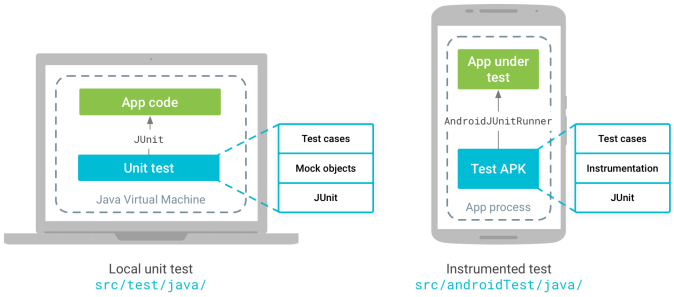


Figure 4: Local and instrumented Android tests. Source: `https://developer.android.com/training/testing/start/index.html#test-types`

10 UI Testing

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.

We may have:

- UI testing on a single app: the Espresso [8] library offers APIs for writing UI tests to simulate user interactions. In general, defining a test case means building a series of `onView(Matcher).perform(ViewAction).check(ViewAssertion)` instructions, i.e. select a View that matches a 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 [9] 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).

11 Runner Tools

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

- UI Exerciser Monkey [10]: gives the possibility to run an application on a physical device or emulator generating a repeatable pseudo-random stream 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 [11]: controls a device or emulator from a workstation by sending specific commands and events defined as a Python program. It also allows to take screenshots during the test execution and store them on the workstation.

12 Lifecycle Testing

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

12.1 Testing Frameworks Support

The built-in instrumented testing framework allows to drive an Activity lifecycle via the methods `callActivityOnStart()`, `callActivityOnPause()`, etc. provided by the `Instrumentation` class. These methods, however, are complex to use because the developer needs to chain the correct methods in each test case to simulate a valid lifecycle transition (e.g. for example to simulate the Activity being closed the developer needs to call in order pause, stop and destroy) and, since they are required to run on the main thread of the application, often need to be executed inside a `Runnable` that may require explicit synchronization with the testing thread.

The Robolectric framework makes things a little easier allowing the developer to specify a chain of transitions like `buildActivity(MyActivity.class).create().start().resume()`, avoiding `Runnable` and synchronization problems. This technique, however, suffers the same problem of the instrumented framework: it requires explicit calls to the correct chain of transitions to simulate realistic behaviors, and so coded by the developers only in very specific situations.

12.2 THOR

One example of tool designed with particular regard to lifecycle handling is THOR [12]. The idea of the tool is to run pre-existing UI test cases (defined in Robotium or Espresso) in adverse conditions to test the application 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 → Resume; Pause → Stop → Restart; Audio focus loss → 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 features like:

- Neutral events are injected in suitable locations to avoid conflicts with the test case: 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 effective and useful for bug detection, THOR is not a user-friendly tool. First of all, the tests are executed 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. In addition to this, the pre-defined test cases on which the tool works need to be well structured: if they do not reach a faulty portion of the application, the neutral events never allow the

bug detection.

13 Race Conditions Testing

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

13.1 CAFA

For example, CAFA [13] 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 determines 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 behaviors, the tool analyzes the traces of the low-level read and write operations, as well as certain branch instructions. While CAFA is a good starting point for race condition analysis, its main problems are that it's fairly slow (it may take hours to analyze an application), only focuses on a very specific type of race conditions and the tool is not publicly available.

13.2 EventRacer

Android EventRacer [14] 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 (i.e. avoid instantiating as many rows as the number of data values). If the content of 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 in listing A.2, i.e. the AsyncTask completing when the Activity has already been rotated.
- 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. They 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. Another critical aspect of the tool is that detection is performed by running the application in an emulator feeding it with pseudo-random events: this approach may skip several sections of an application, especially if they are not immediately available (e.g. they require user sign-in).

13.3 ERVA

The tool ERVA [15] is a further step forward in event race detection. The authors note that the previous tools have several drawbacks like:

- They are prone to false positives, due to:
  - Imprecise Android Component Model: for example, EventRacer may recognize the callbacks `onCreateView()` and `onResume()` as racy, but the Android lifecycle documentation states that they are always called in that order.
  - Implicit Happens-before Relation: two Runnable objects, if posted to the same Handler, are executed in a FIFO order, but EventRacer may wrongly assume that they can be concurrent.
- They cannot verify if it a benign or harmful race. A benign race can be generated by:
  - Control Flow Protection: if the access to a variable is protected by an `if` statement that first checks whether it is null, even if a race occurs there is no impact on the execution.

  - No State Difference: if the program does not depend on the execution order (e.g. a counter is decreased in the same way by several tasks) a race is meaningless.

- They do not give developers a way to reproduce the race.

ERVA addresses these issues by splitting the analysis in two phases:

- Race detection: the tool runs EventRacer itself to collect all the detected races, recording in the meanwhile replay and synchronization information. With the recorded data, ERVA determines if the race is a false positive.
- Race verification: the tool tries to verify if the detected true positives are benign or harmful races. To do so, it exploits the replay data to reproduce the execution by changing the order of some events: if the flipping has no side effect, then the race is benign.

13.4 DEvA

A completely different approach for race detection is provided by DEvA [16]. The idea of the tool is to base its search on static code analysis rather than dynamic analysis: this technique 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 performs 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.

14 Addressed Limitations

The work presented in this thesis addresses some of the limitations of the testing frameworks and tools just presented.

The first part of the thesis provides an easier way to test the application lifecycle, overcoming the restrictions imposed by the existing testing frameworks, like the need to chain the correct callbacks in every test case.

In the second part of the proposed work, a more effective way to test events in Android applications is described, which also approaches the issue of race conditions from a different point of view with respect to the aforementioned tools (testing instead of detection, avoiding the problem of false positives and false negatives).

Part IV
Lifecycle Testing

15 Introduction

Correctly handling a component lifecycle is a key aspect of Android application development, to avoid waste of resources, crashes and unexpected behaviors.

However, the available means to thoroughly test an Activity lifecycle are lacking. The Android testing frameworks supply low-level methods to drive the lifecycle but they need to be carefully chained to simulate realistic behaviors, and existing testing tools like THOR, while effective, present some limitations such as usability issues.

Given this context, the solution proposed by the first part of this thesis provides a more complete support to test a component lifecycle. In particular, two approaches for lifecycle testing are proposed: static analysis and dynamic analysis. The former analyzes the source code of the application to detect possible issues related to the handling of some components in accordance to the host Activity/Fragment lifecycle, for example failing to release a resource that was previously acquired. The dynamic approach, instead, provides to the developer a testing framework to easily drive the lifecycle, with pre-generated test cases for the most common transitions.

The following sections present first the static technique and then the dynamic testing framework.

16 Static Analysis

16.1 Introduction

This section presents static lifecycle checks in depth. First a short introduction of the main concepts of static program analysis is proposed, followed by a detailed explanation of its application in lifecycle testing.

16.2 Static Program Analysis

As opposed to dynamic analysis that requires to run an application, static analysis only inspects the source code. Static program analysis is mainly

employed to highlight possible coding errors (e.g. access a variable that is always null), to formally prove properties (e.g. pre- and post-conditions of a function) and to assess if the software follows a set of coding guidelines (e.g. variable naming conventions). There are several different analysis techniques, here we focus on those that employ the Abstract Syntax Tree and the Control Flow Graph of the problem.

An Abstract Syntax Tree (AST) is a tree representation of the code structure, where each node represents a program construct (e.g. a variable or an instruction). AST Traversal is the static analysis technique that explores the Abstract Syntax Tree to detect likely structural and syntactical problems, such as coding conventions conformity.

A Control Flow Graph (CFG) instead represents the sequence of operations executed along all paths that may be traversed during the execution of the program. Control Flow Analysis employs the CFG to detect possible issues related to the program flow, e.g. accessing a variable that may not have been initialized in all previous paths. Data Flow Analysis is a similar technique that gathers information about the possible set of values in several points of the CFG.

16.3 Static Lifecycle Checks

This part of the thesis focuses on lifecycle checks using a static analysis approach. The idea is to consider some components used in the applications that require special attention, either because of their own lifecycle or because they need to be carefully used in accordance to the host component lifecycle.

Failing to correctly handle an Activity/Fragment lifecycle may lead to unexpected behaviors or resource waste. An unexpected behavior can be for example the loss of user data: if the developer does not react correctly to an Activity being destroyed in a note-taking app the currently written text may be lost. An example of resource waste can be requesting GPS location updates at constant intervals in an application that uses maps and failing to cancel them when the application is paused/stopped: the system will continue querying the device sensors even if the user is not currently looking at the map.

Static program analysis can help detecting some of these problems. For example, if a method call to acquire a certain component is detected but no call to release it is found in the code, then a warning can be shown to the developer.

16.4 Target Components

In this section a more detailed explanation of the target components that are useful to be checked statically is reported.

Each component is analyzed in terms of:

- Release: should the component be always released after it is acquired? If so, the static analysis can check if the call to release is always performed.
- Best Practices: in which states of the host Activity/Fragment lifecycle is recommended/usual to acquire and release the component? If there are best practices, the analysis can check if the calls follow them.
- Double Instantiation: can acquiring the same component more than once cause unexpected behaviors or waste computational power? If so, the static analysis can check if the resource may be acquired multiple times during the execution.

These three aspects were chosen because of their relevance in Android applications and because they are especially suitable for static analysis, being usually characterized by regular and often strict usage patterns.

Among the many components that can be used in Android applications, some were chosen and listed below as examples of targets for static lifecycle analysis. These components were selected for their standardization with regard to initialization and destruction: in most situations, the pattern for acquiring and releasing the linked resource is regular, and static analysis can easily assess the conformity with best practices.

The selected components for static lifecycle analysis are:

- Broadcast Receiver: it allows to receive messages from the system or another application component.
  - Release: the official documentation states that a Broadcast Receiver, if registered with an Activity context (as it is usually the case), should ideally be unregistered before the Activity is done being destroyed, but if it not so the system will clean up the leaked registration anyway and only log an error. In case the receiver is registered using the Application context, however, it will be never unregistered by the system, so missing a call to the unregister method may lead to significant leaks[1]. Moreover, it is reported that the developer should not unregister during an

[1] `https://developer.android.com/reference/android/content/Context.html#getApplicationContext()`

Activity `onSaveInstanceState()` method, because it won't be called if the user moves back in the history stack[2].

  - Best Practices: the only official recommendation on how to handle a Broadcast Receiver in accordance to the lifecycle of the Activity/Fragment that uses it is to unregister it during `onPause()` if it is registered during `onResume()`[3]. The common usage of the receiver is to register during `onResume()` or `onStart()` and to unregister during `onPause()` or `onStop()` respectively.

  - Double Instantiation: registering twice a Broadcast Receiver with the same parameters does not create any correctness issue and the complexity of the method is negligible, so there's no need to check double instantiation for this component. However, if the Broadcast Receiver is unregistered twice the system throws an `IllegalArgumentException`, so it might be useful to warn the developer in advance if two calls are detected.

- Google API Client: the `GoogleApiClient` class allows to connect to the Google Play services for several APIs, like Google+, Google Drive, wearables, etc. The developer can either manage the connection manually or leave it to the system. The following specifications are of course valid for the former case.

  - Release: the API Client should always be disconnected when the application is done using it. This fact is not clearly stated in the documentation, but, given the best practices described in the next element, it is safe to assume that it is required.

  - Best Practices: the official recommendation is to connect during `onStart()` and to disconnect during `onStop()`[4].

  - Double Instantiation: it is not a problem since the call to `connect()` returns immediately if the client is already connected or connecting[5].

- Fused Location Provider API: the `FusedLocationProviderApi` class enables to query information about the current location. In particular, we are interested in the functionality that allows to receive periodic updates.

[2] `https://developer.android.com/reference/android/content/BroadcastReceiver.html`

[3] `https://developer.android.com/reference/android/content/BroadcastReceiver.html`

[4] `https://developers.google.com/android/guides/api-client#start_a_manually_managed_connection`

[5] `https://developers.google.com/android/reference/com/google/android/gms/common/api/GoogleApiClient#public-methods`

- Release: the location updates should be removed by calling the `removeLocationUpdates()` method[6].
- Best Practices: the official documentation encourages the developer to think whether it may be useful to stop the location updates when the Activity is no longer in focus, to reduce power consumption while the app is in background[7].
- Double Instantiation: does not need to be considered since any previous location updates are replaced by the call to `requestLocationUpdates()`[8].

- Camera:
  - `Camera` class: this is the older API to control the device camera, deprecated in API level 21 (Lollipop).
    - Release: releasing the camera is fundamental. Failing to do so means that all the applications on the device will be unable to use it[9].
    - Best Practices: the best practice is to acquire the camera during `onResume()` and release it during `onPause()`[10].
    - Double Instantiation: if the developer tries to acquire the camera twice, the system will throw a runtime exception[11].
  - `camera2` package: this is the new API introduced in Android Lollipop to manage the device camera.
    - Release: although the documentation does not clearly state that the developer *must* release the components, the objects used to manage the camera such as `ImageReader`[12], `CameraDevice`[13], `CameraCaptureSession`[14], etc. all provide

[6] `https://developers.google.com/android/reference/com/google/android/gms/location/FusedLocationProviderApi#removeLocationUpdates(com.google.android.gms.common.api.GoogleApiClient,%20android.app.PendingIntent)`

[7] `https://developer.android.com/training/location/receive-location-updates.html#stop-updates`

[8] `https://developers.google.com/android/reference/com/google/android/gms/location/FusedLocationProviderApi#public-methods`

[9] `https://developer.android.com/reference/android/hardware/Camera.html#release()`

[10] `https://developer.android.com/reference/android/hardware/Camera.html`

[11] `https://developer.android.com/reference/android/hardware/Camera.html#open(int)`

[12] `https://developer.android.com/reference/android/media/ImageReader.html#close()`

[13] `https://developer.android.com/reference/android/hardware/camera2/CameraDevice.html#close()`

[14] `https://developer.android.com/reference/android/hardware/camera2/`

a `close()` method to release the resource. In the sample application by Google[15] these methods are all called during `onPause()` and a `Semaphore` is used "to prevent the app from exiting before closing the camera", so it is safe to assume that releasing these resources is required.

    - Double Instantiation: methods to acquire the components like `setRepeatingRequest()`[16], `createCaptureSession()`[17], etc. replace the previous calls. For this reason, there's no need to check double instantiation for correctness, but since the time complexity of most of these calls is significant (e.g. `createCaptureSession()` "can take several hundred milliseconds") it can be checked for performance reasons.

- Ads View: the `AdView` class is a View used to display banner advertisements.
  - Best Practices: the class provides methods like `destroy()`, `pause()`, etc. that should be called in the Activity/Fragment lifecycle methods with similar name (`onDestroy()`, `onPause()`, etc.)[18].

Other components, such as Services or Threads, should also be carefully managed according to the lifecycle. They are not listed above because, since there are too many different use cases (e.g. a Service may be active during several Activities, work when the application is in background, etc.), it is difficult to extract a list of best practices and, as a consequence, to statically check the code for misuse.

CameraCaptureSession.html#close()

[15] https://github.com/googlesamples/android-Camera2Basic/blob/master/Application/src/main/java/com/example/android/camera2basic/Camera2BasicFragment.java

[16] https://developer.android.com/reference/android/hardware/camera2/CameraCaptureSession.html#setRepeatingRequest(android.hardware.camera2.CaptureRequest,android.hardware.camera2.CameraCaptureSession.CaptureCallback,android.os.Handler)

[17] https://developer.android.com/reference/android/hardware/camera2/CameraDevice.html#createCaptureSession(java.util.List%3Candroid.view.Surface%3E,%20android.hardware.camera2.CameraCaptureSession.StateCallback,%20android.os.Handler)

[18] https://developers.google.com/android/reference/com/google/android/gms/ads/AdView

16.5 Design

Given the chosen target components, the static analysis tool can help the developer detecting the outlined issues and best practices. In particular, the thesis focuses on two significant examples:

- Broadcast Receiver: to implement the static check, the tool only focuses on single Java classes. Even though it is possible to use the same receiver across multiple classes (e.g. passing the reference as a method parameter), it is not a common approach, and so the running complexity may be decreased without loss of much detection power.
  - Release: the static tool detects registrations and unregistrations in the class and, if a pair is not found, shows a warning. The detection is performed considering the name of the variable of the considered Broadcast Receiver: even if it is possible to have two different variables referencing the same receiver, it is not common and so we can avoid performing a complex Control Flow Analysis, and employ an AST Traversal technique instead. This check is performed for all registrations, both with Activity and Application Context, since even in the former case it is better to unregister to avoid possible overheads in the automatic system clean-up.
  - Best Practices: the tool checks if the call to unregister is performed inside the `onSaveInstanceState()` method of a Fragment or an Activity and, if so, shows a warning to the developer. This is performed by AST Traversal to check in which class and method the unregistration is performed.
  - Double Instantiation: the tool detects multiple unregistrations of the same receiver inside the class and, in that case, shows a warning. Only the unregistrations outside a try/-catch block are considered, since if the developer catches the `IllegalStateException` that may be generated it is possible to have multiple unregistrations.
- Google API Client: the analysis is similar to the Broadcast Receiver case. The tool focuses on single Java classes, since it's unusual to connect in one class and disconnect in another.
  - Release: if a call to the connection method is detected but a disconnection is not, the tool shows a warning. Detection is performed via AST Traversal because, given the strict best practice to use `onStart()` and `onStop()`, it is not worth to use a Control Flow Analysis to actually see if the disconnection is called in

every path following the connection.

- Best Practices: if via AST Traversal the tool detects that the current class is an Activity or a Fragment, it shows a warning if connection and disconnection are not performed in `onStart()` and `onStop()` respectively.

16.6 Implementation

16.6.1 Introduction

For the implementation of the static checks [17] Android Lint was chosen, of which a more detailed description is reported in the next section. After this, the actual implementation structure is explained.

16.6.2 Android Lint

Android Lint [18] 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, but 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 employed 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 Guidelines.
- 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 and not hard-coded in Java, making translations feasible.

16.6.3 Custom Android Lint Checks

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. `ClassDeclatation`, `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 developer.

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 `afterCheckFile()` or `afterProject()` hooks, which are called when the current file or the whole project respectively have been analyzed. In order to decide if a problem is present or not, the developer needs to save 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 problem, the developer 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).

The Android Lint tool also provides the means to perform Control Flow Analysis during the detection process. In particular, the `ControlFlowGraph` class allows to build a low-level Control Flow Graph 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. They 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 subclasses 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 enable 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

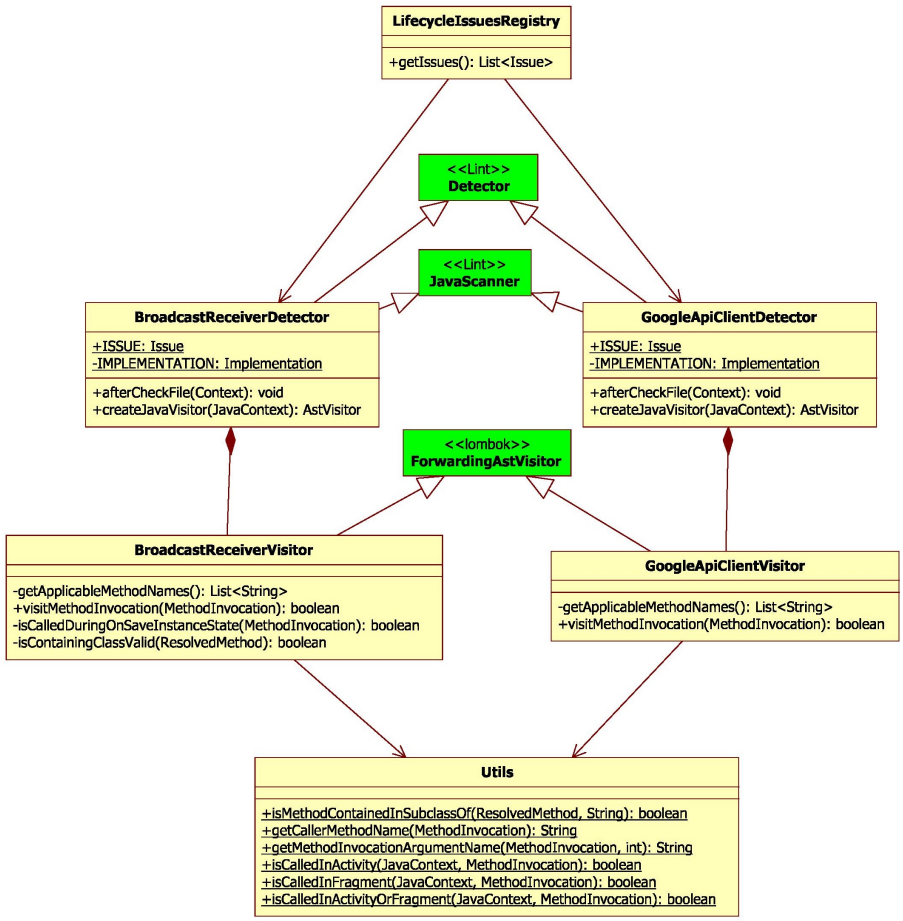


Figure 5: UML Diagram for the Lifecycle Static Checks Implementation

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.

16.6.4 Code Structure

The Detectors `BroadcastReceiverDetector` and `GoogleApiClientDetector` are implementations of `JavaScanner` and they both define a `ForwardingAstVisitor` subclass to implement their AST Traversal behavior.

In particular, `BroadcastReceiverVisitor` employs the `visitMethodInvocation(MethodInvocation)` callback to detect the calls to registration and unregistration. First, it checks that the current

`MethodInvocation` is from the correct class (either `Context` for global receivers or `LocalBroadcastManager` for local receivers) and, if so, it stores the variable name in a list field variable. If it's the unregistration method, it also checks if the caller method is `onSaveInstanceState()` and in case shows a warning to the user. In the `afterCheckFile()` callback, `BroadcastReceiverDetector` checks the stored values consistency, showing an Issue if an unregistration is missing or if the same receiver is unregistered multiple times (without any try/catch block). In that case, it computes the code location handle to show a warning to the user in the correct place.

`GoogleApiClientVisitor` also receives the `visitMethodInvocation(MethodInvocation)` callback and checks if the current invocation is either a connection or a disconnection, setting a boolean flag for each. If the current class is an Activity or a Fragment, it checks that the caller method is `onStart()` or `onReceive()` respectively. During `afterCheckFile()`, the `GoogleApiClientDetector` shows a warning if the connection flag is set but the disconnection one is not, computing the correct location handle.

Finally, the `LifecycleIssuesRegistry` simply sends the `Issue` objects (defined as static fields in the Detector classes) to the Lint tool.

16.7 Evaluation

16.7.1 Real-World Case Study

To show the lifecycle static checks effectiveness, some real-world examples were selected among the open-source applications on the GitHub platform:

- InTheClear [19] is an application for alerting during emergencies. Before commit *cabfc26*[19] a Broadcast Receiver was registered but never unregistered in the `SMSSender` class. Using the static lifecycle tests the developers would have detected the problem immediately, without the need to receive the log notification and open the issue #56[20].
- TrackBuddy [20] is an SMS tracker application. Commit *007b6f7*[21] solves a bug in `LocationService`, which was missing a Google API Client disconnection, a problem that would have been detected by lifecycle checks.

[19] `https://github.com/SaferMobile/InTheClear/commit/cabfc2643bc5820e109433a8ae48f9f6a0788d9f`

[20] `https://github.com/SaferMobile/InTheClear/issues/56`

[21] `https://github.com/byteShaft/TrackBuddy/commit/007b6f7acb39e90d53ef38577d212927fae19d91`

With regard to the analysis performance, it can be said that the Lint tool is usually able to analyze an entire project in less than one minute, checking for all the registered issues. As an example, the project-wide analysis of InTheClear (40 KLOC) took an average of 20 seconds to complete, while for TrackBuddy (60 KLOC) 40 seconds were necessary. The code of the developed lifecycle checks does not present any criticality from the performance point of view, since it employs only a AST Traversal technique on a single Java class, causing negligible time complexity degradation (the average duration of the analysis was the same both with and without the custom Detectors).

16.7.2 Discussion

In the listed examples, the checker correctly recognizes the aforementioned issues as real problems, but it must be noted that static analysis in general can produce false negatives or false positives, reducing its effectiveness. This means that there may be situations where the developer is notified of a non-relevant problem or a real issue is not detected: for example, if the developer decides to employ the checked components in a non-standard way (e.g. register a Broadcast Receiver in an Activity and unregister it in another, passing the reference between them, although such practice is discouraged) the tool would wrongly recognize it as a problem.

From the point of view of applicability, it can be said that the static lifecycle tests target only a specific subset of Android applications, i.e. those that employ the components linked to each Detector. Moreover, they are useful mostly in the early stages of development to recognize inconsistencies when the developer is coding an Activity or Fragment, since the issues would likely be detected at the first run or test case via crashes or logged errors.

Concerning usability, it can be noted that the static checks are completely integrated in the Lint tool available in the development IDE, so each of them behaves exactly as the built-in Detectors. For this reason, the issues are both displayed as a visual clue on the code (the detected problem is highlighted and a warning message is shown) and included in the project-wide analysis run on demand by the developer.

17 Dynamic Analysis

17.1 Lifecycle Test Cases

In this section a dynamic approach to lifecycle testing is presented, as opposed to the static technique described in the previous section.

The idea is to provide to the developer pre-generated test cases that allow to explore the most common lifecycle changes, to overcome the limitation of the built-in testing frameworks that only offer low-level methods that need to be carefully chained each time. With the proposed approach, the complexity of managing the lifecycle transitions is completely hidden to the developer, which can only focus on expressing actions to be performed and conditions to be checked during the test execution.

17.2 Defined Tests

The pre-generated lifecycle test cases offered by the library are:

- Pause: simulates the component being partially hidden, i.e. paused and then resumed. It is useful for example to see if the component correctly frees/stops and reacquires/starts "critical" resources and CPU-intensive operations.
  - `onCreate()`
  - `onStart()`
  - `onResume()`
  - *Actions and assertions before the pause*
  - `onPause()`
  - *Assertions while paused*
  - `onResume()`
  - *Actions and assertions after the pause*
- Stop: simulates the component being completely hidden (in background), i.e. stopped and restarted. The developer can for example check if all resources used by the application are correctly released while the component is in background, to avoid wasting computational power.
  - `onCreate()`
  - `onStart()`
  - `onResume()`
  - *Actions and assertions before the stop*
  - `onPause()`
  - `onStop()`

  - *Assertions while stopped*
  - `onRestart()`
  - `onStart()`
  - `onResume()`
  - *Actions and assertions after the stop*

- Destruction: simulates the component being closed (e.g. back button pressed or the application is killed by the system). In this case two different tests are defined, one where `onDestroy()` is called and another one where it is not, since in a real application this call is not guaranteed[22]. This test case can be used for example to see if the component stops all computations and, if needed, stores unsaved data in memory.
  - `onCreate()`
  - `onStart()`
  - `onResume()`
  - *Actions and assertions before the destruction*
  - `onPause()`
  - `onStop()`
  - [`onDestroy()`]
  - *Assertions after the destruction*

- Recreation: simulates the component being recreated. This can happen for example when a device configuration change happens (e.g. device is rotated) or an application in background is killed by Android to free resources and then restarted. This is usually the most critical lifecycle transition: since a completely new instance of the component is created (with a different reference), all variables that are not passed in the saved instance state `Bundle` are lost, other components (e.g. `AsyncTask`) may not be able to commit their results to the new component, etc.
  - `onCreate()`
  - `onStart()`
  - `onResume()`

[22] `https://developer.android.com/reference/android/app/Activity.html#onDestroy()`

  - *Actions and assertions before the recreation*
  - `onPause()`
  - `onStop()`
  - `onDestroy()`
  - `onCreate()`
  - `onStart()`
  - `onResume()`
  - *Actions and assertions after the recreation*

  The test case also provides a way to check the contents of the saved instance state `Bundle` to see if it contains the correct data passed from the old to the new instance of the component.

- Rotation: simulates the device being rotated, from portrait to landscape modes or vice versa. By default, from the lifecycle point of view this has the same effect of a recreation, but the developer can specify a different behavior changing the `configChanges` attribute in the application manifest. This test case can be useful to see if the component dynamically adapts to the new rotation, e.g. providing a different UI layout.
  - `onCreate()`
  - `onStart()`
  - `onResume()`
  - *Actions and assertions before the rotation*
  - [Possible lifecycle changes, by default recreation]
  - *Actions and assertions after the rotation*

The developer can also chain several of the listed tests together to build a complex execution flow, for example first pausing and resuming an Activity, and then destroying it.

17.3 Design

To allow custom actions and checks during the pre-defined lifecycle tests the system uses several callbacks, for example the *PauseCallback* gives the opportunity to specify some actions and checks before the component is paused, some checks while it is paused and some other actions and checks when the component is resumed again. These callbacks are defined by the

developer inside a test case and then used by the system during the lifecycle tests in the appropriate moments.

The pre-generated lifecycle tests are defined for two frameworks:

- Activity Test Rule tests: the `ActivityTestRule` objects allow the developer to specify which Activity to consider during a test. It is used for:
  - Instrumented unit testing
  - Espresso UI testing
- Robolectric tests: "hybrid" unit testing

Which cover the most used modern testing mechanisms in Android.

17.4 Implementation

The dynamic lifecycle tests library [21] is split in three modules. The reason for this is the Android system of including test libraries: using Gradle, the developer can *testCompile* the libraries used for local tests (i.e. that run on the development computer) and *androidTestCompile* the libraries used for instrumented tests (i.e. that run on real mobile devices or emulators). Since Robolectric is local and Activity Test Rule is instrumented, two distinct modules `RobolectricLifecycleTesting` and `ActivityTestRuleLifecycleTesting` allow the developer to correctly include them without creating useless dependencies. The `LifecycleTesting` module contains the common parts of the other two.

The `LifecycleTest` class inside the `LifecycleTesting` module is the main class of the system. It defines:

- Abstract methods implemented by the sub-modules that allow to specify the means to control the component lifecycle, according to their structure.
- Abstract methods implemented by the end-developer that enable to pass the callbacks for each lifecycle test.
- Helper methods that define the correct sequence of lifecycle callbacks for each of the tests, which can also be used by the end-developer to chain a number of test cases together.
- The actual pre-generated tests that use the previous three sets of methods to drive the lifecycle and to perform actions and checks in between.

Inside the module, several interfaces for the callbacks are also defined.

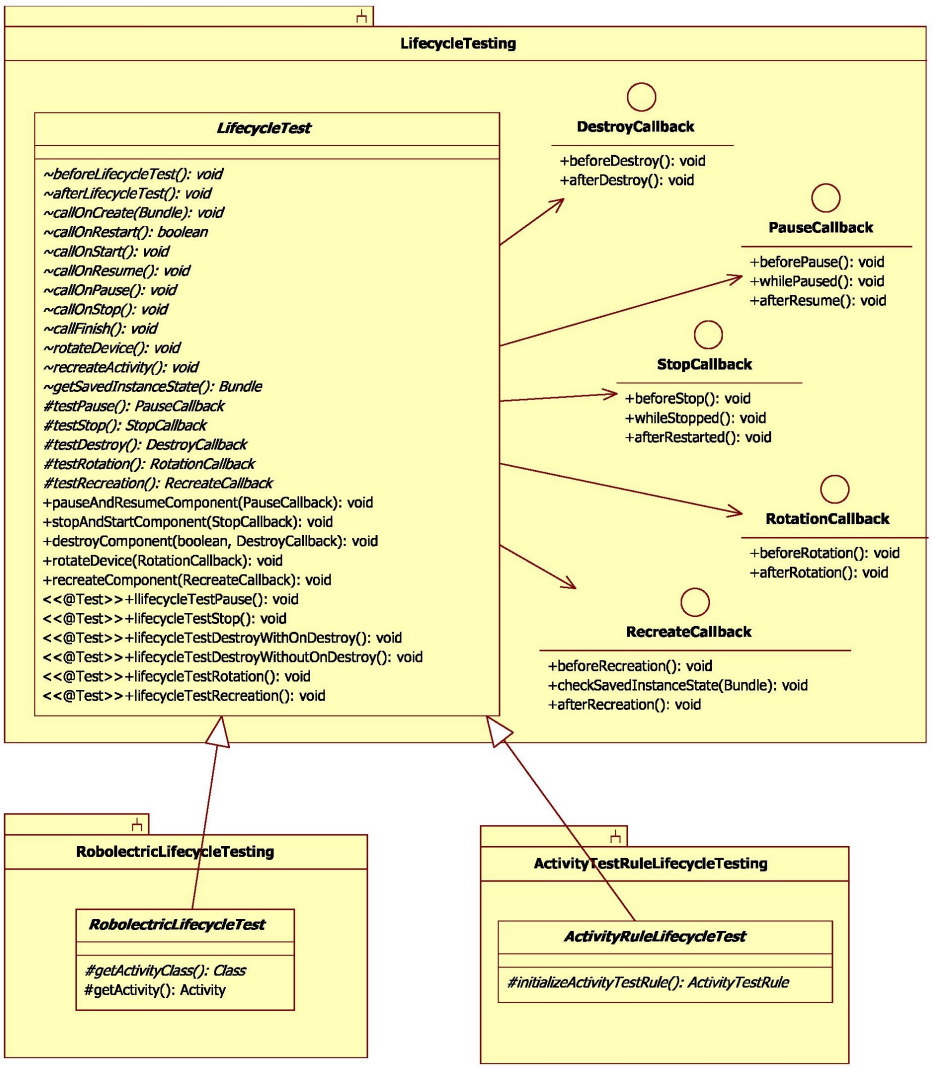


Figure 6: UML Diagram for the Lifecycle Dynamic Tests Implementation

The `RobolectricLifecycleTest` class inside the `RobolectricLifecycleTesting` module is the implementation of the lifecycle tests for the Robolectric framework. Besides implementing the methods to drive the component lifecycle using the Robolectric `ActivityController` class, it also provides the abstract method `getActivityClass()` to allow the developer to define the Activity under test and the utility `getActivity()` that gives the possibility to retrieve the Activity during a callback.

Activity Test Rule tests are covered by the `ActivityRuleLifecycleTest` class inside the `ActivityTestRuleLifecycleTesting` module. It implements the lifecycle methods using the Android `Instrumentation` class and provides the abstract method `initializeActivityTestRule()` to allow the developer to define the Activity Test Rule under test.

17.5 Evaluation

17.5.1 Real-World Case Study

To prove the lifecycle tests effectiveness, the library has been applied to WordPress [22], the Android application that allows to manage directly from mobile devices the websites created with the well-known content management system WordPress [23].

The evaluation has been performed by applying the same lifecycle tests first on an old version of the application (at commit *9f9cb12*[23] on December 6, 2015) [24] and then on the latest version at the time of writing (at commit *bfdf611*[24] on July 15, 2016) [25]. In the six months elapsed between the two versions some lifecycle issues were discovered and fixed, and so the idea is to show that the defined tests fail in the older version while they succeed in the newer one.

The defined tests are:

- Loss of data in profile
  - Bug Category: usability, correctness
  - Lifecycle Transition: rotation
  - Linked Issue: #3608[25]

[23] `https://github.com/wordpress-mobile/WordPress-Android/commit/9f9cb12bb1d8a82d60897b5c3ff954c2f88de4f2`

[24] `https://github.com/wordpress-mobile/WordPress-Android/commit/bfdf611c6a37ddd1fad1980f73a00389b2e22677`

[25] `https://github.com/wordpress-mobile/WordPress-Android/issues/3608`

  - Fixed In: pull request #3912[26]
  - Problem Description: in the profile page, if the user typed a value (e.g. his/her name) and then rotated the device, the inserted data would be lost.
  - Test: the lifecycle test, defined in Espresso, exploits the rotation callback by first typing the name value and then, after the transition, asserting the input contents. The source code of the test can be found in listing A.1.
  - Causes and Solutions: the bug was caused by not saving the instance of the Views at the device rotation. The chosen solution was to employ a `DialogFragment` component that automatically manages button and dialog, including their lifecycle.

- Blank post content
  - Bug Category: correctness
  - Lifecycle Transition: pause/stop
  - Linked Issue: #3575[27]
  - Fixed In: pull request #3577[28]
  - Problem Description: reading a long blog post, pausing and resuming the application caused most of the content to be invisible to the user.
  - Test: the Espresso lifecycle test opens a blog post, scrolls down and, after the pause, checks that all the content is displayed.
  - Causes and Solutions: the `WebView` component used to display the page was paused during `onPause()` but not resumed during `onResume()`, and so the simple solution of the bug was to add the latter call.

- Unexpected page change
  - Bug Category: usability
  - Lifecycle Transition: rotation
  - Linked Issue: #3948[29]
  - Fixed In: pull request #4127[30]

[26] `https://github.com/wordpress-mobile/WordPress-Android/pull/3912`
[27] `https://github.com/wordpress-mobile/WordPress-Android/issues/3575`
[28] `https://github.com/wordpress-mobile/WordPress-Android/pull/3577`
[29] `https://github.com/wordpress-mobile/WordPress-Android/issues/3948`
[30] `https://github.com/wordpress-mobile/WordPress-Android/pull/4127`

- Problem Description: in the settings page, if the user accessed the notifications-specific area and then rotated the device, he/she was redirected back to the general settings screen.
- Test: the Espresso lifecycle test first opens the settings page and clicks on the notifications section button, then after the rotation checks if the same page is still displayed.
- Causes and Solutions: the bug was caused by the missing `NotificationsSettingsActivity` rotation handling. To solve the issue, the developers decided to simply add `android:configChanges="orientation|screenSize"` in the application manifest, to override the automatic recreation of the Activity after the rotation. This solution was chosen for consistency with the other Activities of the application, but it is not ideal since the Android developers themselves suggest it only as a last resort[31].

The three bugs are all correctly reproducible with the dynamic lifecycle testing library, i.e. the tests fail in the old version of WordPress and complete successfully on the latest version.

From the performance point of view it may be noted that the overhead introduced by the lifecycle transitions handling is negligible. For example, running the aforementioned `MyProfileActivityTest` as a standard Espresso test without any lifecycle transition took in average 1.2 seconds, while with the dynamic lifecycle testing library 1.4 seconds. The performance degradation is for the most part due to the overhead introduced by the Android lifecycle management, which is unavoidable if the developer wants to simulate real-world behaviors in the application.

17.5.2 Discussion

Lifecycle tests have a wide applicability range, since the behavior of almost every Activity of an application is directly affected by its lifecycle. Moreover, the tests are available for Activity Test Rule unit/integration testing, Espresso UI testing and Robolectric unit testing, and so they can be applied in most of the testing environments employed in Android development.

Concerning usability, to define the lifecycle test cases the developer only needs to inherit from the classes provided by the library and implement the abstract methods. In each of them, the complexity of handling the

[31] `https://developer.android.com/guide/topics/manifest/activity-element.html`

lifecycle transitions is completely hidden to the developer thanks to the callback mechanism, allowing to focus only on the actions and assertions to be performed during the test.

Part V

Event-Based Testing

18 Introduction

Moving from the specific type of events related to the lifecycle to a more general concept of event, we can also state that the built-in testing support is lacking from this point of view. In general, the only way to test concurrency of events with the provided frameworks is to manually set variables and check their state via conditional statements, which hardly helps in complex environments.

Given this lack of testing capabilities, several approaches related to the issue of race conditions were devised. These tools, like dynamic analysis with EventRacer and static analysis with DEvA, described in detail in section 13, do not focus on manual testing but on automatic detection of concurrency problems among events, mainly related to data races.

The solution proposed in the second part of this thesis aims at being a complementary tool to these techniques. Instead of focusing on automatic detection, it gives the means to manually specify event-related conditions during standard unit, integration or UI tests. This means that this approach presents the advantage of avoiding the issue of false positives and false negatives, intrinsic for detection tools, but the developer has to code test cases by hand to cover the relevant parts of the application.

More in depth, with the proposed event-based testing library the developer is able to specify which events to observe during the execution of the application and then to define consistency checks on their stream, by means of a temporal assertions language.

The language allows to express conditions on the event stream such as ordering among similar events, quantification of events of a certain type during an execution, causality among two or more events and so on. As a consequence, it also provides the means to take into consideration race conditions like the other tools, for example by specifying that a certain event can happen only after another one.

The implementation of this event-based testing approach exploits the ReactiveX library, an innovative system of industrial interest (it is employed for example by Microsoft and Netflix) that allows to observe, transform and listen to events.

The following sections first present a formalization of the temporal assertions

language used to define the consistency checks on the event stream, followed by a detailed explanation of the design, implementation and evaluation of the system.

19 Temporal Assertions Language

19.1 Consistency Checks

An approach to event-based testing is to define consistency checks, i.e. conditions that should be verified in the event stream. For example, we can define the happens-before relationship between two or more events, specify the exact number of events of a certain type that can appear during an execution and so on.

This part of the thesis explains how we can express these consistency checks among events by means of a temporal assertions language. The idea is to specify assertions similar to the standard JUnit-like way, i.e. statements that either succeed or fail and that throw an `AssertionError` in the latter case, but with a validity that is checked during the whole execution instead of the single instant in which the checks are invoked. Note that the language is targeted to express checks concisely and in human-readable format, similarly to most of the JUnit-like assertions. A detailed explanation of the language to express checks is reported in the next sections.

First, let's formally define some terminology:

- Event Stream E: the list of all the events registered during an execution, i.e. from the tool start at time *t1* until the tool stop at time *t2*.
- Event $e \in E$: an event contained in the stream. The event stream is totally ordered: each event has a generation time and given any two events $e1$ and $e2$ we have either $e1 \prec e2$ or $e2 \prec e1$.
- Consistency Check c: a rule among one or more events that should be verified in the stream. Once specified, the check is applied to the stream and returns either a *Success* or *Failure* outcome.
- Matcher m: a specification that describes one or more events in the stream. It is used by the consistency checks to match the events, in order to express a condition on them. For example, considering the events to be names, if $E = \{John, Bill, Mary, Jack, Elizabeth\}$, the matcher $starts\_with\_J$ would match (i.e. return true) the events $John$ and $Jack$.
- $n, i \in \mathbb{N}$

In the proposed formalization, each check is described by:

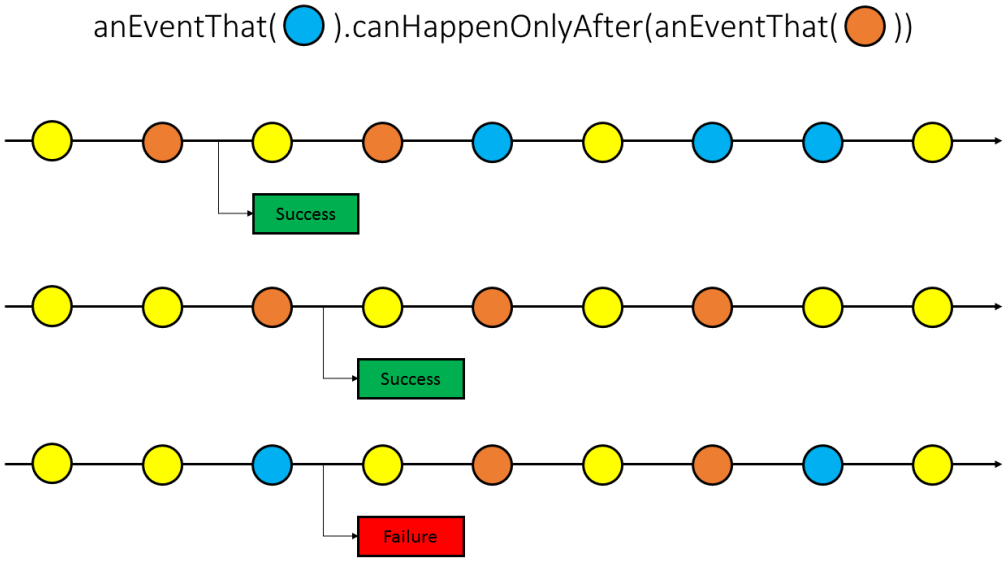
- Code structure: abstract description of the check specification. It has the same structure of the final code but use the aforementioned terminology to describe which components are expected.
- Description: a brief explanation of the check logic.
- First Order Logic (FOL) formalization of the check meaning, expressed using the following logic relationships:
  - $match(e,\ m)$ means that the matcher m matches the event e.
  - $before(e1,\ e2)$ means that the event *e1* is generated in the sequence before *e2*, i.e. $e1 \prec e2$.
  - $between(e2,\ e1,\ e3) \equiv before(e2,\ e1) \wedge before(e1,\ e3)$ means that the event *e1* is generated in the sequence after *e2* and before *e3*, i.e. $e2 \prec e1 \prec e3$.
- Visual representation: shows some examples of event streams and the corresponding check outcome. The outcome is placed in the spot where it is actually produced by the system: some checks may not need to explore the whole stream, i.e. if the condition is fulfilled or violated after an intermediate event the check "short-circuits".
- Real code example: an example of check specification as implemented by the system. The structure of the Java classes and objects will be clear later when the implementation is explained in depth.

19.2 Checks on Single Events

The first type of checks focuses on single events at a time: for example, we may say that an event of a certain type can be detected in the stream only after another event of a certain type.

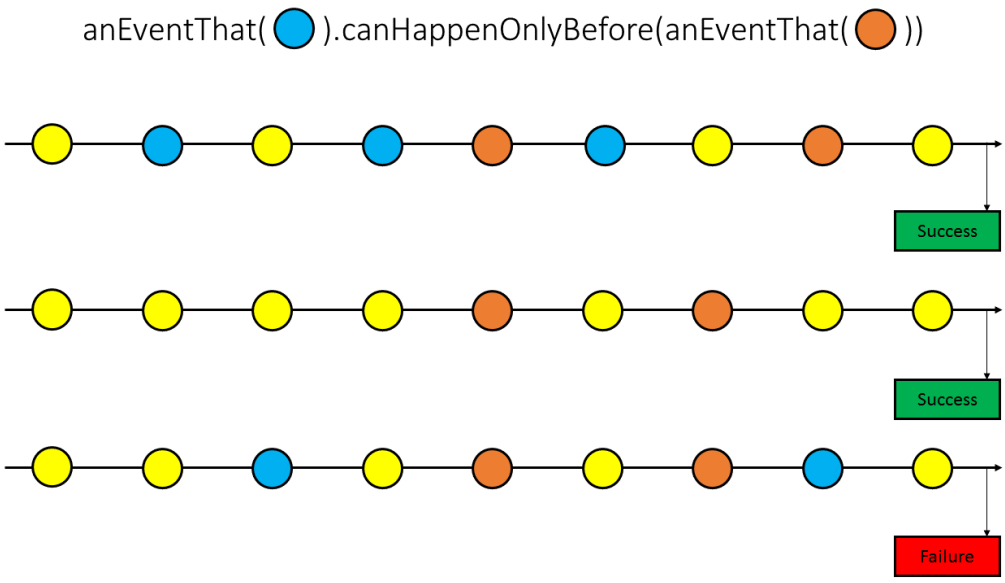
19.2.1 Can Happen Only After

Structure	`anEventThat(m1).canHappenOnlyAfter(anEventThat(m2))`

Description	Checks that the events that match `m1` happen only after any event that matches `m2`, i.e. there cannot be an event that matches `m1` before the first event that matches `m2`.
FOL	$\forall e1\Big(match(e1,\ m1)\ \Rightarrow\ \exists e2\Big(match(e2,\ m2)\ \wedge\ before(e2,\ e1)\Big)\Big)$
Visual	 Note in particular that in the second case the check succeeds even if no blue event occurs: we are just saying that blue events *can* happen after an orange one.

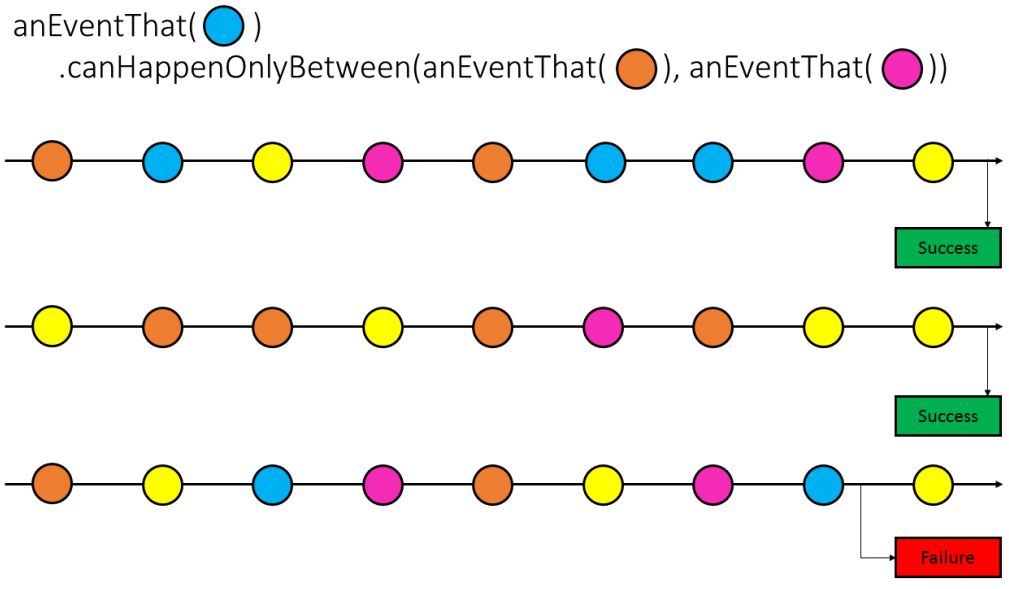
Code Example	`/* Check race condition with maps */`
	`anEventThat(isMarkerPlacement())`
	`  .canHappenOnlyAfter(anEventThat(isMapReady()));`

19.2.2 Can Happen Only Before

Structure	`anEventThat(m1).canHappenOnlyBefore(anEventThat(m2))`
Description	Checks that the events that match `m1` happen only before any event that matches `m2`, i.e. there cannot be an event that matches `m1` after the last event that matches `m2`.
FOL	$\forall e1\Big(match(e1,\ m1)\ \Rightarrow\ \exists e2\Big(match(e2,\ m2)\ \wedge\ before(e1,\ e2)\Big)\Big)$
Visual	

Code Example	`/* Can change form input only before it is submitted */`
	`anEventThat(isTextChangeFrom(usernameTextView))`
	`  .canHappenOnlyBefore(anEventThat(isSubmitForm()));`

19.2.3 Can Happen Only Between

Structure	`anEventThat(m1).canHappenOnlyBetween(anEventThat(m2), anEventThat(m3))`
Description	Checks that the events that match `m1` happen only between an event that matches `m2` and an event that matches `m3`, i.e. there cannot be an event that matches `m1` outside a pair `m2`-`m3`.
FOL	$\forall e1\Big(match(e1,\ m1)\ \Rightarrow\ \exists e2,\ e3\Big(match(e2,\ m2)\ \wedge\ match(e3,\ m3)\ \wedge\ between(e2,\ e1,\ e3)\ \wedge\ \neg\exists e3'\Big(match(e3',\ m3)\ \wedge\ \big(between(e2,\ e3',\ e1)\ \vee\ between(e1,\ e3',\ e3)\big)\Big)\Big)\Big)$
Visual	

Code Example

`/* Can send broadcast only if the Service is working */`
`anEventThat(isSendBroadcast())`
`  .canHappenOnlyBetween(`
`  anEventThat(isServiceStart(mainService)),`
`  anEventThat(isServiceStop(mainService)));`

19.3 Checks on Sets of Events

These checks work on sets of events: for example, we may say that a certain event generates a set of n other events. The cardinality of each set is specified by the quantifiers:

- Exactly
- At most
- At least

Only the formulas for "exactly" are written since the others can be derived by analogy.

First Order Logic specifications use the counting quantifiers [26] as a notational shorthand, i.e. $\exists_{=n}x$ means that there exist exactly n elements x.

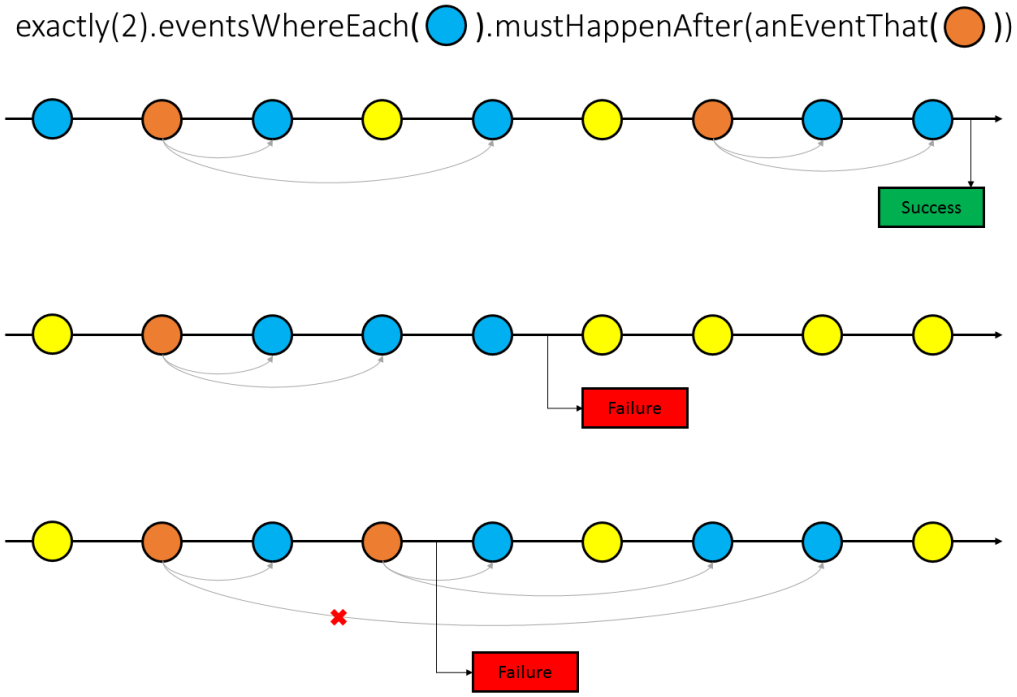
19.3.1 Must Happen After

Structure
`exactly(n).eventsWhereEach(m1).mustHappenAfter(anEventThat(m2))`
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

$$\forall e2\Big(match(e2,\ m2)\ \Rightarrow\ \exists_{=n} e1\Big(match(e1,\ m1)\ \wedge\ before(e2,\ e1)\ \wedge\ \neg\exists e2'\Big(match(e2',\ m2)\ \wedge\ between(e2,\ e2',\ e1)\Big)\Big)\Big)$$

Visual



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 *not* match one of the three blue events that follow the second orange event).

Code Example

```
/* If a location change happens, the TextView must be updated exactly once */
exactly(1).eventsWhereEach(isTextChangeFrom(locationTextView))
   .mustHappenAfter(anEventThat(isLocationChange()));
```

19.3.2 Must Happen Before

Structure

```
exactly(n).eventsWhereEach(m1).mustHappenBefore(anEventThat(m2))
```

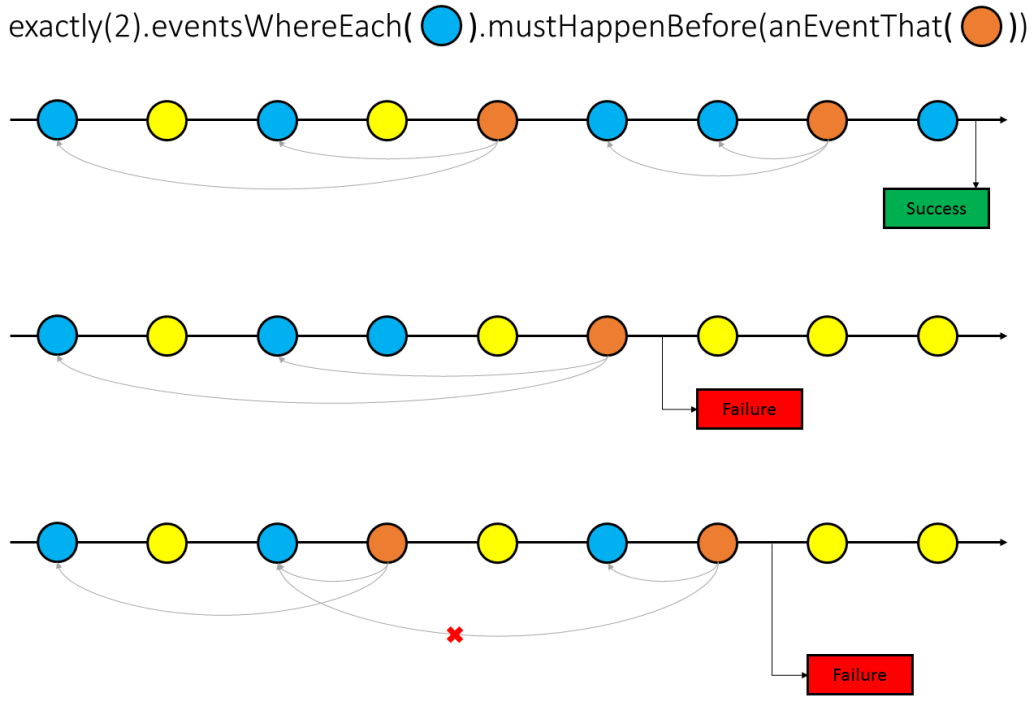
Description

Checks that exactly `n` events that match `m1` happen exclusively before every event that matches `m2`. "Exclusively" means that the sequence of `n` events must be after the previous (if any) event that matches `m2`.

FOL

$$\forall e2\Big(match(e2,\ m2)\ \Rightarrow\ \exists_{=n} e1\Big(match(e1,\ m1)\ \wedge\ before(e1,\ e2)\ \wedge\ \neg\exists e2'\Big(match(e2',\ m2)\ \wedge\ between(e1,\ e2',\ e2)\Big)\Big)\Big)$$

Visual



Code Example	

```
/* To setup the list of data the system must perform exactly 3 database
    queries */
exactly(3).eventsWhereEach(isDatabaseQuery())
   .mustHappenBefore(anEventThat(isListSetup()));
```

19.3.3 Must Happen Between

Structure	

```
exactly(n).eventsWhereEach(m1).mustHappenBetween(AnEventThat(m2),
    AnEventThat(m3))
```

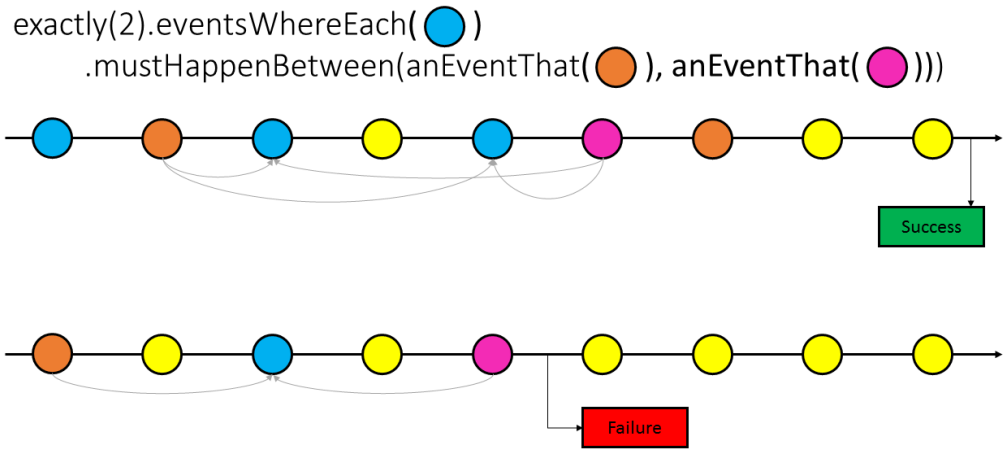
Description	

Checks that exactly `n` events that match `m1` happen between every pair of events that match `m2` and `m3` respectively.

FOL	

$$\forall e2\Big(match(e2,\ m2)\ \Rightarrow\ \exists e3\Big(\big(match(e3,\ m3)\ \wedge\ before(e2,\ e3)\ \wedge\ \neg\exists e2',\ e3'\big(match(e2',\ m2)\ \wedge\ between(e2,\ e2',\ e3)\ \vee\ match(e3',\ m3)\ \wedge\ between(e2,\ e3',\ e3)\big)\big)\ \iff\ \exists_{=n} e1\Big(match(e1,\ m1)\ \wedge\ between(e2,\ e1,\ e3)\Big)\Big)\Big)$$

Visual



Code Example

```
/* During the execution of the Activity the text is changed exactly four
    times */
exactly(4).eventsWhereEach(isTextChangeFrom(myTextView))
   .mustHappenBetween(
      anEventThat(isActivityLifecycleEvent(ON_RESUME)),
      anEventThat(isActivityLifecycleEvent(ON_PAUSE)));
```

19.4 Checks on the Whole Stream

The third type of checks concerns the entire stream of events: they give the possibility, for example, to specify the number of all the events of a certain type in the whole sequence generated by an execution.

19.4.1 Match In Order

Structure

```
allEventsWhereEach(m).matchInOrder(m1, m2, ... , mn)
```

Description

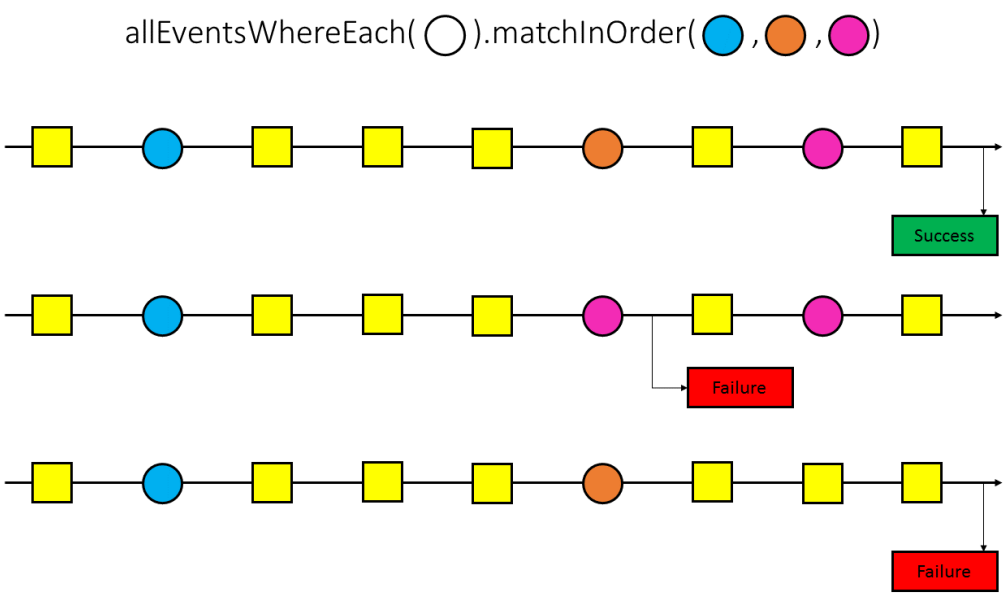
Checks that *all* the events that match `m` match in order $\mathtt{m}_1$, $\mathtt{m}_2$,..., $\mathtt{m}_n$.

FOL

$$\forall e\Big(match(e,\ m)\ \Rightarrow\ \exists i\Big(indexOf(e,\ m,\ i)\ \wedge\ match(e,\ m_i)\Big)\Big)$$

$$indexOf(e,\ m,\ i)\ \equiv\ \exists_{=i} e'\Big(match(e',\ m)\ \wedge\ before(e',\ e)\Big)$$

Visual



Note in particular the last case: the check implicitly states that the events must be exactly 3 and they must satisfy the three matchers. If the third one is missing, the check fails.

Code Example

```
/* Defines the only valid Fragment backstack changes sequence */
allEventsWhereEach(isFragmentBackstackChange())
   .matchInOrder(
      isFragmentBackstackPush(mainFragment),
      isFragmentBackstackPush(languageFragment),
      isFragmentBackstackPush(gameFragment),
      isFragmentBackstackPop(mainFragment));
```

19.4.2 Are Ordered

Structure

```
allEventsWhereEach(m).areOrdered(f)
```

Description

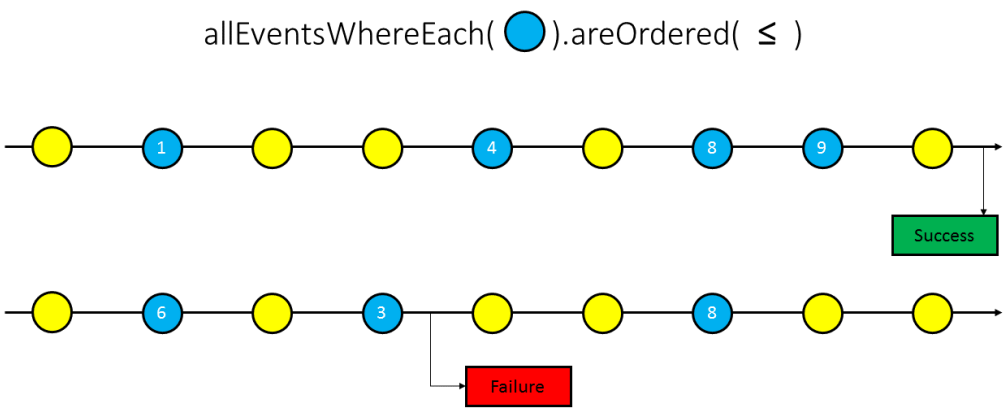
Checks that *all* the events that match `m` are in the order defined by the comparator function f (receives two events and returns an integer < 0 if the first is less than the second, 0 if they are equal, and > 0 if the first is greater than the second).

FOL

$$\forall e,\ e'\Big(match(e,\ m)\ \wedge\ match(e',\ m)\ \wedge\ before(e,\ e')\wedge$$
$$\neg\exists e''\Big(match(e'',\ m)\ \wedge\ between(e,\ e'',\ e')\Big)\ \Rightarrow\ ordered(f,\ e,\ e')\Big)$$

$$ordered(com,\ e1,\ e2)\ \equiv\ \Big(\Big(com\ <\ 0\ \Rightarrow\ before(e1,\ e2)\Big)\vee$$
$$\Big(com\ =\ 0\ \Rightarrow\ before(e1,\ e2)\ \vee\ before(e2,\ e1)\Big)\vee$$
$$\Big(com\ >\ 0\ \Rightarrow\ before(e2,\ e1)\Big)\Big)$$

Visual



Code Example

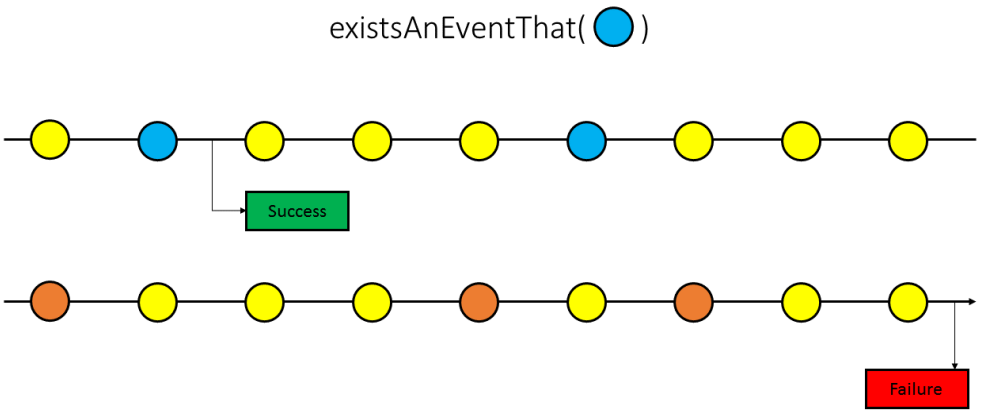
```
/* The text in the countdown is updated in the correct (inverse) order */
allEventsWhereEach(isTextChangeFrom(countdownView))
   .areOrdered(new Comparator<TextChangeEvent>()
           {
              @Override
              public int compare(TextChangeEvent lhs, TextChangeEvent rhs)
              {
                 String t1 = lhs.getText();
                 String t2 = rhs.getText();
                 return Integer.compare(Integer.valueOf(t2),
                                 Integer.valueOf(t1));
              }
           }));
```

Note that in Java 8 or above it is possible to replace the `Comparator` object with a *lambda* expression.

19.5 Existential Checks

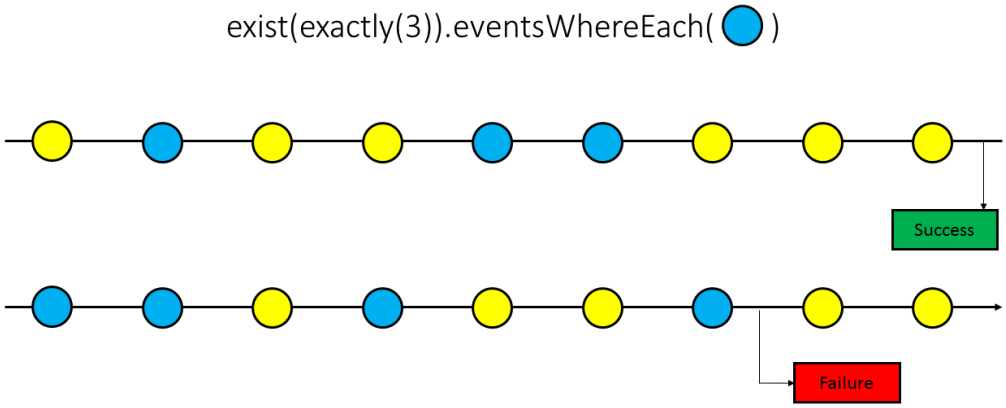
These checks assess the existence of one or more events in the stream. Similarly to the previous categories, only the formulas for "exactly" will be formalized.

19.5.1 Exists An Event

Structure	`existsAnEventThat(m)`
Description	Checks that at least one event that matches the matcher `m` exists anywhere in the sequence.
FOL	$\exists e\Big(match(e,\ m)\Big)$
Visual	
Code Example	`/* The database must be opened sooner or later during the execution */` `existsAnEventThat(isOpenDatabase());`

19.5.2 Exist Events

Structure	`exist(exactly(n)).eventsWhereEach(m)`

Description	Checks that the events that match `m` are exactly `n` in the whole stream.
FOL	$\exists_{=n} e\Big(match(e,\ m)\Big)$
Visual	

Code Example	`/* In this execution the system sends exactly 10 broadcasts */`
	`exist(exactly(10))`
	`  .eventsWhereEach(isSendBroadcast());`

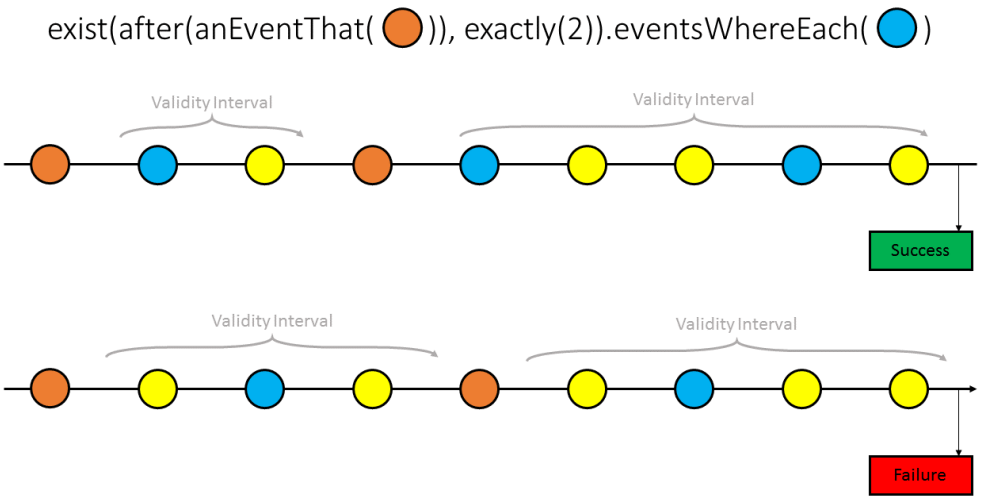
19.5.3 Exist Events After

Structure	`exist(after(anEventThat(m2)), exactly(n)).eventsWhereEach(m1)`
Description	Similar to the unbounded existential quantifier described in the previous section but with a restriction on the validity interval: exactly `n` events that match `m1` must exist exclusively after *an* event (i.e. at least once) that matches `m2`.

FOL

$$\exists_{=n} e1 \Big(match(e1,\ m1)\ \wedge\ \Big(\exists e2 \Big(match(e2,\ m2)\ \wedge\ before(e2,\ e1) \wedge \neg \exists e2' \big(match(e2',\ m2)\ \wedge\ between(e2,\ e2',\ e1) \big) \Big) \Big) \Big)$$

Visual



Code Example

```
/* At least one click must be followed by a broadcast */
exist(
   after(anEventThat(isClickOn(myButton))),
   exactly(1))
      .eventsWhereEach(isSendBroadcast());
```

19.5.4 Exist Events Before

Structure

```
exist(before(anEventThat(m2)), exactly(n)).eventsWhereEach(m1)
```

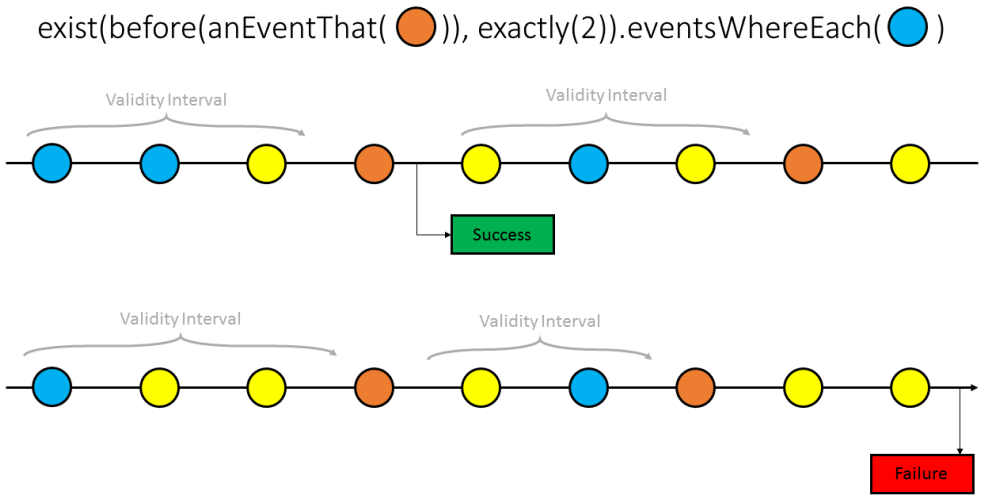
Description

Similar to the unbounded existential quantifier but with a restriction on the validity interval: exactly `n` events that match `m1` must exist exclusively before *an* event (i.e. at least once) that matches `m2`.

FOL

$$\exists_{=n} e1 \Big(match(e1,\ m1)\ \wedge\ \Big(\exists e2 \Big(match(e2,\ m2)\ \wedge\ before(e1,\ e2) \wedge \neg \exists e2' \big(match(e2',\ m2)\ \wedge\ between(e1,\ e2',\ e2) \big) \Big) \Big) \Big)$$

Visual



Code Example

```
/* At least once, the Service starts after a Fragment initialization */
exist(
   before(anEventThat(isServiceStart(mainService))),
   exactly(1))
      .eventsWhereEach(isFragmentLifecycleEvent(ON_CREATE));
```

19.5.5 Exist Events Between

Structure

```
exist(between(anEventThat(m2), anEventThat(m3)),
    exactly(n)).eventsWhereEach(m1)
```

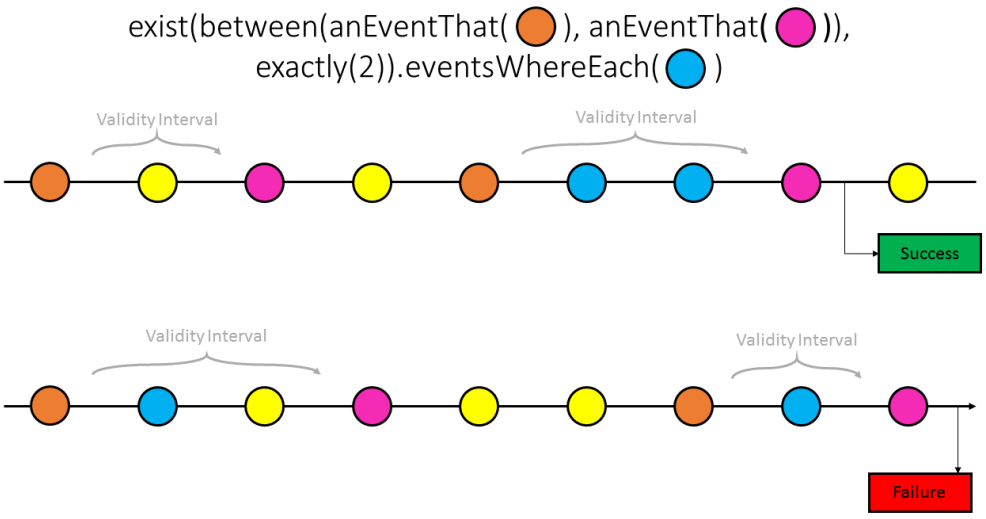
Description

Similar to the unbounded existential quantifier but with a restriction on the validity interval: exactly `n` events that match `m1` must exist exclusively between *a* pair (i.e. at least once) of events where the first matches `m2` and the second matches `m3`.

FOL

$$\exists_{=n} e1 \Big(match(e1,\ m1) \ \wedge \ \Big(\exists e2,\ e3 \Big(match(e2,\ m2) \ \wedge \ match(e3,\ m3) \wedge$$
$$between(e2,\ e1,\ e3) \ \wedge \ \neg \exists e2',\ e3' \big(match(e2',\ m2) \ \wedge \ between(e2,\ e2',\ e3) \vee$$
$$match(e3',\ m3) \ \wedge \ between(e2,\ e3',\ e3) \big) \Big) \Big) \Big)$$

Visual



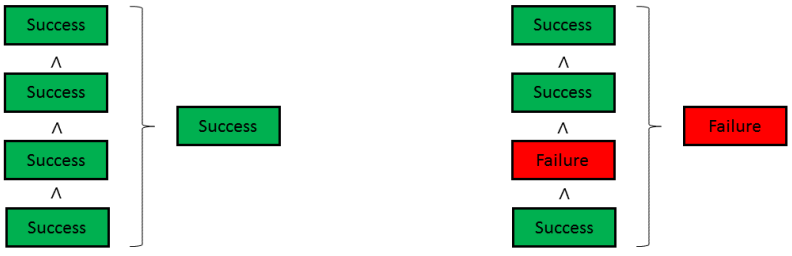
Code Example	

```
/* At least once, the Service must query the database during its execution */
exist(
   between(anEventThat(isServiceStart(mainService)),
        anEventThat(isServiceStart(mainService))),
   exactly(3))
      .eventsWhereEach(isDatabaseQuery());
```

19.6 Connectives between Checks

These constructs allow to specify the standard logic connectives between one or more of the previously defined consistency checks.

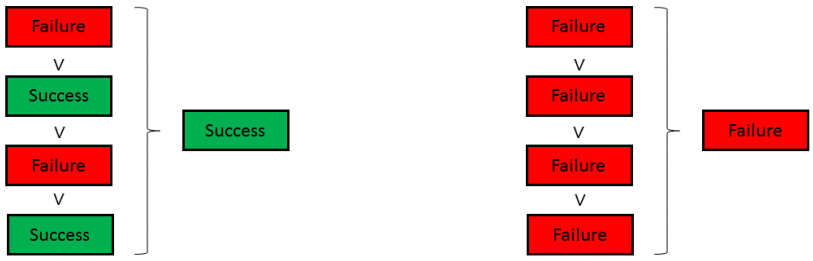
19.6.1 And

Structure	`allHold(c1, c2,...)`
Description	All sub-checks `c1`, `c2`, etc. must succeed.
FOL	$c1 \ \wedge \ c2 \ \wedge \ ...$
Visual	

Code Example

`/* The Service must only be started after the button is clicked */`
`allHold(`
`  anEventThat(isServiceStart(mainService))`
`  .canHappenOnlyAfter(isClickOn(startButton)),`
`  exactly(1).eventsWhereEach(isServiceStart(mainService))`
`  .mustHappenAfter(isClickOn(startButton)));`

19.6.2 Or

Structure
`anyHolds(c1, c2,...)`
Description
At least one sub-check must succeed.
FOL
$c1 \lor c2 \lor ...$
Visual


Code Example	

```
/* Either succeed or fail download (no situations where the user is not
    notified) */
anyHolds(
   exist(exactly(1))
      .eventsWhereEach(isDownloadSuccess()),
   exist(exactly(1))
      .eventsWhereEach(isDownloadError()));
```

19.6.3 Not

Structure	

```
isNotSatisfied(c)
```

Description	

Inverts the outcome of the sub-check.

FOL	

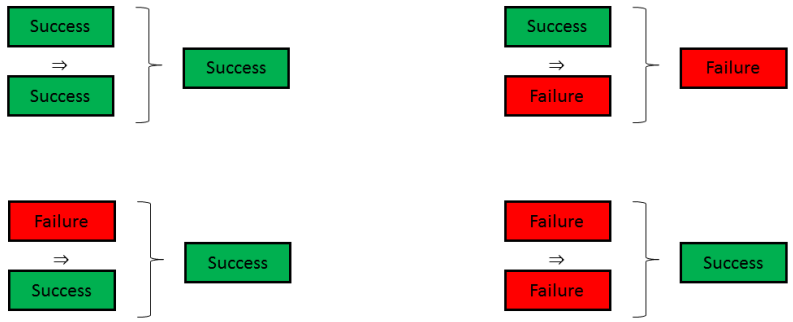
$\neg c$

Visual	

Code Example	

```
/* In this execution the email draft cannot be saved */
isNotSatisfied(
   existsAnEventThat(isSaveDraft()));
```

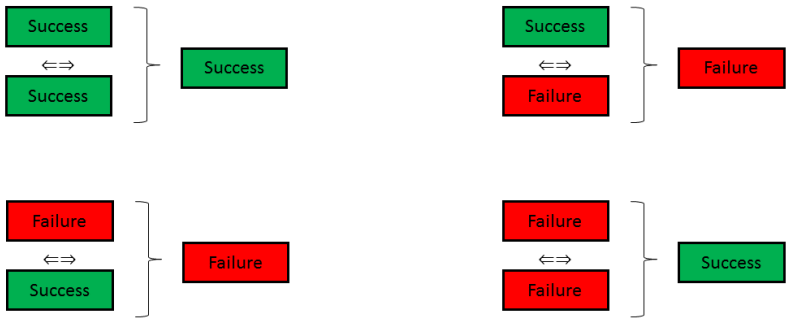
19.6.4 Single Implication

Structure	`providedThat(c1).then(c2)`
Description	`c2` is checked only if `c1` succeeds.
FOL	$c1 \Rightarrow c2$
Visual	

Code Example	`/* If the list order changes, then we must update the database */`
	`providedThat(`
	`  exist(atLeast(1))`
	`  .eventsWhereEach(isListOrderChange()))`
	`.then(`
	`  exist(exactly(1))`
	`  .eventsWhereEach(isSaveDatabase()));`

19.6.5 Double Implication

Structure	`isSatisfied(c1).iff(c2)`

Description	Succeeds only if both sub-checks fail or both succeed.
FOL	$c1 \iff c2$
Visual	

Code Example	`/* Send email if and only if the user clicks on the send button */`
	`isSatisfied(`
	`  existsAnEventThat(isClickOn(sendButton)))`
	`  .iff(`
	`  existsAnEventThat(isSendEmail()));`

20 Design

To implement the event-based testing approach described so far, the following main components are employed:

- Events: objects that represent events in the application.
- Event Generators: structures that create the events whenever something happens in the application.
- Checks: ways to specify the consistency rules defined in the previous section.
- Results: the outcomes of the checks.

- Event Monitor: the main interface of the system, receives the events, applies the checks and produces the results.

Figure 7 shows the design UML class diagram. The *EventMonitor* offers methods to add one or more *EventGenerator* objects to define the event stream, to add one or more *Check* objects to be verified and handles the *Result* objects in some way (e.g. makes a test case fail if the *Result* is a failure).

Each *Check* contains the logic of a consistency check: it receives each *Event* of the stream in order and acts accordingly. A *Check* has the possibility to short-circuit its behavior (i.e. interrupt the stream of events if the consistency check succeeds or fails before the end of the sequence). In any case, when it short-circuits or the stream comes to an end, the *Check* produces one and only one *Result* object.

Each *Result* contains an *Outcome* and a message describing it. In particular, an *Outcome* can be a *Success*, a *Failure* and a *Warning*. A *Warning* outcome has not been included in the formal specifications of the checks in the previous section for simplicity: its meaning is that the *Check* succeeded with some minor error or in a peculiar situation (for example a check that specifies that an event A always generates exactly one event B may return *Warning* if no event A has been found in the sequence).

A *Descriptor* allows to specify a *Check*, i.e. it is the main component of the temporal assertion language syntax. It describes one or more events of the stream and provides methods to express a condition on them. In particular, *SingleEvent* describes a single event of a certain type at a time, *SetOfEvents* describes a set of events of a certain type at a time and *AllEvents* describes all the events of a certain type in the whole stream. To determine the type of the events identified by the *Descriptor*, a *Matcher* object is used.

The cardinality of the descriptor *SetOfEvents* is defined by a *Quantifier* that has an integer value as input. A quantifier can be *Exactly*, *AtMost* or *AtLeast*.

Finally, a *CheckConnective* enables to transform one or more *Check* objects into a single *Check*, i.e. it specifies one of the standard logic connectives *Not*, *And*, *Or*, *SingleImplication* and *DoubleImplication*.

21 Implementation

For the implementation of event-based testing [27], the ReactiveX library [28] was chosen. This innovative reactive programming paradigm focuses on data flow: this means generating (statically or dynamically) and

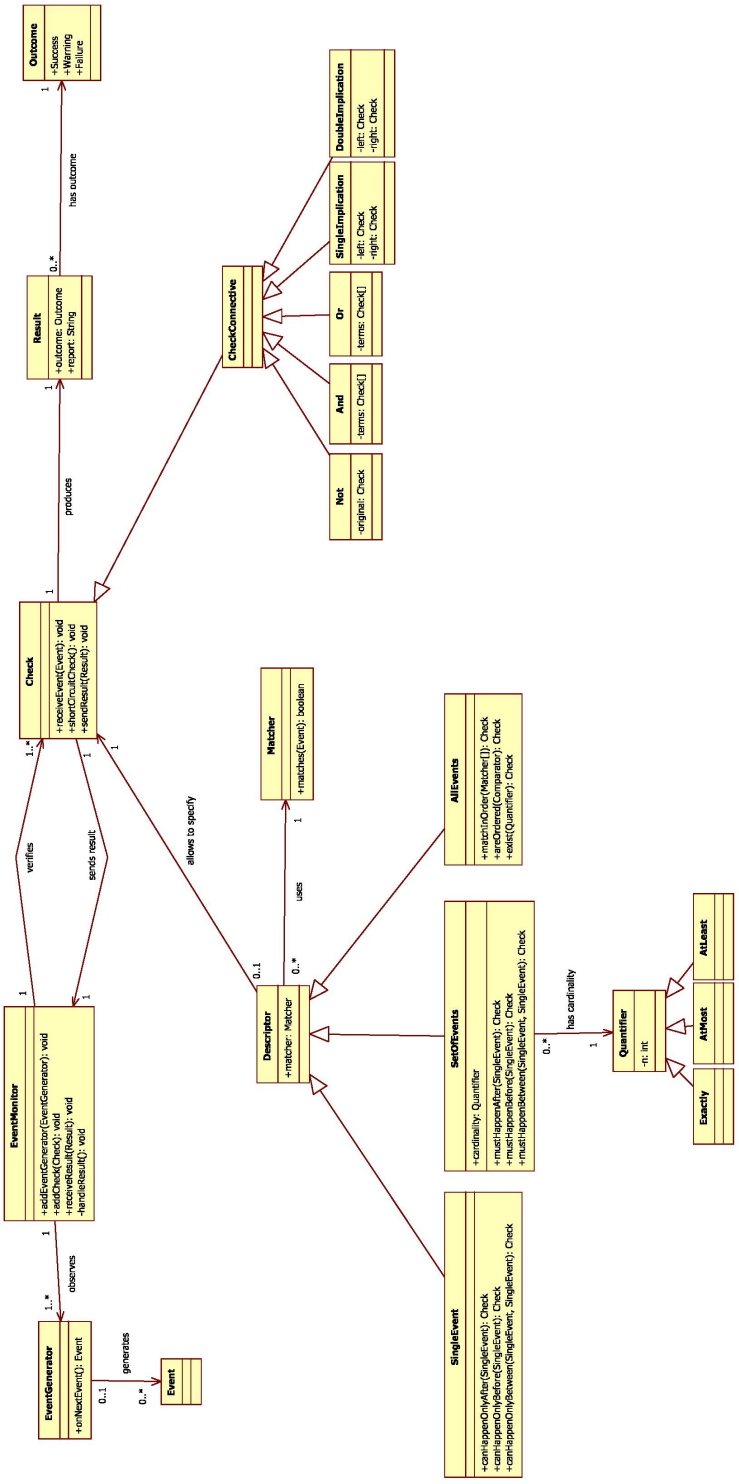


Figure 7: UML Diagram for the Event-Based Testing Design

propagating a flow of events, allowing interested components to react to them. The idea of reactive programming is fire-and-forget messaging: send a request and asynchronously wait for the response to be ready or, even more importantly, in case of response sets wait for individual results to be forwarded (without waiting for the whole set to be computed).

In the next sections a more detailed introduction to the ReactiveX library is presented, followed by the implementation of the proposed event-based testing tool.

21.1 ReactiveX

The Reactive Extensions (ReactiveX or Rx) are a reactive programming library to compose asynchronous and event-based programs. As defined by its authors, ReactiveX is a combination of the best ideas from:

- The Observer pattern (design pattern where a subject automatically notifies the so-called observers of its state changes).
- The Iterator pattern (design pattern where an iterator is used to traverse a collection of elements, like an array).
- Functional Programming (declarative programming paradigm where computation is performed via mathematical functions that are not allowed to change the state of the system).

This approach is asynchronous because many instructions may execute in parallel and their results (events) are later captured, in any order, by the listeners. For this reason, the main idea to perform a computation is not to call methods like in classic sequential programming but to define a mechanism to react to results when they are ready.

ReactiveX programming paradigm is essentially based on three steps:

- Create: the Observable components are used to generate event or data streams.
- Transform: Operators allow to modify the event streams (e.g. change each event or filter out some of them) and compose them (e.g. join two streams).
- Listen: Subscriber components can listen to event streams and receive their elements one by one, to perform some computation.

An Observable is in charge of emitting events: it generates zero, one or more than one events (depending on the specific implementation), and then terminates either by successfully completing or with an error. Observables

can be "hot" (emit events even if no Subscriber is listening) or "cold" (emit events only after a Subscriber is registered).

An Observable can be modified by an Operator: most Operators can be applied on an event stream generated by an Observable to return a new modified event stream. For example, the Operator `map(function)` allows to apply a function to each event in the stream (e.g. a stream of numbers modified by a map with a summing function may become a stream where all the original numbers have been increased by 1). Since the result of an Operator is an Observable, Operators can be applied in chain (i.e. apply an Operator on the result of another Operator) to achieve complex modifications.

A Subscriber consumes the events emitted by the Observables (that can either be "original" or the result of one or more Operators). Subscribers react to asynchronous results: for example, an Observer may send a network response whenever it is ready, the Subscriber receives it and shows the information to the user.

The advantages of the Rx paradigm are:

- Cross-Platform: it is available in many programming languages, like RxJava, RxSwift, RxJS, RxPHP, etc.
- Can be used for any application, from Front-End (e.g. UI events and API responses) to Back-End (e.g. asynchronicity and concurrency).
- Operators usually make computations less verbose and more readable.
- Error handling: if an error occurs in one of the steps of the computation the exception is automatically intercepted by Rx and forwarded to the user via appropriate callbacks.
- Easy concurrency: Rx allows to easily specify in which threads the components should be run, without worrying about implementation details.
- Extensible: a developer can define custom Observables, Operators and Subscribers to achieve anything an application may require.

The characteristics of ReactiveX make this programming paradigm very suitable for many applications. It is successfully employed in industry: examples of users are Microsoft, Netflix and GitHub.

21.2 RxJava and RxAndroid

RxJava [29] is the Java implementation of ReactiveX. It is an open source project initially developed by Netflix for server-side concurrency. The main

reason for its adoption and development was to avoid Java Futures (results of asynchronous computation) and callbacks because both are expensive when composed, especially if nested.

In addition to implementing all functionalities of ReactiveX paradigm (Observables, Subscribers, Operators, etc.), RxJava also has the advantages of being:

- Lightweight: zero dependencies and single small JAR to contain the whole library.
- Polyglot: supports Java 6 or higher and JVM-based languages such as Groovy, Clojure, JRuby, Kotlin and Scala.
- Composable: several RxJava Libraries are available to developers to manage common use cases.

RxAndroid [30] is a RxJava module that provides specific bindings for the Android platform, for example to easily specify the main (UI) thread as the observing thread or, more in general, a custom Looper. RxAndroid can in turn be extended by other modules, some of which are listed in the next section.

21.3 Events Observable in Android

Several modules allow Android developers to observe many events inside an application. For example, we can observe:

- UI Widgets: RxBinding [31] module allows to observe user inputs or changes on UI widgets like TextView (e.g. clicks, text change), app bar menu (e.g. option selected), etc.
- Settings: RxPreferences [32] module gives the possibility to receive events from the Shared Preferences system (storage provided by Android to save the app settings).
- Files: RxFileObserver [33] fires events for file accesses or changes.
- Database: StorIO [34] allows to manage and observe an SQLite database.
- Network: ReactiveNetwork [35] detects network changes (e.g. WiFi or mobile connection).
- External API Requests: Retrofit [36] offers Observables to receive network responses (REST client).
- Broadcasts: RxBroadcast [37] builds an event stream from a Broadcast Receiver results.

- Location: ReactiveLocation [38] allows to observe location changes.
- Sensors: ReactiveSensors [39] fires events from hardware sensors.
- Permissions: RxPermissions [40] enables the reception of events from the permissions manager.
- Google Maps: RxGoogleMaps [41] fires events related to Google Maps.
- Google Wear: RxWear [42] allows to observe messages to and from a connected smartwatch.

The utility module RxLifecycle [43] gives the opportunity to bind the listed Observables to the lifecycle of an Activity or a Fragment to avoid leaks. For example, without any binding to the lifecycle, an Observable that emits text change events on a TextView never ends (i.e. never calls `onCompleted()` or `onError()`) because it has no way of understanding when the text stops changing. In that situation the Observable will run even after the Activity has been removed and, keeping its reference, won't allow the garbage collector to delete the instance (memory leak). Thanks to RxLifecycle the developer can bind the Observable until a lifecycle event occurs (e.g. Activity is stopped), enabling it to terminate correctly.

21.4 The System

The structure of the implementation of event-based testing is similar to what has been designed in section 20.

A class diagram of the system is proposed in figure 8. Before starting the detailed explanation, note that many classes, like the descriptors, have a private constructor and a static method to create an instance. Similarly to the standard way of building assertions in JUnit, this is just syntactic sugar that allows to statically import the methods and to avoid writing the `new` keyword. These methods, and as a consequence their classes, are worded as close as possible to the English language: for example, the descriptor for single events is not called *SingleEvent* but `AnEventThat` in order to build readable statements like:

```
anEventThat(isTextChange())
   .mustHappenAfter(anEventThat(isButtonClick())
```

21.5 Event Monitor

The `EventMonitor` is a singleton class that is set up via the `initialize()` method. Once this is done, the developer can add Observables

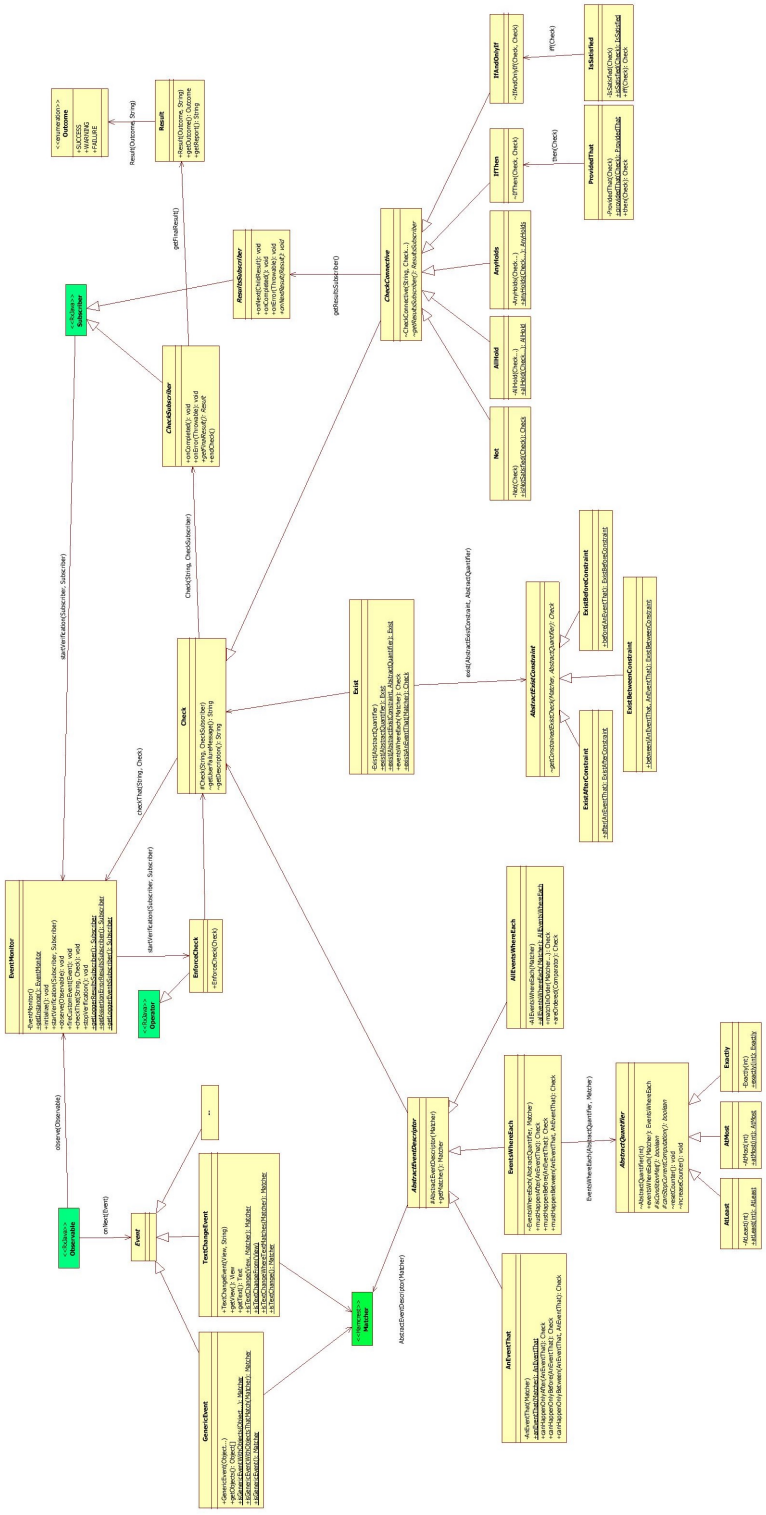


Figure 8: UML Diagram for the Event-Based Testing Implementation

via `observe(Observable)` to build the event stream and checks via `checkThat(String, Check)`. This latter method also takes a String message as parameter representing an error shown if the check fails, similarly to standard assertions.

At this point a call to `startVerification(Subscriber<Event>, Subscriber<Result>)` is performed to start the validation process on the defined stream: the two Rx Subscribers are used by the developer to receive all the events of the stream in order and all the results of the checks respectively. The class provides some static methods to get simple pre-generated Subscribers for this purpose, e.g. passing `EventMonitor.getAssertionErrorResultsSubscriber()` as the second parameter the app crashes (or makes the test case fail) if a failure result is found.

When the developer calls `stopVerification()` the stream is interrupted and all the remaining check results are generated.

Finally, the class also supplies the utility method `fireCustomEvent(Event)` that allows to directly generate an `Event` without creating an `Observable`.

Note that the `EventMonitor` observes the events on the thread specified by each Observable (usually the main thread) and runs the checks in a separate thread to avoid possible slowdowns of the UI thread.

The `EventMonitor` can be used as:

- Runtime monitor during the manual debugging phase of the application development: just like the standard assertions placed in production code, the developer can use the monitor for example inside an Activity to log all events and the results of the checks.
- Testing mechanism: the developer can start the monitor at the beginning of a test and add some specific checks, then execute the actions allowed by the chosen testing framework and finally stop the monitor at the end. The monitor works as a standard assertion mechanism, making the test fail in case a consistency check is not successful. Note that this mechanism, since it's not framework-specific, can be used for any type of test, from unit to UI, and with any library. A code example of the EventMonitor used as a testing mechanism can be found in listing A.3.

21.6 Event Generators

As introduced, the event generators are implemented as `Observable` objects provided by RxJava, an example of which can be found in listing A.5. Each Observable added to the `EventMonitor` builds the event stream: in particular, all the Observables are combined using the RxJava Operator `merge()`, which transforms a set of Observables into a single one that emits all the events in their respective order.

The events generated by these Observables are subclasses of `Event`, of which just two have been reported in the diagram since there could be hundreds of them. An `Event` class may also provide one or more static methods to create matchers, as is shown for example in listing A.4.

21.7 Checks

A `Check` implements is logic using a `CheckSubscriber`, a subclass of the RxJava `Subscriber`. This component receives all the events of the stream via `onNext(Event)` and performs some internal computation, possibly short-circuiting the validation calling `endCheck()`. The callback `getFinalResult()` is called to return the unique `Result` of the check. An example of check implementation can be found in listing A.6.

The `EventMonitor` applies a check on the event stream by calling the Operator `EnforceCheck` on it, which is in charge of transforming a stream of `Event` objects into a stream containing a single `Result`.

Connectives between checks are implementations of `CheckConnective`, which is in turn a subclass of `Check`. Each of them implements a `ResultsSubscriber` that, analogously to a `CheckSubscriber`, receives the results of its terms, performs some computation and then produces a single `Result` as output. `Not`, `AllHold` and `AnyHolds` (implementations of the logic *negation*, *and* and *or* respectively) provide static methods as constructors (e.g. `allHold(check1, check2, check3)`), while `IfThen` and `IfAndOnlyIf` (implementations of *single* and *double implication*) are built starting from the `ProvidedThat` and `IsSatisfied` classes respectively (e.g. `providedThat(check1).then(check2)`), to make the calls more human-readable.

21.8 Descriptors

The descriptors are subclasses of `AbstractEventDescriptor`. `AnEventThat` (single event) and `AllEventsWhereEach` (all the events in the whole stream) provide static constructors, while `EventsWhereEach` descriptors (sets of

events) do not have one because they are built starting from a quantifier, e.g. `exactly(10).eventsWhereEach(...)`.

The `Exist` class provides the means to express the existential quantifiers, i.e. `existsAnEventThat(...)` and `exist(...).eventsWhereEach(...)`. The latter type uses quantifiers and constraints to define the cardinality of the set in the first case and the possible validity interval in the second.

Existential constraints are subclasses of `AbstractExistConstraint`. They simply implement an abstract method to define the constraint logic and provide a static method to be used in the existential check expressions.

Quantifiers are subclasses of `AbstractQuantifier`. Each of them uses an internal counter and overrides several methods like `isConditionMet()` and `canStopCurrentComputation()` to implement its logic. All of them have a private constructor and a static method like the descriptors.

The matchers that describe the type of events identified by each descriptor are implemented using the external library JavaHamcrest [44], a well-known way to describe objects. The library provides several matchers on the most common objects (like Strings, e.g. `isEmptyString()`, `startsWith(String)`, etc.), some connectives to compose matchers (e.g. `allOf(Matcher...)`) and the means to implement custom matchers, which have been used to define matchers for the events.

22 Evaluation

22.1 Real-World Case Study

We have chosen the WordPress for Android app as a real-world example of event-based testing application. The idea is to define some temporal assertions on the latest version at the time of writing [25], to discuss the expressiveness of the language, the generation effort and the types of events that can be observed.

In particular, this evaluation focuses on the `EditPostActivity`, which allows the users to write and publish a post on their blog. The Activity contains an instance of `EditorFragment`, which manages the fields for writing the post content, and uses the `PostUploadService` to publish the post and the `MediaUploadService` to add an image or video to the content.

22.1.1 Structure

The test suite of the aforementioned example is defined in Espresso and can be visualized in listing A.3. The external structure of the test is entirely defined with the Espresso framework, the event-based testing library only requires to initialize, start and stop the Event Monitor, and to add Observables and checks.

More in detail, the Activity under test is defined by a `IntentsTestRule`, an object similar to `ActivityTestRule` but specific for Intent mocking (used to simulate the image selection), provided by the Espresso Intents [45] module. The test case uses the callbacks provided by the `IntentsTestRule` to manage the Event Monitor:

- During `beforeActivityLaunched()`, called before each test starts, the monitor is initialized.
- During `afterActivityLaunched()`, called just after the tested Activity is started, Observables and checks can be added. Since the method is called for each test, the Observables and checks added here are valid for all the defined test cases.
- During `afterActivityFinished()`, called after each test ends, the Event Monitor verification is terminated.

After the definition of the `IntentsTestRule` object, the developer can specify one or more Espresso test cases that use the Event Monitor. First of all, in each of them further Observables and checks, which are specific for that test, can be added, and then the validation process is started. At this point, the developer defines the standard UI actions of an Espresso test, to drive the application execution.

Given this structure, the Event Monitor executes in the background of a normal Espresso test, observing all the events generated by the UI actions and validating the defined checks, similarly to the standard assertions mechanism.

22.1.2 Events

The test for the `EditPostActivity` observes both imported and custom events, including:

- Text change events on title and content fields of the post.
- Toast (quick and short messages displayed to the user as feedback) display.

- Media upload start, progress and completion.
- Post upload start.
- Click on menu options, like the "Publish" button.
- Lifecycle events of the Activity and Fragment under test.
- HTML mode toggle event (while writing the post the user can switch between visual and HTML modes).

The imported events include for example the text change events of title and content of the post, observed thanks to the RxBinding module. Since they are easily observable by just calling a single method, these events do not require any particular coding effort by the developer.

Most of the events, however, are specific for the WordPress application, for example the HTML toggle event and the media upload progress event. As such, these events need to be created by the developers, together with the proper Hamcrest matchers, similarly to what can be seen in listing A.4. The generation effort of the event classes is very low, to define the custom events for the WordPress application less than a couple of minutes per class were required.

Once the custom events are defined, they either need to be fired manually (i.e. modifying the production code of the application) using the `EventMonitor#fireCustomEvent(Event)` method (in case of lack of valid listeners, like in the case of the HTML toggle event) or an Observable must be generated. An example of the latter case is the media upload progress events Observable, defined in listing A.5. The generation effort of the Observables highly depends on the specific component (i.e. available listeners): in this case, to define the Observable for media upload progress events took approximately five minutes.

22.1.3 Checks

The defined checks, valid for all the tests, enforce the following conditions:

- The HTML mode cannot be activated while a media item is uploading, for an application logic constraint.
- The post title and content cannot change after the publication procedure has started.
- Clicking on the "Publish" button must always be followed either by a Toast display (e.g. an error occurred) or by the publication procedure start, i.e. there cannot be situations where the button is unresponsive.

- If the Editor Fragment is detached from the host Activity (e.g. the application is closed) while a media upload is in progress, the upload must be interrupted.
- Before the publication procedure starts, all media uploads must complete.
- The media upload progress values must be sent in order, i.e. there cannot be situations where the Service sends incoherent percentages of uploaded content.

The test case `testPublishError()` also adds a specific check that states that the click on the "Publish" button must generate an error saying that the post is empty.

Once the checks are defined, the developer can stress-test the application (e.g. try several sequences of actions, drive the application lifecycle, etc.) to see if the defined temporal assertions hold in all possible situations.

Regarding the generation effort, once the condition to be verified has been identified, the generation of the check specification took approximately five minutes each.

22.1.4 Check Results

Since the objective of this evaluation was not to stress-test the application for bug detection but just to show how the library can be applied in a real-word context, only a couple of Espresso tests were specified to drive the application execution and, as a consequence, all the defined check results were successful.

For example, the outcomes printed by the tool during the `testUploadImage()` test were:

- `[SUCCESS] No event that is any HTML toggle event happens between a pair of events where the first is media upload progress and the second is any media upload outcome (success/failure/cancel)`
  `REPORT: The condition was never met between any of the 1 pairs`
- `[SUCCESS] IF (Exists an event that is post upload start) THEN (Every event that is post change happens before an event that is post upload start)`
  `REPORT: Every event that is post change was found before {Post upload start}`

- `[SUCCESS] Every event that is click on menu option <2131821737> is followed by at least 1 events where each (is post upload start or is any toast display)`
  `REPORT: Check verified for each of the 1 events where each is click on menu option <2131821737>`
- `[SUCCESS] IF (exactly 1 events where each is a fragment lifecycle event "onDetach" are between at least one pair of events where the first is media upload progress and the second is any media upload outcome (success/failure/cancel)) THEN (Every event that is a fragment lifecycle event "onDetach" is followed by at least 1 events where each is media upload cancel)`
  `REPORT: The pre-condition didn't hold: The condition was never met between any of the 1 pairs`
- `[SUCCESS] NOT TRUE THAT (at least 1 events where each is post upload start are between at least one pair of events where the first is media upload progress and the second (is any media upload outcome (success/failure/cancel) or is media upload progress))`
  `REPORT: The condition was never met between any of the 131 pairs`
- `[SUCCESS] All events where each is media upload progress are in the order defined by the comparator: org.wordpress.android.ui.posts.EditPostActivityTest11@a746538`
  `REPORT: All 131 events were in order`

The printed results first display the outcome of the check (as mentioned before, in this case all successes), then a sentence explaining the assertion meaning, and finally a report on what has been recorded. This latter part is not particularly useful for successful checks, but in case of failures it displays which events caused the issue to better identify the problem.

22.1.5 Performance

In the case of the defined test suite, executing the same Espresso test with and without the Event Monitor (with 2 tests, 7 checks each and more than 130 observed events) a performance degradation of less than 3% was registered (an average of 40 seconds with the monitor active against an average of 39 seconds without any event-related condition).

22.2 Discussion

22.2.1 Applicability

The event-based testing library is most suitable for applications that are characterized by a high number of different operations that can possibly run concurrently. This usually means applications that heavily employ the network (i.e. that query the web in background), the device sensors (e.g. receive GPS events to update a map) and/or, in general, that perform several sub-routines in parallel thanks to Services and threads.

However, the library may be applicable also to more "sequential" applications, especially for the event counting and ordering capabilities, e.g. the developer might be interested in easily quantifying the number of click events during an execution.

Event-based testing is especially useful for applications that already use the ReactiveX library in production code since the Observables to generate the events are already built.

The library can be applied both as a testing support tool, expressing conditions during test cases built in Espresso, Robolectric or any other framework, or as a debugging tool directly in production code in the early stages of development, to observe how the event stream is built.

22.2.2 Effectiveness

In general, to discuss the effectiveness of the event monitor tool we can focus on three aspects:

- Event Stream Composition: it may be noted that, thanks to the available listeners and RxAndroid extensions, many events can be easily observed in an Android application, like user interaction on most View components, sensor data, broadcasts, etc. However, there are many events that cannot easily be observed, mainly because Android does not provide any listener on which to build the ReactiveX Observable (e.g. no listener is available to see when a View element becomes visible or hidden) or because it only allows to manage a single listener (e.g. only one focus change listener is allowed per View component and, if the developer uses one inside the production code of the application, RxJava cannot use another to build the Observable). This latter problem could be solved by employing a composite listener (a single listener that notifies in turn other sub-listeners), but it would require additional work and refactoring. Of course, these two issues

can limit the event stream composition and, as a consequence, the checks that can be expressed on it.

- Assertions Language Expressiveness: the proposed assertion language allows to express temporal constraints (events can happen only after/before/between other events), causality (events must happen after/before/between other events), existence, ordering and quantification among events, as well as correlating check results via logic connectives. This expressiveness gives the possibility to specify the majority of temporal operators, but may be extended in future work to support more complex assertions, for example like the possibility of correlating the matched events between several checks.
- Results Handling: the result of each consistency check contains an outcome (success, failure or warning) and a report message to understand its meaning. For example, the result of the check asserting that a text change event can only happen after a Fragment has started can contain the report *The event {Text change "MyText" from TextView1} was found before an event that is a Fragment lifecycle event "onStart"*, which clearly shows what event made the check fail. Moreover, the event monitor offers the possibility of printing the whole event stream as it was registered during the execution, allowing the developer to analyze it to help in application debugging.

22.2.3 Usability

As mentioned in section 22.1, the generation effort to build components used by the system depends on the specific application and on what the developer wants to achieve.

Concerning the composition of the event stream, if the events that need to be observed are covered by one of the RxAndroid extensions listed in section 21.3, the generation of the Observables is straightforward. However, if the developer needs to observe Android events not covered by any extension or events specific to the application under test, it may be required to build custom Observables. Usually, creating a custom Observable is not a complex task because of the RxJava library support, but in some cases it may require some effort.

With regard to the consistency checks, the effort required for their specification is mostly due to the "logic" and temporal approaches. Developers are used to express sequential conditions on the current state of the component under test, so they might find difficult to adapt to the temporal assertions language, that requires to specify conditions that must hold throughout the entire test execution. Nevertheless, care has been taken to offer a language

as close as possible to the English language, to allow the developers to easily write and understand the meaning of a check at first glance.

Finally, it may be noted that the library is not framework-specific. This means that it can be easily employed in any testing framework, like Espresso, Robolectric or the built-in instrumented unit testing, provided that it is able to observe all the required events in the application.

22.2.4 Performance

As it can be observed in section 22.1, the performance overhead introduced by the event-based testing library is, in general, negligible.

This is due to the fact that the RxJava library provides a very lightweight and scalable event handling system and to the implementation of the checks as Subscribers that perform very simple operations at each event of the stream. Moreover, the check logic is implemented in a separate thread from the main UI thread of the Android application, in order not to decrease the app responsiveness.

It can also be noted that is some situations it may be required to edit the production code to insert custom event firing, when a listener is not available. This, however, does not create any issue when the application is run outside the event-based test case. If the `EventMonitor` component is not initialized, any call to add Observables, checks or fire custom events is ignored and so the performance is not degraded. Moreover, it is possible to completely remove these calls in "release" builds (e.g. the application is made available on the Play Store) with tools like ProGuard [46].

22.2.5 Extensibility

The library can be extended by the developer to adapt it to the specific needs of the application under test. In particular, a developer can implement:

- Custom Events and Matchers: it is enough to extend the `Event` class, providing any parameter and Hamcrest matcher required.
- Custom Observables: to inject custom events in the stream the developer just needs to create a ReactiveX Observable.
- Custom Subscribers: the library provides simple Subscribers for both the event stream and the check results, but the developer can build custom behaviors by simply implementing a ReactiveX Subscriber.

This allows to manage the event monitor results in any way, from logging to saving them in a database or file.

- Custom Checks: it is also possible to build customized checks and check connectives, by implementing a ReactiveX Subscriber that, receiving all the events of the stream, keeps a state and returns one final result.

22.2.6 Comparison with Testing Frameworks

Comparing temporal assertions with the standard Android testing frameworks like Espresso, Robolectric and the instrumented unit tests, the event-based testing technique can be seen both as an extension and as an alternative.

It is an extension because the event monitor is a tool that needs to be executed during a standard test case, since it is necessary to define the tested components and the actions (e.g. clicks) to be performed.

It is, however, also an alternative to the standard assertions that can be used during these tests. If we consider the example of assessing that an AsyncTask starts only after the user clicks on a button, it can be easily expressed with a temporal check but it is very complex to specify it with standard assertions. It would require for example to continuously poll the AsyncTask variable before the click action to check if the task is running (or to manually setup a test listener invoked when the task starts) and, using boolean variables, assert that one is always false before the other becomes true.

22.2.7 Comparison with Race Detection Tools

Comparing the proposed solution with race detectors like EventRacer and DEvA, it is first important to stress the fact that the approach is different. The mentioned tools try to detect possible race conditions, either dynamically or statically, without requiring the developer to write any test case. In the event-based testing library, instead, the focus is on manually generated test cases that allow to verify conditions on the event stream. For this reason, event-based testing is proposed as a complementary tool, and not as an alternative.

From the point of view of race conditions, it can be noticed that the two techniques mostly focus on different types races.

- EventRacer and DEvA focus their attention mainly on data races, i.e. they detect when two or more callbacks that use the same variable may

be in conflict. If we consider the example (reported in listing A.7) of a button click callback where a Service is invoked and a connection callback where the Service is initialized, a race might be detected by the tools. In this example, it does not necessarily mean that the click callback cannot happen before the connection one, but just that the developer should handle that situation, e.g. by checking if the Service variable is null in the click callback before using it and, if so, just show a message to the user.

- The event-based testing approach, instead, focuses on race conditions between events governed by a causality constraint, i.e. a given event can happen only after/before/between other given events. This means that a race condition between these events cannot occur in any possible execution, e.g. the callback *A* under no circumstances can happen after the callback *B*. In addition to this, the concept of race condition in this context is not necessarily dependent on data shared by two callbacks, but a relation between any event that is registered during the execution. An example (reported in listing A.8) of this type of race condition can be the situation where the application logic imposes that text change events of an email contents can happen only before the user clicks on the "Send" button.

Given this distinction, we can have races that:

- Can only be recognized by race detection tools: the example of the click-connection race cannot be expressed with temporal assertions since the two callbacks may happen in any order. The proposed language is not designed to express pure data race conditions, but to operate on the causality relation between method callbacks.
- Can only be tested by the event-based approach: the example of the text changes before the click cannot be recognized by the race detectors since there is no variable in common between the click and the text change callbacks.
- Can be both recognized by race detectors and tested with the event-based technique: an example (reported in listing A.9) of this is the event of placing a marker on a map when this has not been initialized yet. These two events generated by the application without user interaction must never be in the wrong order, otherwise the application might crash or result in the marker not being placed on the map. This condition can be expressed with a temporal assertion and, since we have a common variable (the map object), it may be recognized by race detectors.

Moreover, with the proposed library the developer is able to focus on events in general, and not only on race conditions like the aforementioned tools.

This means that there is the possibility of looking at the event stream generated by an execution from a broader perspective, being also able to express conditions on existence, ordering and quantification of events.

Part VI

Conclusions and Future Work

23 Conclusions

This thesis presented a comprehensive testing approach that focuses on the events generated during the execution of an Android application, covering different development stages and tasks. It analyzed first a subset of events, lifecycle transitions, and then focused on a more generic concept of events.

Section 16 proposed a static analysis technique for lifecycle testing, focusing on the management of selected components according to the host Activity/Fragment lifecycle. This approach is mainly useful in the early stages of development to warn the developer of possible criticalities, concerning performance or correctness.

In section 17, instead, it is reported the description of a dynamic approach for lifecycle testing. The idea is to provide a more complete support for lifecycle testing, hiding to the developers the complexity of driving the transitions and so allowing them to just focus on actions and assertions defined in the test case. The relevance of thoroughly testing an Activity or a Fragment with regard to its lifecycle is an important step to ensure confidence of correctness of the component, since lifecycle handling can often generate many issues.

Part V proposed an event-based testing library. The idea is to first build an event stream, observing different Android events defined by the developer, and then to express consistency checks on it, using the defined temporal assertions language. This allows to check for a specific type of race conditions and to assess existence, ordering and quantification of the events generated during the application execution.

Finally, it is interesting to note that the three techniques proposed by the thesis can all be employed at the same time. For example, the developers may use the static lifecycle checks to easily spot problems while coding the application, then define test cases that exploit both temporal assertions and lifecycle tests to verify the consistency in the event stream during critical lifecycle transitions.

24 Future Work

Possible future extensions on the proposed work include:

- For static lifecycle testing:
  - Provide more static checks implementations, such as camera, location updates, etc.
  - Extend the static analysis scope by providing more complex checks. It may be possible not only to focus on release, double instantiation and best practices but also, for example, to build a complete graph of usages per each component with regard to the host Activity/Fragment lifecycle and to analyze it for possible critical aspects.
  - Provide a mechanism to automatically fix, whenever possible, the detected problems.
- For dynamic lifecycle testing:
  - Provide a better testing interface: for example, it may be possible to avoid the subclassing and the implementation of the abstract callback methods by providing annotations that automatically generate the required tests. This would allow to define more tests of the same type (e.g. two rotation tests in the same test suite) and to avoid methods that return null if the developer is not interested in some test cases.
  - Offer the possibility to test not only Activities but also Fragments. This would require to create a new testing mechanism (e.g. Fragment Test Rule), since it is not provided by Android.
  - Automatically generate appropriate lifecycle test cases for selected Activities or Fragments, with automated assertion placement.
- For event-based testing:
  - Provide a better support for building the event stream, i.e. allow in some way to observe events that do not have an Android listener or to easily observe other events without the need to edit the production code (e.g. it may be possible to observe any callback invocation with aspect-oriented programming).
  - Increase the assertion language expressiveness, adding more checks and allowing to specify dependencies among sub-checks in check connectives.

– Allow, if possible, to specify the consistency checks in an easier and less verbose way.

– Provide a better check results handling: for example, it may be possible to write a detailed report on the Event Monitor outcome, with cleaner interface (e.g. generating an HTML file) and more information to better understand and reproduce the behavior in other tests.

– Evaluate the possibility of reordering the events in the stream (e.g. by automatically generating delays in the invocation of callbacks) to better test the consistency among events.

– Provide automated assertion placement support, allowing the developer to start from relevant pre-generated temporal assertions.

Appendices

A Code Listings

Listing A.1: Example of lifecycle test. In particular, it is the test case defined for the `MyProfileActivity` in the WordPress app that checks if, after a rotation, the inserted name is kept.

```
@RunWith(AndroidJUnit4.class)
@MediumTest
public class MyProfileActivityTest extends
    ActivityRuleLifecycleTest<MyProfileActivity>
{
    @Override
    protected ActivityTestRule<MyProfileActivity>
        getActivityTestRule()
    {
        return new ActivityTestRule<>(MyProfileActivity.class);
    }

    @Nullable
    @Override
    public RotationCallback testRotation()
    {
        return new RotationCallback()
        {
            private String name;

            @Override
            public void beforeRotation()
            {
                // Open first name dialog
                onView(withId(R.id.first_name_row))
                        .check(matches(isDisplayed()))
                        .perform(click());

                // Type name
                name = "MyFirstName"+(new Random().nextInt(100));
                onView(withId(R.id.my_profile_dialog_input))
                        .check(matches(isDisplayed()))
                        .perform(replaceText(name));

                // Confirm
                onView(withText("OK"))
                        .perform(click());

                // Check name in the button
```

```
38                  onView(withId(R.id.first_name))
39                          .check(matches(allOf(isDisplayed(),
                                withText(name))));
40              }
41
42              @Override
43              public void afterRotation()
44              {
45                  // Check name in the button
46                  onView(withId(R.id.first_name))
47                          .check(matches(allOf(isDisplayed(),
                                withText(name))));
48              }
49          };
50      }
51  }
```

Listing A.2: Example of race condition: if the user rotates the device while the AsyncTask is running, the application crashes with an `IllegalStateException`.

```
1   public class Activity1 extends AppCompatActivity
2   {
3       @Override
4       protected void onCreate(Bundle savedInstanceState)
5       {
6           super.onCreate(savedInstanceState);
7           setContentView(R.layout.activity_1);
8
9           new MyAsyncTask().execute("parameter1", "parameter2");
10      }
11
12      private class MyAsyncTask extends AsyncTask<String, Void,
            String>
13      {
14          private ProgressDialog progressDialog;
15
16          @Override
17          protected void onPreExecute()
18          {
19              progressDialog = new ProgressDialog(Activity1.this);
20              progressDialog.setMessage(getString(R.string.loading));
21              progressDialog.setCancelable(false);
22              progressDialog.show();
23          }
24
25          @Override
26          protected String doInBackground(String... params)
27          {
```

```
28              String operationResult = null;
29
30          /* Perform some background operation, e.g. download image
                from the Internet */
31
32              return operationResult;
33          }
34
35          @Override
36          protected void onPostExecute(String result)
37          {
38              progressDialog.dismiss();
39          }
40      }
41  }
```

Listing A.3: Example of Espresso test that uses the event-based testing library, defined for the `EditPostActivity` of the WordPress app.

```
1   @RunWith(AndroidJUnit4.class)
2   @MediumTest
3   public class EditPostActivityTest
4   {
5       private SourceViewEditText editorTitleView;
6       private SourceViewEditText editorContentView;
7
8       @Rule
9       public IntentsTestRule<EditPostActivity> activityTestRule =
10              new IntentsTestRule<EditPostActivity>(
11                    EditPostActivity.class,
12                    false,
13                    false)
14              {
15                  @Override
16                  public void beforeActivityLaunched()
17                  {
18                      // Initialize the monitor before each test
19                      EventMonitor.getInstance().initialize();
20                  }
21
22                  @Override
23                  public void afterActivityLaunched()
24                  {
25                      // Add observables used by all tests
26                      EventMonitor.getInstance()
27                       .observe(EventBusObservable.mediaUploadEvents());
28                      getEditorViews();
29                      EventMonitor.getInstance()
30                       .observe(EventUtils.postChanges(editorTitleView,
```

```
                                editorContentView));

                // Add checks valid for all tests
                EventMonitor.getInstance().checkThat(
                        "Switched to HTML even if a media item is
                            uploading!",
                        anEventThat(isHtmlToggle())
                        .cannotHappenBetween(
                            anEventThat(isMediaUploadProgress()),
                            anEventThat(isMediaUploadOutcome())));

                EventMonitor.getInstance().checkThat(
                        "Post content changed after the upload
                            started!",
                        providedThat(
                            existsAnEventThat(isPostUploadStart()))
                        .then(
                            anEventThat(isPostChange())
                            .canHappenOnlyBefore(
                                anEventThat(isPostUploadStart()))));

                EventMonitor.getInstance().checkThat(
                        "Clicking on 'publish' didn't perform the
                            expected actions!",
                        atLeast(1).eventsWhereEach(
                            anyEvent(
                                isPostUploadStart(),
                                isToastDisplay()))
                        .mustHappenAfter(
                            anEventThat(
                             isMenuClick(R.id.menu_save_post))));

                EventMonitor.getInstance().checkThat(
                        "Media upload wasn't cancelled when editor
                            was removed!",
                        providedThat(
                            exist(
                                between(
                                    anEventThat(
                                      isMediaUploadProgress()),
                                    anEventThat(
                                      isMediaUploadOutcome())),
                                exactly(1))
                            .eventsWhereEach(
                                isFragmentLifecycleEvent(ON_DETACH)))
                        .then(
                            atLeast(1).eventsWhereEach(
                                isMediaUploadCancel())
                            .mustHappenAfter(
```

```
76                                  anEventThat(
77                                     isFragmentLifecycleEvent(
78                                          ON_DETACH)))));
79
80                  EventMonitor.getInstance().checkThat(
81                          "Race condition between publish and upload
                                media!",
82                          isNotSatisfied(
83                              exist(
84                                  between(
85                                      anEventThat(
86                                        isMediaUploadProgress()),
87                                      anEventThat(
88                                          anyEvent(
89                                              isMediaUploadOutcome(),
90                                              isMediaUploadProgress()))),
91                                  atLeast(1))
92                              .eventsWhereEach(isPostUploadStart())));
93
94                  EventMonitor.getInstance().checkThat(
95                          "Media upload progress updates are not
                                sent correctly!",
96                          allEventsWhereEach(isMediaUploadProgress())
97                          .areOrdered(new
                                Comparator<MediaUploadProgressEvent>()
98                          {
99                              @Override
100                             public int
                                    compare(MediaUploadProgressEvent
                                    e1, MediaUploadProgressEvent e2)
101                             {
102                                 return
                                        Float.compare(e1.getProgress(),
                                        e2.getProgress());
103                             }
104                         }));
105             }
106
107             @Override
108             public void afterActivityFinished()
109             {
110                 // At the end of each test, stop the verification
111                 EventMonitor.getInstance().stopVerification();
112             }
113         };
114
115 @Test
116 public void testUploadImage()
117 {
```

```
118         // Start activity and verification, and mock the image
                selection
119         launchActivity();
120         startVerification();
121         setupCameraResult();
122 
123         // Test actions
124         onView(withId(R.id.format_bar_button_html))
125                 .perform(click());
126 
127         onView(withId(R.id.sourceview_title))
128                 .perform(replaceText("My Title"));
129 
130         onView(withId(R.id.sourceview_content))
131                 .perform(replaceText("My Content."));
132 
133         onView(withId(R.id.format_bar_button_html))
134                 .perform(click());
135 
136         onView(withId(R.id.format_bar_button_media))
137                 .perform(click());
138 
139         onView(withText(R.string.select_photo))
140                 .perform(click());
141 
142         onView(withContentDescription(R.string.publish_post))
143             .perform(click());
144     }
145 
146     @Test
147     public void testPublishError()
148     {
149         // Start activity
150         launchActivity();
151         Context context = InstrumentationRegistry.getTargetContext();
152 
153         // Add checks specific for this test
154         EventMonitor.getInstance().checkThat(
155                 "The error toast wasn't displayed!",
156                 exactly(1).eventsWhereEach(
157                     isToastDisplay(equalTo(
158                         context.getString(
159                               R.string.error_publish_empty_post))))
160                 .mustHappenAfter(
161                     anEventThat(isMenuClick(R.id.menu_save_post))));
162 
163         // Start verification
164         startVerification();
165 
```

```
166         // Test actions
167         onView(withId(R.id.format_bar_button_html))
168                 .perform(click());
169
170         onView(withContentDescription(R.string.publish_post))
171                 .perform(click());
172     }
173
174
175     /*** OTHER TEST CASES ***/
176
177
178
179
180
181     /************** HELPER METHODS **************/
182
183     private void startVerification()
184     {
185       // Log events and throw AssertionError if a result fails
186         EventMonitor.getInstance().startVerification(
187                 EventMonitor.getLoggerEventsSubscriber(),
188                 EventMonitor.getAssertionErrorResultsSubscriber());
189     }
190
191     private void launchActivity()
192     {
193         /* Omitted for simplicity. Launches Activity with correct
                Intent parameters. */
194     }
195
196     private void setupCameraResult()
197     {
198         /* Omitted for simplicity. Mocks the "select image"
                operation using Espresso Intents library. */
199     }
200
201     private void getEditorViews()
202     {
203         /* Omitted for simplicity. Retrieves title and content views
                from the Editor Fragment contained in the Activity. */
204     }
205 }
```

Listing A.4: An example of custom event implementation for event-based testing: the HTML toggle event in the WordPress app. It can have any parameter (in this case just a boolean), provide several static methods to build Hamcrest matchers and define the `toString` method, whose output is used by the EventMonitor.

```
1  public class HtmlToggleEvent extends Event
2  {
3      private boolean active;
4
5      public HtmlToggleEvent(boolean active)
6     {
7          this.active = active;
8      }
9
10     public static Matcher<HtmlToggleEvent> isHtmlToggle()
11    {
12         return new FeatureMatcher<HtmlToggleEvent,
               Void>(anything(""), "is any HTML toggle event", "")
13       {
14             @Override
15             protected Void featureValueOf(final HtmlToggleEvent
                   actual)
16          {
17             return null;
18             }
19         };
20     }
21
22     public static Matcher<HtmlToggleEvent> isHtmlToggle(final
           boolean active)
23    {
24         return new FeatureMatcher<HtmlToggleEvent,
               Boolean>(equalTo(active), active ? "is an HTML mode
               activation" : "is an HTML mode deactivation", "")
25       {
26             @Override
27             protected Boolean featureValueOf(final HtmlToggleEvent
                   actual)
28          {
29             return actual.active;
30             }
31         };
32     }
33
34     @Override
35     public String toString()
36    {
37         return active ? "{HTML mode activated}" : "{HTML mode
               deactivated}";
```

```
38      }
39  }
```

Listing A.5: An example of custom Observable for event-based testing: generates the media-related events in the WordPress application. `EventBusObservable` just provides a static method to easily get the Observable, `EventBusSubsriber` is an abstract class created for reuse and `MediaUploadOnSubscribe` is the actual implementation of the Rx Observable that fires the events.

```
1   public class EventBusObservable
2   {
3       public static Observable<BusEvent> mediaUploadEvents()
4      {
5           return Observable.create(new
                MediaUploadOnSubscribe()).onBackpressureDrop();
6       }
7   }
8
9   abstract class EventBusSubscriber
10  {
11      EventBusSubscriber()
12     {
13          EventBus.getDefault().register(this);
14      }
15  }
16
17  class MediaUploadOnSubscribe implements
        Observable.OnSubscribe<BusEvent>
18  {
19      @Override
20      public void call(final Subscriber<? super BusEvent> subscriber)
21     {
22          new EventBusSubscriber()
23        {
24              public void
                    onEventMainThread(MediaEvents.MediaUploadSucceeded
                    event)
25           {
26                  if(!subscriber.isUnsubscribed())
27              {
28                      subscriber.onNext(new MediaUploadSuccessEvent());
29                  }
30              }
31
32              public void
                    onEventMainThread(MediaEvents.MediaUploadFailed
                    event)
```

```
33          {
34                  if(!subscriber.isUnsubscribed())
35              {
36                      subscriber.onNext(new MediaUploadFailureEvent());
37                  }
38              }
39
40              public void
                    onEventMainThread(MediaEvents.MediaUploadProgress
                    event)
41          {
42                  if(!subscriber.isUnsubscribed())
43              {
44                      subscriber.onNext(new
                            MediaUploadProgressEvent(event.mProgress));
45                  }
46              }
47          };
48      }
49  }
```

Listing A.6: An example of check implementation for event-based testing: the "can happen only between" of `AnEventThat`. The Subscriber is implemented as a finite state machine that handles the events in order, changing state and/or short-circuiting the check if we have a match.

```
1   public Check canHappenOnlyBetween(final AnEventThat eventBefore,
        final AnEventThat eventAfter)
2   {
3      return new Check(
4            "Every event that "+getMatcher()+" happens only between a
                 pair of events where the first
                 "+eventBefore.getMatcher()+" and the second
                 "+eventAfter.getMatcher(),
5
6            new CheckSubscriber()
7            {
8               private final static int OUTSIDE_PAIR = 0;
9               private final static int INSIDE_PAIR = 1;
10              private final static int FOUND_E1_OUTSIDE = 2;
11
12              private final State state = new State(OUTSIDE_PAIR);
13
14              private boolean foundAtLeastOneE1 = false;
15              private int eventsInCurrentPair = 0;
16
17              @Override
18              public void onNext(Event event)
```

```
19              {
20                 switch(state.getState())
21                 {
22                    case OUTSIDE_PAIR:
23
24                       if(getMatcher().matches(event))
25                       {
26                          state.setState(FOUND_E1_OUTSIDE);
27                          state.setEvents(event);
28                          endCheck();
29                       }
30
31                       else if(eventBefore.getMatcher().matches(event))
32                       {
33                          state.setState(INSIDE_PAIR);
34                       }
35
36                       break;
37
38                    case INSIDE_PAIR:
39
40                       if(eventAfter.getMatcher().matches(event))
41                       {
42                          state.setState(OUTSIDE_PAIR);
43                          eventsInCurrentPair = 0;
44                       }
45
46                       else if(getMatcher().matches(event))
47                       {
48                          foundAtLeastOneE1 = true;
49                          eventsInCurrentPair++;
50                       }
51
52                       break;
53                 }
54              }
55
56              @NonNull
57              @Override
58              public Result getFinalResult()
59              {
60                 Outcome outcome = null;
61                 String report = null;
62
63                 if(state.getState()==INSIDE_PAIR &&
                       eventsInCurrentPair<=0)
64                 {
65                    state.setState(OUTSIDE_PAIR);
66                 }
```

```
67
68              switch(state.getState())
69              {
70                 case OUTSIDE_PAIR:
71
72                    if(foundAtLeastOneE1)
73                    {
74                       outcome = Outcome.SUCCESS;
75                       report = "Every event that "+getMatcher()+"
                             was found inside a pair";
76                    }
77
78                    else
79                    {
80                       outcome = Outcome.WARNING;
81                       report = "No event that "+getMatcher()+" was
                             found in the sequence";
82                    }
83
84                    break;
85
86                 case INSIDE_PAIR:
87
88                    outcome = Outcome.FAILURE;
89                    report = "At the end of the stream,
                          "+eventsInCurrentPair+" events that
                          "+getMatcher()+" were found but no event
                          that "+eventAfter.getMatcher()+" was there
                          to close the pair";
90
91                    break;
92
93                 case FOUND_E1_OUTSIDE:
94
95                    outcome = Outcome.FAILURE;
96                    report = "Event "+state.getEvent(0)+" was found
                          outside a pair";
97
98                    break;
99              }
100
101             return new Result(outcome, report);
102          }
103       }
104    );
105 }
```

Listing A.7: Race condition that can be recognized by detectors like EventRacer or DEvA but cannot be expressed with a temporal assertion.

```
public class Activity2 extends AppCompatActivity
{
    private MyService service = null;

    @Override
    protected void onCreate(Bundle savedInstanceState)
    {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_2);

        Button button = (Button)
            findViewById(R.id.perform_operation);
        button.setOnClickListener(new View.OnClickListener()
        {
            @Override
            public void onClick(View view)
            {
            // The commented if-then-else sequence is one of the
                solutions to handle the race condition

                // if(service!=null)
                // {
                     int num = service.doSomeOperation();
                     Toast.makeText(Activity2.this, "Result: "+num,
                         Toast.LENGTH_SHORT).show();
                // }
                // else
                // {
                //   Toast.makeText(Activity2.this,
                //   "Wait for the service to start",
                //   Toast.LENGTH_SHORT).show();
                // }
            }
        });
    }

    @Override
    public void onStart()
    {
        super.onStart();

        if(service==null)
        {
            bindService(new Intent(this, MyService.class),
                connection, Context.BIND_AUTO_CREATE);
        }
    }
```

```
44
45      @Override
46      protected void onStop()
47      {
48          super.onStop();
49          if(service!=null)
50          {
51              unbindService(connection);
52          }
53      }
54
55      private ServiceConnection connection = new ServiceConnection()
56      {
57          @Override
58          public void onServiceConnected(ComponentName className,
                IBinder service)
59          {
60              MyService.LocalBinder binder = (MyService.LocalBinder)
                    service;
61              Activity2.this.service = binder.getService();
62          }
63
64          @Override
65          public void onServiceDisconnected(ComponentName arg0)
66          {
67              Activity2.this.service = null;
68          }
69      };
70  }
```

Listing A.8: Race condition that can be expressed with a temporal assertion but cannot be recognized by detectors like EventRacer or DEvA.

```
1   public class Activity3 extends AppCompatActivity
2   {
3       @Override
4       protected void onCreate(Bundle savedInstanceState)
5       {
6           super.onCreate(savedInstanceState);
7           setContentView(R.layout.activity_3);
8
9           TextView emailContent = (TextView)
                findViewById(R.id.email_content);
10        emailContent.addTextChangedListener(new TextWatcher()
11        {
12              // perform some operation when/before/after text changes
13          });
14
15          Button sendEmail = (Button) findViewById(R.id.send_email);
```

```
        sendEmail.setOnClickListener(new View.OnClickListener()
        {
            @Override
            public void onClick(View view)
            {
                // send email code
            }
        });
    }
}
```

Listing A.9: Race condition that can be both recognized by detectors like EventRacer or DEvA and expressed with a temporal assertion.

```
public class Activity4 extends AppCompatActivity implements
    OnMapReadyCallback
{
    private GoogleMap map;

    @Override
    protected void onCreate(Bundle savedInstanceState)
    {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_4);

        MapFragment mapFragment = (MapFragment)
            getFragmentManager().findFragmentById(R.id.map);
        mapFragment.getMapAsync(this);
    }

    @Override
    protected void onResume()
    {
        super.onResume();
        if(map!=null)
        {
            map.addMarker(new MarkerOptions()
                    .position(new LatLng(0, 0))
                    .title("MyMarker"));
        }
    }

    @Override
    public void onMapReady(GoogleMap map)
    {
        this.map = map;
    }
}
```

List of Figures

1 Activity Lifecycle 18
2 Fragment Lifecycle 20
3 Android Event Handling 23
4 Local and Instrumented Android Tests 26
5 Lifecycle Static Checks Implementation UML 44
6 Lifecycle Tests Implementation UML 51
7 Event-Based Testing Design UML 81
8 Event-Based Testing Implementation UML 86

References

[1] Statistics about smartphones and applications. http://www.statista.com/statistics/330695/number-of-smartphone-users-worldwide/, http://www.statista.com/statistics/276623/number-of-apps-available-in-leading-app-stores/.

[2] Michal Young and Mauro Pezze. Software Testing and Analysis: Process, Principles and Techniques. John Wiley & Sons, 2005.

[3] 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.

[4] More information about the Android OS. http://source.android.com/devices/tech/.

[5] JUnit. http://junit.org.

[6] Mockito. http://mockito.org/.

[7] Robolectric. http://robolectric.org/.

[8] Espresso. https://google.github.io/android-testing-support-library/docs/espresso/.

[9] UI Automator. https://developer.android.com/training/testing/ui-testing/uiautomator-testing.html.

[10] UI Exerciser Monkey. https://developer.android.com/studio/test/monkey.html.

[11] monkeyrunner. https://developer.android.com/studio/test/monkeyrunner/index.html.

[12] Christoffer Quist Adamsen, Gianluca Mezzetti, and Anders Møller. Systematic execution of android test suites in adverse conditions. In Proceedings of the 2015 International Symposium on Software Testing and Analysis, ISSTA 2015, pages 83–93, New York, NY, USA, 2015. ACM.

[13] 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.

[14] Pavol Bielik, Veselin Raychev, and Martin Vechev. Scalable race detection for android applications. SIGPLAN Not., 50(10):332–348, October 2015.

[15] Yongjian Hu, Iulian Neamtiu, and Arash Alavi. Automatically verifying and reproducing event-based races in android apps. In Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016, pages 377–388, New York, NY, USA, 2016. ACM.

[16] 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.

[17] Static Lifecycle Checks, Repository. `https://github.com/Simone3/LifecycleLintChecks`.

[18] Lint. `http://tools.android.com/tips/lint`.

[19] SaferMobile. InTheClear. `https://github.com/SaferMobile/InTheClear`.

[20] byteShaft. TrackBuddy. `https://github.com/byteShaft/TrackBuddy`.

[21] Dynamic Lifecycle Tests, Repository. `https://github.com/Simone3/DynamicLifecycleTesting`.

[22] Automattic, Inc. WordPress for Android. Google Play `https://play.google.com/store/apps/details?id=org.wordpress.android`, Source Code `https://github.com/wordpress-mobile/WordPress-Android`.

[23] Automattic, Inc. WordPress. `https://wordpress.com/`.

[24] WordPress, Test Repository - December 6, 2015 Version. `https://github.com/Simone3/WordPress-Android-2015-12-06`.

[25] WordPress, Test Repository - July 15, 2016 Version. `https://github.com/Simone3/WordPress-Android-2016-07-15`.

[26] Ian Pratt-Hartmann. Logics with counting and equivalence. In Proceedings of the Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS), CSL-LICS '14, pages 76:1–76:10, New York, NY, USA, 2014. ACM.

[27] Event-Based Testing, Repository. `https://github.com/Simone3/TemporalAssertionsTesting`.

[28] ReactiveX. http://reactivex.io/.

[29] RxJava. https://github.com/ReactiveX/RxJava.

[30] RxAndroid. https://github.com/ReactiveX/RxAndroid.

[31] RxBinding. https://github.com/JakeWharton/RxBinding.

[32] RxPreferences. https://github.com/f2prateek/rx-preferences.

[33] RxFileObserver. https://github.com/phajduk/RxFileObserver.

[34] StorIO. https://github.com/pushtorefresh/storio.

[35] ReactiveNetwork. https://github.com/pwittchen/ReactiveNetwork.

[36] Retrofit. http://square.github.io/retrofit/.

[37] RxBroadcast. https://github.com/cantrowitz/RxBroadcast.

[38] ReactiveLocation. https://github.com/mcharmas/Android-ReactiveLocation.

[39] ReactiveSensors. https://github.com/pwittchen/ReactiveSensors.

[40] RxPermissions. https://github.com/tbruyelle/RxPermissions.

[41] RxGoogleMaps. https://github.com/sdoward/RxGoogleMaps.

[42] RxWear. https://github.com/patloew/RxWear.

[43] RxLifecycle. https://github.com/trello/RxLifecycle.

[44] JavaHamcrest. http://hamcrest.org/JavaHamcrest/.

[45] Espresso Intents. https://google.github.io/android-testing-support-library/docs/espresso/intents/.

[46] ProGuard. http://proguard.sourceforge.net/.