A Runtime Verification Framework For Kotlin Mobile Applications

Master's thesis in Computer science and engineering

Denis Furian

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2020

Master's thesis 2020

A Runtime Verification Framework For Kotlin Mobile Applications

Denis Furian

UNIVERSITY OF GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2020

A Runtime Verification Framework For Kotlin Mobile Applications

Denis Furian

© Denis Furian, 2020.

Supervisor: Gerardo Schneider, Department of Computer Science and Engineering
Industrial Advisor: Boris Tiutin, Opera Software AB
External Advisors: Christian Colombo, University of Malta
Yliès Falcone, University Grenoble Alps
Examiner: Wolfgang Ahrendt, Department of Computer Science and Engineering

Master's Thesis 2020
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2020

A Runtime Verification Framework For Kotlin Mobile Applications
Denis Furian
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

The Kotlin programming language has recently been introduced to Android as the recommended language for development. We investigated whether we could use this language to improve the state of the art for Runtime Verification on mobile devices and focused on creating an API to monitor the execution of coroutines, one of the main Kotlin functionalities that are not featured in Java. This API should be employed by Android programmers to carry out concurrent tasks in a monitored environment and verify at runtime that the Kotlin guidelines and best practices for coroutines are being followed.
We identified a number of such guidelines and redefined them as properties to either monitor through Runtime Verification or enforce through Runtime Enforcement; we then tested them on an in-house Android app built using our API. In this report we present the API and the results of our tests concerning performance overhead and memory usage, as well as our ideas for future development.

Keywords: Runtime Verification, Runtime Enforcement, Android, Kotlin, Aspect-oriented Programming, Monitor, Coroutine, Concurrency, Structured Concurrency

Acknowledgements

I wish to express my gratitude towards my supervisor Gerardo Schneider, who has lent me his expertise and continued support throughout the project. I would like to thank Christian Colombo, from the University of Malta, and Yliès Falcone, from the University Grenoble Alps, for their invaluable support in directing my efforts in examining the state of the art for Runtime Verification and identifying properties. My thanks go also to Boris Tiutin and the rest of the Opera Software office in Gothenburg, without whom this project wouldn't have been possible. Finally, I want to thank Wolfgang Ahrendt, my examiner, for introducing me to the field of Runtime Verification and giving me the opportunity to write this thesis.

Denis Furian, Gothenburg, February 2020

Contents

List of Figures xii

List of Tables xiii

List of Listings xvii

1 Introduction 1
1.1 Problem Statement . . . 1
1.1.1 RV of Android . . . 2
1.2 Goals and Challenges . . . 3

2 Background and Related Work 5
2.1 Runtime Verification and its Application to Android . . . 5
2.1.1 Expressing Properties . . . 6
2.1.2 Monitoring the Execution . . . 7
2.1.2.1 Application-centric RV . . . 8
2.1.2.2 Device-centric RV . . . 9
2.1.3 Existing Tools . . . 10
2.1.4 Limits of the Current Situation . . . 11
2.2 The Kotlin Programming Language . . . 11
2.2.1 Non-Nullable Types . . . 11
2.2.1.1 Interoperability with Java and Platform Types . . . 12
2.2.2 Default Function Arguments . . . 13
2.2.3 Absence of Checked Exceptions . . . 13
2.2.4 Coroutines . . . 15
2.2.4.1 Types of Coroutines . . . 17
2.2.4.2 Termination of a Coroutine . . . 19
2.2.4.3 Coroutines in Android . . . 19
2.2.5 Problems and Limitations . . . 20

3 Prototype for a Monitoring Tool 21
3.1 Target Properties . . . 21
3.1.1 Property 1: DestroyedWithOwner . . . 21
3.1.2 Properties of tasks that return a value . . . 23
3.1.2.1 Property 2: NormalAsync . . . 23
3.1.2.2 Property 3: ExceptionalAsync and Property 4: NeedHandler . . . 23

3.1.3 Property 5: NoBlockUI and Property 6: UpdateUI 24
3.1.4 Property 7: ResumeIfNeeded 24
3.1.5 List of Monitors 25
3.2 First Approach: Hardcoding Monitors into the Application 26
3.2.1 Definition of tasks to execute on coroutines 27
3.2.1.1 Property 8: AlwaysOneJob 27
3.2.1.2 Property 9: SuccessWithJSON 28
3.2.2 Class implementation of monitors 29
3.2.3 Running the monitors in the `ViewModel` 30
3.2.3.1 Considerations 36
3.3 Second Approach: Using Annotations to Generate Compile-time Monitors 37

4 Implementation of an API for Monitoring Kotlin Coroutines 39
4.1 The Interface `MonitoredComponent` and its API 39
4.1.1 First Version of the API 39
4.1.1.1 Keeping Track of Tasks 41
4.1.1.2 Handling Exceptions 43
4.1.1.3 Remembering Failed Tasks 45
4.1.1.4 Referencing the Correct Task 47
4.1.2 Implementation of the Methods `launch` and `async` 49
4.1.3 The `MonitoredApplication` Class 53
4.1.4 The `MonitoredActivity` Class 54
4.1.5 The `MonitoredViewModel` Class 54
4.2 The Finalised API 57
4.2.1 The Methods `launch` and `async` 57
4.2.2 The New `MonitoredComponent` Interface 58
4.2.2.1 The New `MonitoredActivity` and `MonitoredViewModel` Classes 60

5 Experimental Validation 63
5.1 Testing the API on a Proof-of-concept App 63
5.1.1 Test results 64
5.2 Benchmarking the Application 67
5.2.1 Benchmarking Tool 68
5.2.2 Results 68
5.3 Memory Footprint 72

6 Conclusion 73
6.1 Considerations 73
6.2 Perspectives and Future Work 74

Bibliography 80

List of Figures

2.1 Example of a connected graph representing an execution. 6

2.2 Example of an aspect implemented using the AspectJ syntax: this syntax allows to define pointcuts and pieces of advice in a way that is similar to methods. 8

2.3 Example of an aspect implemented using AspectJ annotations: they can be used to mark methods of a standard Java class as pointcuts (with the `@Pointcut` annotation) or pieces of advice (e.g. by using the annotations `@Before`, `After` and `Around`). 9

2.4 The Kotlin compiler assigns Java objects with platform types unless the Java code uses specific annotations. The upper part of the figure contains Java code with definitions for the methods `getNullValue`, `getAnnotatedNullValue`, which is annotated with `@Nullable`, and `getAnnotatedNotNullValue`, which is annotated with `@NotNull`. The lower part of the figure showcases the return types assigned to each method by the Kotlin compiler. 15

2.5 The Java method `receiveSomething` is not notified about the Kotlin function `getFoo` throwing an `IOException`. 16

2.6 The `@Throws` annotation ensures that Java method `receiveSomething` will not compile unless the call to `getFoo` handles the possible exception. 17

2.7 By using `withContext` it is possible to switch context for the currently running coroutine without spawning a new one. The figure displays the source code on the upper half and the output on the lower half. Compare the first part of the output (before the first empty line) with the second: in the latter there are three different coroutines (marked as `SECOND COROUTINE#3`, `#4`, `#5`) whereas in the former there is only one (marked as `SECOND COROUTINE#2`), indicating that no new coroutine was created. 18

3.1 Graph of the lifecycle of an Android `Activity`, taken from the official Android documentation [63]. 22

3.2 Notification from the Android OS that the UI thread is blocked for the application aptly named “App blocking the UI thread”. 24

3.3 Suspend points marked on lines 55, 58, 61 and 63. 25

3.4 The image browsing app developed for testing hard-coded monitors. . 27

3.5 Simplified representation of the app's architecture: once the `buttonTags` UI element is tapped by the user, it triggers the `search` method which, in turn, launches the `getImages` method on a coroutine. This coroutine launches an asynchronous task to read bytes from the remote endpoint (in the `readText` method) and parses the result with the method `parseFlickrImageJson` to obtain one or more images. The images are used by the `updateResultLiveData` method to notify the activity, which receives the new data and displays in the method `loadNewData`. 28
3.6 Representation of the inner workings of the `Monitor` class described in Figure 3.7. The method `check` reads an input trigger and starts a series of internal operations that may or may not cause a change in the monitor's state. The method yields a state that is either an "OK state" or an "error state". 30
3.7 UML representation of the `State`, `Property` and `Monitor` classes. The `Trigger` class is a type alias for `String`. 31
3.8 Flow chart detailing the logic for the functions `doNext` and `afterResult` of the `MonitoredViewModel` class. 32
3.9 Flow chart for the function `maybeDoNext` of the `MonitoredViewModel` class. 33
3.10 Updated version of the architecture in Figure 3.5 with added information on instrumentation: the coroutine builder methods `launch` and `async`, displayed in a different colour, are wrapped inside the `maybeDoNext` and `afterResult` methods (shown respectively in Figures 3.9 and 3.8). Throughout the execution of each method inside the viewmodel, the internal state of the `Monitor` instance is updated with the number of search operations currently ongoing ("active searches ++" signifying an increment by 1 and "active searches −−" a decrement by 1). 35

4.1 Representation of the logic followed by the updated `launch` method for saving a `CoroutineExceptionHandler` inside the context. 48
4.2 New behaviour of the `launch` and `async` methods with regard to the `CoroutineContext`: the methods `beforeTask` and `beforeLaunch/Async` progressively update the context, leaving the user free to define *how* it should be updated. 58

5.1 Simplified call diagram for the proof-of-concept app. Each rounded rectangle represents a class and each rectangle inside it represents its methods, with the orange rectangles being the one employing coroutines and, therefore, that we were interested in monitoring. The arrows represent the call flow starting from the activity. Note that inside the `BrowsePicturesViewModel` class there is a second flow that ends in error and updates the UI accordingly. 64

List of Tables

3.1 Summary of the properties that we wanted to verify and which of the monitors defined in section 3.1.5 would carry out the necessary controls. 26
3.2 Updated version of Table 3.1 containing properties 8 and 9. 29

5.1 Technical specifications of both devices employed in our benchmarks. 68
5.2 Benchmark results for the `launch` method listing the fastest, slowest and average times for the "standard" version and the instrumented one, with the overheads in the rightmost columns. All values are in nanoseconds (ns) save for the last column. 69
5.3 Benchmark results for the `async` method listing the fastest, slowest and average times for the "standard" version and the instrumented one, with the overheads in the rightmost columns. All values are in nanoseconds (ns) save for the last column. 69
5.4 Benchmark results for the `async` method, called a thousand times in parallel. The table lists the fastest, slowest and average times for the "standard" version and the instrumented one, with the overheads in the rightmost columns. All values are in nanoseconds (ns) save for the last column. 69
5.5 Benchmark results for the `ViewModel` simulation listing the average time for the "standard" version and the instrumented one, with the overheads in the rightmost column. All values are in nanoseconds (ns). 71
5.6 Memory reserved by the `Application` instance, not instrumented (second column) and then instrumented (third and fourth columns). All values are in bytes. 72

List of Listings

1 Difference in declaration for nullable and non-nullable types in Kotlin: the former are marked with a question mark (`?`). 11

2 The Kotlin compiler uses smart casts on nullable objects after they have been checked for null. 12

3 Short-circuit evaluation applied to nullable objects: when a nullable variable is asserted non-null in a skippable expression, the compiler tries to skip the assertion. 13

4 Platform types can not be used explicitly and the user is not allowed to assign them to a variable. 14

5 Example of function with default arguments: calling the function `foo` without specifying any value for its arguments `a`, `b` and `c` will populate them with the indicated values (or evaluated expressions). 14

6 Implementations of the **AlwaysOneJob** and **SuccessWithJSON** properties expressed using the `Property` class. 34

7 Implementation of the `search` method: the code on lines 5-8 is only used to manually inform the viewmodel of a new search operation being initiated, with invocations of the methods `readState` and `updateState` to respectively read and change the internal state of the monitor. . . 37

8 “Ideal” implementation of the `search` method where the boilerplate code featured in Listing 7 is hidden. 37

9 Our version of the coroutine builder methods: the signature is thus the same as the standard one for `CoroutineScope.launch` [56] and `CoroutineScope.async` [57] with the sole addition of the `MonitoredComponent` as an argument. 40

10 Declarations for the internal data structures in the `MonitoredComponent` interface. 40

11 Formal definition of the methods with which the `MonitoredComponent` interface handled the initialisation, start and termination of tasks. . . 42

12 Implementation of our version of the launch method in order to save references to the launched tasks. As can be seen, our version of launch was actually a “wrapper” function calling the default launch (line 8) after performing some control operations. The call on line 12 was inside a finally block, meaning it would be executed whenever the task was cancelled. The task might actually be cancelled before the call on line 15, which would result in the launchedTasks array *not* holding any references to the task: for this reason we used the key argument in the method onComplete rather than the task itself. The same applied to our version of the async method, which performed the same operations and then call the default version of the same method. 43
13 Implementation of our version of the launch method in order to ensure that one CoroutineExceptionHandler instance should always be in the context. The assignment at lines 6-7 ensured that the handler we would use be either the one already in the context or, should no such object exist in the context, the defaultHandler data structure defined inside the component. By employing the expression context[CoroutineExceptionHandler.Key] we queried the coroutine context for its current handler (receiving a null if no handler was defined). It should be noted that the lines 11-15 add technically nothing to the function: ignoring CancellationExceptions is standard coroutine behaviour since they are just a notice of cancellation, and any Exception thrown inside a block of code is automatically rethrown (as stated in 2.2.3). 44
14 Our implementation of the launch method in order to both keep record of running tasks and ensure the presence of a CoroutineExceptionHandler instance in the coroutine context. 45
15 Our implementation of the async method in order to both keep record of running tasks and rethrow any exceptions that should be raised during the task's execution. 46
16 Declaration of the WrongDispatcherException class. 47
17 The extension function that we implemented as a fake accessor for the Throwable class. It returns true when called on an instance of CalledFromWrongThreadException, allowing us to distinguish objects of this type by matching their class name. 49
18 Complete implementation for the launch method. 51
19 Complete implementation for the async method. 52
20 Declaration and initialisation of the data structure we used to keep record of the recommended coroutine dispatchers. 53
21 Code of the MonitoredActivity, subclass of Activity that implements the MonitoredComponent interface. 55
22 Implementation of the extension function getMonitoredViewModel that provides any instance of MonitoredActivity with a matching viewmodel and initialises it automatically. 55

23 Code of the MonitoredViewModel, subclass of ViewModel that implements the MonitoredComponent interface. 56
24 Signatures of the new methods replacing the old API. 59
25 Implementation of the MonitoredScope object and its two extension functions as short-hands for invoking the method launch or async on the component's coroutineScope field. 61
26 Snippets of the code referenced by the recommendedDispatchers data structure upon inspection. 66
27 An example of a coroutine that bypasses our monitoring 66
28 An example of nested coroutine builders, each using a randomised dispatcher and lacking an exception handler. 67
29 Implementation of the test for simulating the non-monitored execution. 70
30 Implementation of the test for simulating the monitored execution. . 71

1

Introduction

Throughout the last decade smartphones have become a huge part of everyday life, growing from portable telephones to complex computers with which users can not only communicate but also access their bank account, monitor their health, play video games, find their car in a parking area and more.
All sorts of applications can be found in virtual stores for Android and iOS and, in turn, there is a growing need to guarantee that they pose no threat to the user's smartphone by means of a faulty implementation (e.g. an app consuming more battery than needed) or malicious operations (e.g. an app leaking personal data).
Runtime Verification [1, 2, 3, 4, 5] is one of the possible methods of monitoring smartphone applications and reporting unwanted behaviour: it consists of a series of checks that verify at runtime whether the target program complies with a set of user-defined properties. In the case of devices using Google's Android operating system, there have been several attempts to employ RV (e.g. [30, 6, 7]). The Android operating system is however constantly changing, with Android 10 [8] being the newest version at the time of writing, and monitoring tools have to stay up to date with the more recent releases.
With this project we want to observe the current state of Runtime Verification on Android and determine how it can be pushed further. Aiming to improve the state of the art, we focused our efforts on the newly supported **Kotlin language** [9]: more precisely, we wished to investigate how its unique features over Java could be monitored.
The project was developed in cooperation with **Opera Software AB** [19], a Norwegian software company primarily known for its desktop web browser Opera, its mobile web browser Opera Mini and its recent shift to mobile development with fintech applications for Android (e.g. [20, 21, 22]). Opera Software, or just "Opera", was founded in 1995 and to this day has headquarters in Oslo, Norway, and offices in Poland (Wrocław), Sweden (Linköping, Göteborg, Stockholm) and China (Beijing), focusing on browsers for most of its existence and promoting Web standards through participation in the W3C.

1.1 Problem Statement

Runtime Verification is one of several methods used for ensuring the correctness of programs. It involves the monitoring of a running program or system while observing that it complies with one or more given properties during its execution; a report is then generated detailing whether any property has been violated.

Runtime Verification, or “RV”, can be split in two major tasks:

1. **defining the properties** that the system must comply with: they will be expressed with a statement that will be more or less complex depending on their nature;
2. **generating the monitors** that will verify these properties: this is often platform-dependent as it involves modifying a program or wrapping it in an observable layer.

When monitors are not part of the target system but, rather, a second program, a need arises to connect the monitors to the system in such a way they can effectively observe the execution. The operation of detecting traces of events from the target program and relaying them to a monitoring system is known as **instrumentation** [4] and can be carried out on the target program's binary code or directly on its source code.

1.1.1 RV of Android

Since the introduction of the **Android operating system**, there has been a growing interest in applying RV to it. This has spawned several research efforts throughout the past years (e.g. [13, 30, 7]). For most of Android's lifetime, its main language for development has been Java; naturally, most attempts to introduce RV to Android have seen the employment of Java-specific tools (e.g. [12]) to instrument applications.

As of 2017, the status of preferred development language for Android has shifted to **Kotlin**, a more recent programming language that can interoperate with Java while also introducing a set of unique features.

There are a number of differences between Java and Kotlin; the most relevant for our research are:

- **checked exceptions**: compared to Java, Kotlin does not force the programmer to handle expected exceptional behaviour; when an API can be expected to throw an exception, the Kotlin compiler will raise no error or warning. Behind this choice is a push to favour special return types instead of exceptions [37], when possible;
- **coroutines**: introduced as a concept in the 1960s [16], they are lightweight tasks that can run concurrently inside threads; they do not exist in Java and Kotlin introduced them as an alternative to callbacks [17] in Android development, especially for dealing with lifecycle-aware structures.

We found that most of the recommended usage practices of Kotlin can be monitored by means of static analysis.

On the other hand, coroutines are quite different. In Kotlin they are intended for use with structured concurrency [42], which means entry and exit points must be made clear and all tasks are either completed or cancelled before the end of the execution. This approach, coupled with Kotlin's choice to not enforce exception handling, means that a whole coroutine scope will be terminated should any task crash [18].

Coroutines are designed with the goal of running either attached to a single thread or moving from one thread to another. They are implemented as stackless, which

reduces the overhead caused by saving the current state between each thread jump, and can move between main and secondary threads depending on the task they need to carry out.
We deemed these aspects of coroutines to be interesting to investigate in our project to further RV on Android.

1.2 Goals and Challenges

In this thesis project we aimed to define a RV framework for Android applications with the three main tasks of:

1. expressing a range of system properties as wide as possible;
2. making the monitoring system translate said properties into meaningful Android code without loss of information;
3. defining a feedback to be returned by the monitoring system in the form of either a corrective action or a report produced post-execution.

With this framework we aimed to address the research questions expressed below.

What are the meaningful properties of Kotlin coroutines that we can verify by employing RV?

We decided to focus our efforts in coroutines, which are well-suited for Runtime Verification and Enforcement. We could inspect official guidelines, design patterns and recommended practices to identify examples of good app behaviour and transform these examples into properties to observe or enforce.

Can we use the Kotlin language to monitor properties of the more specific Kotlin features?

We set out to use the Kotlin language to either adapt an existing tool for RV of Android or create our own. This process would ideally allow us to maximise our ability to monitor Kotlin properties.

How can we push further the current situation for RV of Android applications by using Kotlin?

We would evaluate our framework to determine what possible use cases it could suit best and how it could advance the state of the art for RV of Android. Our evaluation would include testing the way that properties are observed as well as measuring additional impacts of our solution to the target app's execution; for example, performance overheads caused by the monitoring effort.

Our research question would be addressed by using the **Design Research** methodology [14]: we would formulate hypotheses on the problem at hand and, subsequently, design a solution for each hypothesis and implement it into an artefact. This, once tested, would turn out to either validate or invalidate the starting hypothesis.
We would follow this procedure iteratively for each of our research questions.

This project can be split in two major phases: a theoretical one for observing the state of RV on the Android platform and a more practical one for contributing to the current situation.
The **first phase**, which covered the first half of the project, consisted in us assessing the situation on multiple fronts:

- investigating the state of Android RV gave us an overview of what had already been tried and what had not yet been accomplished;
- examining the unique features of the Kotlin language, as well as its similarities and differences with Java, helped us single out what could be interesting for our research.

At the conclusion of the first phase we could determine that not only was the situation of application-centric RV quite advanced for Java Android applications but it also was for Kotlin ones. Android applications run on Dalvik bytecode, which is converted from JVM bytecode: since Kotlin on Android compiles to JVM, any monitor that works on a Java application will therefore also work on a Kotlin one.
Our investigation presented us with a choice to focus on either delving deeper into RV with little concern for development language or shifting our main focus to Kotlin. We opted for the latter and chose coroutines as our main interest with the following motivations:

- **multithreading** is an important topic in Android development: multiple apps and services are always running in either the foreground or the background and often multiple threads are used within the same application; several APIs have been developed to improve the control a programmer has over a multithreaded environment and coroutines contribute in their own unique way;
- we found several limitations with the current **debugging tools** for coroutines, especially on Android where the debug agent is not supported due to a lack of libraries;
- while it is possible to implement coroutines in Java [51], they are so far only a **language feature** in Kotlin and, as such, are expected to be expanded in the future and receive continued support by JetBrains.

The goal of our project turned therefore from "improving the state of the art of Android RV" to a more specific "expanding Android RV to coroutines for both monitoring a concurrent environment and strengthening the available debugging tools", with our attention turning from the final user of an Android application to the developer.
After establishing the final scope of our project, we moved onto the **second phase**, which consisted of actually implementing our solution.

2

Background and Related Work

For this project we focused our efforts in improving the state of the art for Runtime Verification on the Android mobile platform. There have been several research efforts on RV of Android with variations in its approach to both defining properties and monitoring them.

From its launch in 2008 to early 2019, Android applications could almost exclusively be written in Java [24] and therefore most implementations of monitoring tools for Android direct their efforts towards translating properties into Java code.

This chapter will cover both RV and the Kotlin language, listing some of their features and their relevance to Android: section 2.1 describes the main research topics involved with RV for Android and gives an overview of the current situation and how it can be advanced further; section 2.2 lists the main features of Kotlin and the differences between this language and Java, focusing towards the end on Kotlin coroutines.

2.1 Runtime Verification and its Application to Android

The nature and the goal of a program can determine what properties should be considered for monitoring. Some programs handle personal data and it is therefore critical to ensure this data is not leaked; some other programs carry out money transactions and they need to be robust enough to handle failures with minimal impact.

Properties can also involve the platform on which a program is run and the impact it has on the platform's resources. Properties can be expressed at different levels of abstraction as well as combine platform-specific issues with general concerns such as security: for example, limiting data access of a company app running on an BYOD (or "Bring Your Own Device", i.e. using one's personally owned devices rather than officially provided ones) smartphone of an employee [10].

While properties can be as simple as "user's location is never tracked", the more interesting cases are the ones where properties are defined by a context, given as either the current state of the execution ("user's location is never tracked *while the device is charging*") or a trace, i.e. a sequence of actions that matches the current execution ("user's location is only tracked *after the user has taken a picture*").

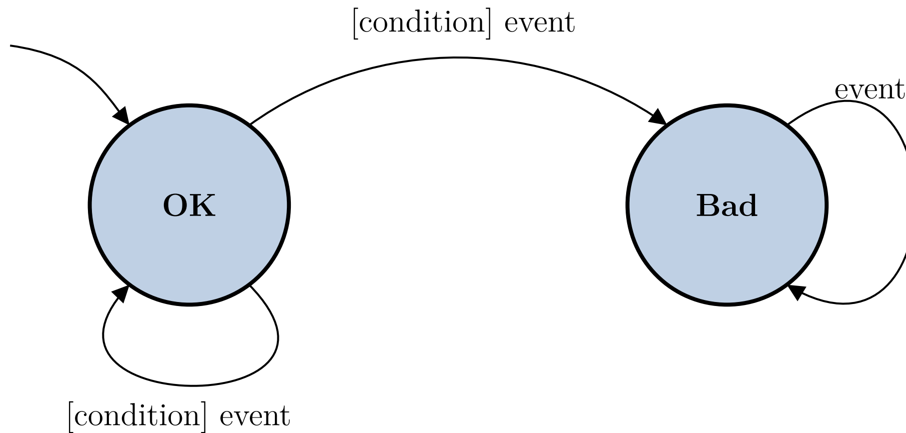

Figure 2.1: Example of a connected graph representing an execution.

2.1.1 Expressing Properties

The more complex properties are, the more they require an expressive language to represent them. While some simple "if A, then B" properties can most likely be written in a straightforward way, others (such as the examples given in the previous paragraph) require a more articulate formula. Previous works on this topic have used both **logic-based approaches** (mostly based on variations of linear temporal logic [25]) and **finite-state automata** [30] to represent stateful properties and traces.

Automata are stateful control mechanisms designed to read an input, evaluate it and advance their state depending on the outcome of the evaluation. A **finite-state automaton** can only enter one from a limited number of states; the automaton starts in what is called the "initial" state and determines the next one by means of a transition function: this is a function that considers the combination of current state and latest input to decide what state to advance to.
When used for properties, automata can be represented as connected graphs with variables to keep track of the current evaluation of the property:

- each node represents a number of execution states grouped in the same equivalence class: usually there are one or more "bad" states which are associated with the breach of a property;
- each edge contains a condition and an event, where the condition is a statement that is checked when the event is detected.

The initial state is usually a "good" state and, with the violation of a property, the automaton enters one of the "bad" states.
An example of such a graph can be seen in Figure 2.1. The execution is intentionally left simple for ease of understanding, with only one kind of event, one condition and no variables. The graph can be read as:

- execution begins in the "OK" state;
- whenever an event is triggered but the condition is not met, the "OK" state is maintained;
- if an event is triggered and the condition is met, the state shifts to "Bad";
- once in "Bad" state, any event will maintain the "Bad" state regardless of the

condition being met or not.

2.1.2 Monitoring the Execution

Instrumentation is the phase that follows the definition of properties. As mentioned in section 1.1, it consists of connecting the target application to a monitoring system that can detect traces of events.
The paradigm of **Aspect-Oriented Programming** is focused on separating "cross-cutting concerns", which are anything that influences multiple functionalities in a program, e.g. error reporting or security. Monitors fit quite well in this category, so AOP is naturally seen as an ideal way of implementing them [11].
Aspect-Oriented Programming as a paradigm is well-suited for the purpose of adding monitors to a program in a cross-cutting way, which means weaving code into several places regardless of their functionality. As mentioned earlier in this document, the most common way of implementing monitors for Android application is by employing tools that translate properties to Java monitoring code. The Java implementation of the AOP paradigm is called **AspectJ** [12] and employs a dedicated compiler to "read" directives and "inject", or "weave", them into the target program. By writing monitoring code through an aspect-oriented approach, AspectJ can weave monitors into applications that are already compiled, making potentially any application in the Google Play Store able to be controlled.
AspectJ works by reading one or more "aspects": they are implementation units composed of pieces of "advice". Their structure can be explained as follows:

- a **join point** is a well-specified point in the execution flow of the program, e.g. a method call, an instantiation or a value being returned;
- a **pointcut** identifies one or more join points through a filter on several search parameters such as package, class, method, arguments or annotations, to name a few;
- an **advice** is a block of code that can be executed before, after or around a pointcut, as well as when the join point matched by the pointcut returns a value or when it throws an exception.

An **aspect** includes one or more pieces of advice as well as inter-type declarations of objects that can store data or observe the behaviour of specific instances.
Figures 2.2 and 2.3 showcase different ways of defining an aspect: though AspectJ features two types of syntax for writing advice code, the general idea is that a piece of advice executes a given block of code at a given time before or after an event takes place, if that event has been identified by means of a pointcut. The examples show different definitions for the same advice code:

1. two pointcuts called `callSetInt` and `callGetInt` identify the join points where the methods `setInt(..)` and `getInt(..)` for the class `foo.bar.Baz` are called;
2. a piece of advice is executed before each of the pointcuts:
    - one is called before the `callSetInt` pointcut is executed and increases an internal counter;
    - the other is called before the `callGetInt` pointcut is executed and decreases the counter.

```
2
3 public aspect MyOtherAspect {
4     int counter = 0;
5
6     pointcut callSetInt(): (call(* foo.bar.Baz.setInt(..)));
7
8     before(): callSetInt(){
9         counter++;
10    }
11
12    pointcut callGetInt(): (call(* foo.bar.Baz.getInt(..)));
13
14    before(): callSetInt(){
15        counter--;
16    }
17 }
```

Figure 2.2: Example of an aspect implemented using the AspectJ syntax: this syntax allows to define pointcuts and pieces of advice in a way that is similar to methods.

After instrumenting an application with the above example, the advice code will run just before the above methods are called on an instance of the `foo.bar.Baz` class.

2.1.2.1 Application-centric RV

A widely used approach in RV has been the monitoring of an application of choice: the user chooses a set of properties to verify for one app, then expresses those properties by means of some RV tool that translates them to code, which is used to instrument the app. Most Android tools employ a combination of logic and automata to define properties and a technology based on AspectJ for weaving monitors into the target application.
The AspectJ compiler instruments the compiled .class files by weaving aspect code into them [23]. This way, it is possible to recompile any Android app to add monitoring code:

1. the APK (Android Package) file of an app is processed through a Dalvik decompiler such as, for example, dex2jar [15], yielding a JAR (Java Archive) containing the bytecode files of the application;
2. the JAR is recompiled by `ajc` (the AspectJ compiler) and the monitors are added as pieces of advice;
3. the instrumented JAR is recompiled into an APK that is installed over the previous version.

This process proves to be a powerful tool for weaving monitors into the bytecode of any Android application; so long as a verifiable property can be somehow converted to a Java algorithm, it can be woven into any app.

While AspectJ is designed with Java in mind, it can also be made to work with Android applications written (partially or entirely) in Kotlin. Kotlin is a cross-platform language (as is explained in section 2.2) and, when employed for Android development, it compiles to JVM bytecode. AspectJ works by reading compiled Java classes, so it can read compiled Kotlin classes in the same fashion.
With a certain degree of knowledge of how the Kotlin compiler translates Kotlin

```
 2
 3 import org.aspectj.lang.JoinPoint;
 4 import org.aspectj.lang.annotation.Aspect;
 5 import org.aspectj.lang.annotation.Before;
 6 import org.aspectj.lang.annotation.Pointcut;
 7
 8 @Aspect
 9 public class MyAspect {
10     private int counter = 0;
11
12     @Pointcut("call(* foo.bar.Baz.setInt(..))")
13     public void callSetInt() {}
14
15     @Before("callSetInt()")
16     public void beforeCallSetInt() {
17         counter++;
18     }
19
20     @Pointcut("call(* foo.bar.Baz.getInt(..))")
21     public void callGetInt() {}
22
23     @Before("callGetInt()")
24     public void beforeCallGetInt() {
25         counter--;
26     }
27 }
```

Figure 2.3: Example of an aspect implemented using AspectJ annotations: they can be used to mark methods of a standard Java class as pointcuts (with the `@Pointcut` annotation) or pieces of advice (e.g. by using the annotations `@Before`, `After` and `Around`).

code to JVM bytecode it is possible to instrument a Kotlin app with AspectJ: the programmer only needs to declare pointcuts using whatever Java code the Kotlin compiler generates.

In the field of app-centric RV there is a push to both:

- define properties in a way that is as flexible as possible to translate them to code;
- improve the reach of AspectJ monitors to cover not just the app but its libraries as well.

2.1.2.2 Device-centric RV

One of the limitations of the application-centric approach described above is that property monitoring (or enforcement, where applicable) is restricted to the application level: the device, as a whole, is left largely unaffected. This means that properties involving the device as a whole can not be verified: if, for example, a parent wants to limit their child's time spent on the Internet, they will need to act directly on the child's device rather than on single applications.
There are generally two approaches to running runtime verification on a system with multiple applications running on it:

- having a single, central monitor watch over the applications, receiving constant

updates about what is happening in each of them (which has been carried out on Android in the past [27]);

- having several monitors, one for each relevant application, communicate with one another to coordinate the verification process (as seen in DMaC [28]).

Both solutions come with advantages and drawbacks [26]. The first approach, known as "orchestration", allows a centralised structure to keep track of all events happening in the device, but it also means that it is involved in a lot of traffic, with high risk of bottlenecks; the second approach, despite being more distributed and not requiring one monitor to do the work of many, comes at the cost of needing intercommunication between the several monitors (and possibly shared memory).

2.1.3 Existing Tools

Throughout the existence of Android, several tools for RV have been developed, both for device-centric verification (as mentioned previously) and for the application-oriented approach.

RV-Droid [6] is one such tool, and one of the first to be developed. It enabled Runtime Verification (as well as Enforcement) by means of allowing the device owner to instrument a target application among the ones installed: the user would choose the target app and select an amount of possible properties to verify chosen from a range. RV-Droid would then translate those properties into monitors, either locally or by means of a remote server, and subsequently weave the monitors into the app.

The monitors would finally be integrated into the target application by the process described in section 2.1.2.1 but by means of a modified version of AspectJ.

This kind of verification had the advantage of being easily set up and relatively unintrusive: the user would only have to download an app from Google's Play Store, without the need for rooting their device or performing other, heavier modifications. Unfortunately it came at the same time with the limitation of not being able to instrument code from the libraries used by the target app: libraries provided by the Android runtime, as well as the ones downloaded dynamically as the app is executed, elude the static weaving process.

Another, more recent Android tool for RV is **ADRENALIN-RV** [7], which addresses these issues by:

- instrumenting the core libraries provided by the Android OS during the device boot operations;
- constantly weaving monitors into dynamically-generated library code by means of a remote server exchanging data with the tool installed onto the device.

This approach allows an extensive coverage when verifying a single application but it also comes at the cost of requiring a heavy customisation to the target device: the device needs to be injected with monitors during its boot phase and then needs to be constantly connected to the remote server for the continuous instrumentation process.

```
val a: Foo = null// yields a compilation error
val b: Foo? = null// correct
```

Listing 1: Difference in declaration for nullable and non-nullable types in Kotlin: the former are marked with a question mark (?).

2.1.4 Limits of the Current Situation

While ADRENALIN-RV may have pushed the effectiveness of monitors, the field of RV is always striving to achieve a more powerful language to define properties, which is the “other” topic of interest of this field.
Furthermore, there is a growing interest in monitoring the intercommunication between processes [29], allowing the verification of properties that either span multiple applications or involve one app exploiting another app’s resources and permissions to perform actions otherwise not allowed.

2.2 The Kotlin Programming Language

Kotlin is a programming language developed by JetBrains in 2011. Since October 2017 it has been officially supported for Android development alongside Java. As of 2019 it has become the preferred language by Google.
One of Kotlin’s main features is interoperability with Java: this allows any Kotlin program to execute Java code and employ Java libraries. Being a cross-platform language, Kotlin can compile to JVM, JavaScript or LLVM; on the Android platform it targets the JVM.
For the purposes of this project, we considered compilation from Kotlin to JVM as our main focus. There are, however, some peculiarities coming from the cross-platform nature of the language; they will be addressed later as they could impact mobile development.
Kotlin has several differences with Java, some of which are described in the sections below, complete with how Kotlin’s exclusive features are translated to JVM code by the compiler. To do this we compiled Kotlin classes and then decompiled them through the Procyon [41] decompile tool. This process gave us a translation of our Kotlin code to Java which we used for observing how the Kotlin compiler converts the unique features of the former language to the latter.

2.2.1 Non-Nullable Types

One of the main features setting apart Kotlin from Java is the introduction of **nullable and non-nullable types**. Every Kotlin type is by default not nullable and can only be treated as nullable with the `?` symbol, as shown in Listing 1.
The purpose behind this is to avoid null references [31] as much as possible. The Kotlin compiler can also recognise cases where a nullable object is not null: in these cases it will relax its checks, as shown in Listing 2.
There is still the possibility of triggering a `NullPointerException`: this happens when a nullable value is forcefully read by using the `!!` operator. This way the

```
val foo: Bar? = someValue()
if(foo != null) {
    /* foo is not null */
}
```

```
val foo: Bar? = someValue()
if(foo == null) return
// after this point,
// foo is not null
```

Listing 2: The Kotlin compiler uses smart casts on nullable objects after they have been checked for null.

programmer can still use nullable types but has a better way of avoiding null references.

It's worth noting that some non-nullable types are already present in Java: they are primitive types and not classes, therefore they are never instantiated. When translating Kotlin to JVM code, the compiler assigns variables with a primitive type if:

- they are explicitly typed as non-nullable:
  - Kotlin's non-nullable `Int` and `Boolean` types compile to Java's `int` and `boolean` primitive types;
  - Kotlin's nullable `Int?` and `Boolean?` types compile to Java's `Integer` and `Boolean` classes;
- they are declared as nullable (e.g. `Int?`) but are initialised to a non-null value and are never assigned a null value throughout their lifecycle;
- their inferred Kotlin type is non-nullable: declaring a variable `val a = kotlin.random.Random.nextInt()` causes `a` to evaluate to `final int`.

Compiled boolean expressions make use of short-circuit evaluation, which means that an expression is not evaluated if the result is inconsequential (e.g. in the expression `true || (foo && bar)`, evaluation of `foo && bar` is skipped); this feature is used by the compiler to avoid assertions when possible (see Listing 3 for an example).

It is possible to check whether some `lateinit var foo` field has been initialised by calling the `isInitialized` property on its reference: `::foo.isInitialized` will return `false` if the field has no value. In the JVM this translates to checking whether the backing field is `null`.

2.2.1.1 Interoperability with Java and Platform Types

By design Kotlin can be used together with Java: interoperability between the two is a selling point of the language.

When using Java classes in Kotlin code, all null checks are relaxed; instead of forcing the programmer to check for null references, Kotlin marks any Java objects as "platform types" [50], that may or may not be null. The reason for this is ensuring that "safety guarantees for (Java objects in Kotlin) are the same as in Java".

Kotlin does not allow to use platform types explicitly (see Listing 4), so the programmer is forced to treat them as either nullable or non-nullable. It is possible to coerce Kotlin into treating a Java object as either nullable or not by using the `@Nullable` and `@NotNull` annotations (from the library `org.jetbrains.annotations`) in the Java source code as shown in section 2.4.

```
// Kotlin code:
fun foo(a: Boolean, b: Boolean?){
    var c = a && b!!
}
```

```
// Compiled Java code:
public final void foo(final boolean a, @Nullable final Boolean b) {
    boolean b2 = false;
    Label_0023: {
        if (a) {
            if (b == null) {
                Intrinsics.throwNpe();
            }
            if (b) {
                b2 = true;
                break Label_0023;
            }
        }
        b2 = false;
    }
    final boolean c = b2;
}
```

Listing 3: Short-circuit evaluation applied to nullable objects: when a nullable variable is asserted non-null in a skippable expression, the compiler tries to skip the assertion.

2.2.2 Default Function Arguments

It is possible to assign a default value to one or more arguments in a Kotlin function, as seen in Listing 5. If a function with default parameters is always called with explicit values for all of its arguments (e.g. `foo(0, 3, 2)`), then the compiler translates it to a "normal" Java method; otherwise it splits it into:

- the "normal" method without default arguments;
- an additional "default" method (e.g. `foo$default`) with the original arguments, plus an additional one that indicates which arguments had an explicit value. This method initialises all arguments that were not provided by the caller and then calls the "normal" method using these values.

2.2.3 Absence of Checked Exceptions

Exception handling is treated differently in Kotlin than it is in Java: the most important difference is that Kotlin does not use Java's **checked exceptions**. While a Java method may be forced to enclose one or more lines of code in a `try-catch` block (or add the `throws` keyword to a method's signature), Kotlin imposes no such requirement.

The use of checked exceptions is a divisive topic as there are both arguments in

```
val valueFromJava = javaObject.getFooValue()// typed as Foo!
val foo: Foo? = valueFromJava// allowed
val bar: Foo = valueFromJava// allowed but, when translated
                            // to bytecode, adds an assertion
                            // that valueFromJava is not null
val baz: Foo!// compile error: Unexpected token
```

Listing 4: Platform types can not be used explicitly and the user is not allowed to assign them to a variable.

```
fun foo(a: Int = 1, b: Int = a + 2, c: Int? = b) = c ?: b
```

Listing 5: Example of function with default arguments: calling the function `foo` without specifying any value for its arguments `a`, `b` and `c` will populate them with the indicated values (or evaluated expressions).

favour [34, 35] and against [32, 33] it. The absence of this feature in Kotlin means that no method is forced to handle exceptions and therefore any exception not caught right away will be automatically re-thrown to the calling method.
The documentation provided by JetBrains, however, discourages programmers from catching and throwing exceptions [37] and recommends instead the use of **special return types**.
These types are "special" in that they wrap their values with additional information in case something has gone wrong. Typical special return types are Maybe and Either:

- the **Maybe** (or **Option**) type contains either a value or "Nothing": before an instance is used in an expression, the programmer must check that it contains an actual value;
- the **Either** type is a more complex version of that: instead of containing *maybe* a value, it contains one of two values called "left" and "right". This type is usually employed to carry error information: a "right" value is mnemonically associated with the expected result of an expression, while a "left" value is usually an error trace or other information describing a failure.

The Kotlin documentation suggests using special return types defined in the external library Arrow: `Option` [39] and `Either` [38] as mentioned above, plus a `Try` that behaves more similarly to a traditional try-catch [40]. The rationale behind this is most likely to discourage the use of exceptions as special return values.
Despite JetBrains' encouragement to use special return types instead of Exceptions, Kotlin does not come with any such types by default, forcing the user to either make their own custom ones or importing external libraries. The Android framework, due to its Java origin, makes use of exceptions and therefore Kotlin applications for Android are forced to either adopt exceptions or implement wrappers that simulate the aforementioned types.
The compiler raises an error whenever a null check is skipped and the use of `lateinit` can already guarantee the use of a value without having to initialise it to `null`. Of course, this alternative is not always possible since `lateinit` may not be

```
import org.jetbrains.annotations.NotNull;
import org.jetbrains.annotations.Nullable;

public class JavaClass<T> {
    public static JavaClass getNullValue() {
        return null;
    }

    @Nullable
    public static JavaClass getAnnotatedNullValue() {
        return null;
    }

    @NotNull
    public static <T> JavaClass<T> getAnnotatedNotNullValue() {
        return new JavaClass<>( o: null);
    }
```

```
TestWithNull.kt
fun main() {
    val maybeNull : JavaClass<(raw) Any!>!  = JavaClass.getNullValue()
    val definitivelyNull : JavaClass<(raw) Any!>?  = JavaClass.getAnnotatedNullValue()
    val notNullOfPlatformType : JavaClass<Any!>  = JavaClass.getAnnotatedNotNullValue<Any>()
    val notNullOfNullableType : JavaClass<Any?>  = JavaClass.getAnnotatedNotNullValue<Any?>()
```

Figure 2.4: The Kotlin compiler assigns Java objects with platform types unless the Java code uses specific annotations. The upper part of the figure contains Java code with definitions for the methods `getNullValue`, `getAnnotatedNullValue`, which is annotated with `@Nullable`, and `getAnnotatedNotNullValue`, which is annotated with `@NotNull`. The lower part of the figure showcases the return types assigned to each method by the Kotlin compiler.

used on primitive types such as `Int`, `Boolean` and others.
Kotlin's interoperability with Java gives Java the ability to make calls to Kotlin code: This means that a Java method may invoke a Kotlin function that throws an unchecked exception. The compiler raises no errors in such cases (see Figure 2.5), which can prove a problem on the Java side. As a workaround for this, the Kotlin function throwing the exception can use the annotation `@Throws` to specify which exception types it might throw; in this case, the compiler will raise an error when an exception is not handled on the Java side (see Figure 2.6).
The opposite statement also holds true, as Kotlin code can invoke methods of Java classes. There is however no way to translate `throws` declarations from a Java method to its Kotlin caller, which means that a piece of Kotlin code invoking a Java method that may throw a certain exception may not be required to catch that exception.

2.2.4 Coroutines

Coroutines are an alternative to threads for implementing asynchronous programming: they focus on concurrency rather than parallelism, are generally lightweight and require none of the typical structures for mutual exclusion. In Kotlin they are intended for use with structured concurrency [42], which means entry and exit points must be made clear and all tasks are either completed or cancelled before the end of the execution.

Figure 2.5: The Java method `receiveSomething` is not notified about the Kotlin function `getFoo` throwing an `IOException`.

A single thread can run several coroutines: they follow a pattern of *suspend/resume* where they can be suspended at any time, their state is saved and then restored whenever they resume. A coroutine can also **suspend on one thread and resume on another** after transferring its state.
There are four predefined types of `CoroutineDispatchers` that are used to determine where a coroutine is going to run:

- the `Default` one is meant for "heavy" computations (e.g. sorting or parsing data) and executes outside of the main thread;
- the `Main` one is the thread where the application is running: in Android it is also known as "UI thread";
- the `IO` one is ideal for blocking operations such as network or database access;
- the `Unconfined` one can execute in any thread (or thread pool) and can switch from one thread to another at every resume. Its use is discouraged due to the lack of control of where a given coroutine will end up running.

An additional option is available for dispatching coroutines on user-defined thread pools, using functions like `newSingleThreadContext`; threads are expensive resources, so this option should be used sparingly.
Despite its name, the `Default` dispatcher is not actually the "default" choice: whenever a new coroutine is started, its scope will be the same as that of the caller. In other words, the `Main` thread will always be used unless otherwise specified. The function `withContext(..)` can be used to change the dispatcher of a coroutine as

Figure 2.6: The `@Throws` annotation ensures that Java method `receiveSomething` will not compile unless the call to `getFoo` handles the possible exception.

shown in Figure 2.7.

The lightweight nature of coroutines and their ability to be moved almost effortlessly between threads can be a concern when it comes to debugging: some developer blogs have, for example, detailed some problems with breakpoints being ignored [47] when a coroutine jumps between threads. The coroutine library does provide some tools to assist the debugging process:

- by enabling “debug mode” [43] any call to `Thread.currentThread().name` will yield the name for both the active thread and the current coroutine;
- the debug agent is a tool that keeps a record of all running coroutines; it does however require an API that is not provided by the Android runtime [44], making it not work in that context.

2.2.4.1 Types of Coroutines

A coroutine can be executed in multiple ways and this comes with heavily different use-cases.

- A standard job will simply execute whatever instructions are inside it.
- An `async` coroutine is expected to eventually yield some value: this is delivered as a `Deferred` value as soon as the coroutine is created. The caller can then `await` the end of the computation. If the coroutine is cancelled before it can return a value, however, it will throw an exception. Because of Kotlin’s lack

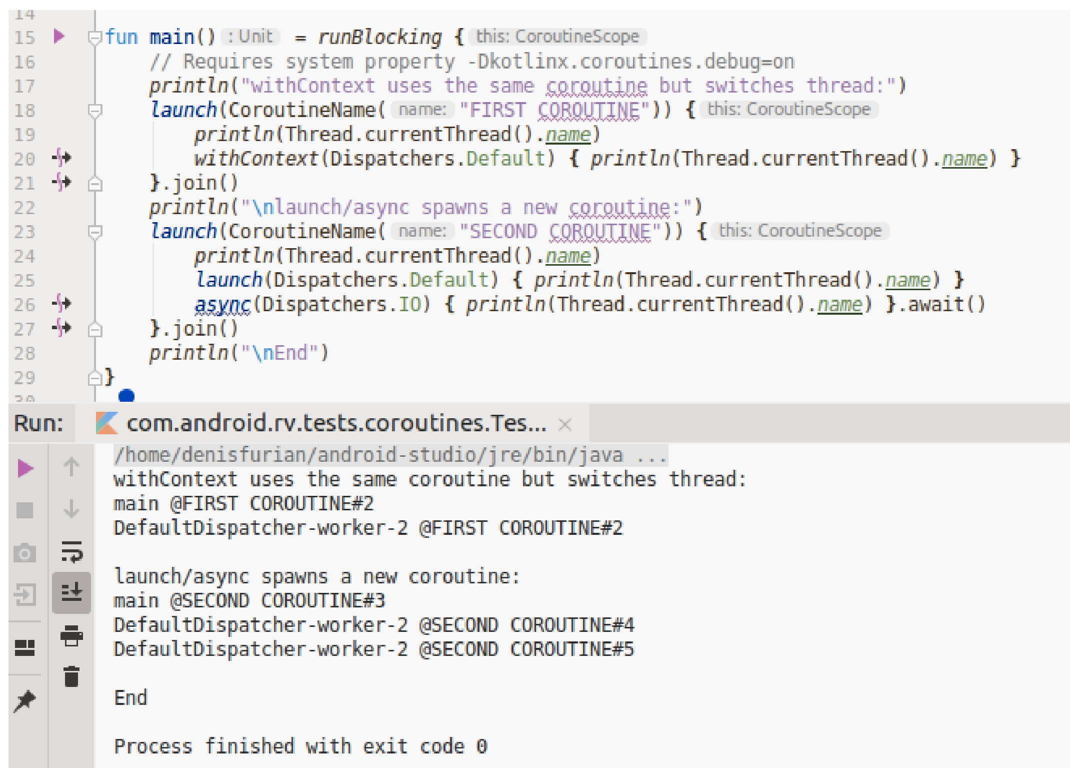

Figure 2.7: By using `withContext` it is possible to switch context for the currently running coroutine without spawning a new one. The figure displays the source code on the upper half and the output on the lower half. Compare the first part of the output (before the first empty line) with the second: in the latter there are three different coroutines (marked as `SECOND COROUTINE#3`, `#4`, `#5`) whereas in the former there is only one (marked as `SECOND COROUTINE#2`), indicating that no new coroutine was created.

of checked exceptions, the compiler will issue no warnings when `await` is used.

- A `produce` coroutine will return not a value but, rather, a stream of values to be delivered to a `ReceiveChannel`. This channel can be listened to in order to retrieve messages.
- An `actor` coroutine is similar to `produce` with the main difference being that it consumes values as opposed to sending them: it reads from a `SendChannel` that can be delivered messages to.

In the specific case of channels, Kotlin provides a `select` expression that allows to:

- listen to multiple `ReceiveChannels` and process the first message received from one of them;
- keep multiple `SendChannels` open and deliver a message to the first one available (i.e. the first channel that is neither closed or busy processing another message).

The current implementation of channels and the related builder functions, `actor` and `produce`, are still experimental and subject to change. Furthermore, some functionalities are still not completely working at the moment.

A similar entity to coroutines are **flows**: they are asynchronous streams of elements,

which means they deliver one value at a time as opposed to returning a single value at the end (e.g. `async`) or sending values through a channel like `produce`.
Flows are implemented as “cold” streams: their job is executed only when the `Flow`’s `collect` method is invoked: this makes them different from coroutines. By employing a buffer (which is, as of now, still an experimental feature) it is possible for a flow to keep emitting values even when the caller consumes them at a slow rate.
For the purposes of this project, we decided to forego experimental features such as channels and buffered flows. Experimental libraries can be subjected to changes in their implementation: it would therefore be risky to base our research on their current state. It would however be a good idea to keep an eye on their situation and possible updates.

2.2.4.2 Termination of a Coroutine

At any time the execution of a coroutine can be cancelled by invoking the `CoroutineScope`’s `cancel` method: this causes a `CancellationException` to be sent to that coroutine. This kind of exception is not treated as a “real” exception as much as a prompt to terminate. Any coroutine receiving this exception will quickly execute the following steps:

1. execute any code it might find inside a `finally` block;
2. recursively cancel all of its children (by forwarding the same `CancellationException` to them;
3. terminate.

Depending on the type of coroutine some additional events will occur:

- an `async` coroutine will pass the `CancellationException` instead of its expected return value; as mentioned earlier this behaviour will cause the corresponding `await` function to receive the exception;
- coroutines bound to channels (i.e. `produce` and `actor`) will close their channel: this causes any attempt to communicate to that channel to yield a `null` value.

Coroutines will also terminate if any other kind of exception is thrown. Since coroutines follow the paradigm of structured concurrency, cancellation in this case is propagated both downstream and upstream: this means that both the children *and the parent* of the failed coroutine will terminate. There is one way to prevent the cancellation from propagating upstream: the failing coroutine must be spawned by a `SupervisorJob`. In this case, the parent will not be affected by the failure and will simply receive the exception that caused the failure.
It is worth noting that catching an exception in a `try-catch` block will not prevent termination. It will, however, allow for a quick handling such as, for example, logging. Any code inside a `finally` block will be executed.

2.2.4.3 Coroutines in Android

Coroutines are well-ingrained in the MVVM (Model-View-ViewModel) architecture used by Android [54] and are specifically meant to be used for long-running tasks or CPU-intensive computations launched by the ViewModel [45]. The Ktx library provides the `ViewModel` class with its own `viewModelScope` that is automatically

destroyed whenever the ViewModel itself is cleared: this ensures that all coroutines running in the `viewModelScope` will be safely terminated.
The Ktx library provides coroutine support for other life-cycle components, too [46]. The `LifecycleScope` is a state-aware scope that calls coroutines when the enclosing life-cycle object is created, started or resumed; specifically it suspends a coroutine until the object reaches the intended state.

2.2.5 Problems and Limitations

In the previous section we detailed some features that, when combined, we perceive as problems of the Kotlin language:

- the absence of checked exceptions, combined with the way coroutines handle exceptions, can potentially result in some exceptions such as `CancellationException` not being handled;
- the recommendation to forego exceptions as special return types, combined with `CancellationException`s being used as special return types, sets the behaviour of coroutines apart from the rest of the language;

There are some more issues with how Kotlin handles coroutines: these are a side effect of the compilation to native code. While this is not a restriction of Kotlin/JVM, it does impact how the same block of code can be translated to JVM or, for example, iOS. The main problem comes with threading rules defined in LLVM: data can be either mutable or shared and, if shared, it must be frozen: it can never be "unfrozen" and is therefore completely immutable [48]. As a direct consequence, all coroutines in Kotlin/Native code must be defined in the same thread if they are to communicate with each other. JetBrains is aware of this issue but has no estimation for when it will be fixed [49].
A possible consequence of this problem is that multi-platform apps, created in Kotlin for both Android and iOS and compiled respectively to JVM and LLVM, will either have runtime crashes or not work with coroutines at all. This is obviously an edge case but it is worth mentioning.

3

Prototype for a Monitoring Tool

In this chapter we go through the properties that we deemed interesting to monitor on Kotlin coroutines: section 3.1 contains the description for each property as well as a table containing a summary of all of them. The following sections detail our first approaches in translating these properties to a monitoring system in Kotlin. Section 3.2 details our first approach using monitors hardcoded to the architectural components of an app, as well as our definition of several app-specific properties. Section 3.3 describes our second approach using annotations.

3.1 Target Properties

We had already considered some properties in the planning phase, and we integrated them with other behaviours that we wanted to verify (or enforce, in some cases). The result was a list of four scenarios that will be described in this section.
For convenience we decided to give each property a number and a name: these are also used as titles for the sections describing each property.

3.1.1 Property 1: DestroyedWithOwner

Each activity in an Android application has its own lifecycle and can be destroyed and recreated multiple times: specifically, an `Activity` is destroyed when the OS is low on memory and needs to save space, when the screen is rotated (unless a specific configuration setting is employed [62]) or, quite simply, whenever its `finish` method is invoked.
Whenever the same activity is recreated, a new object is instantiated and initialised. For this reason, any object that can "survive" an activity should either keep no references to it, or be destroyed together with it.
This applies not just to `Activity` but also to `LifecycleOwner` and its subclasses as well as other outliers such as, for instance, `ViewModel`. We use the term "lifecycle component" to refer to any Kotlin (or Java) class fitting this description. In short, **lifecycle components should not be leaked**.
In the case of coroutines, they execute a given block of code which may or may not contain references to a lifecyle component. It would be wasteful if a background thread were to download a large amount of bytes to display an image on an activity that was destroyed minutes before, with an obviously negative impact on performances (for the currently running app, the device battery and possibly the user's mobile data contract). It would be even worse if a new instance of this task were to

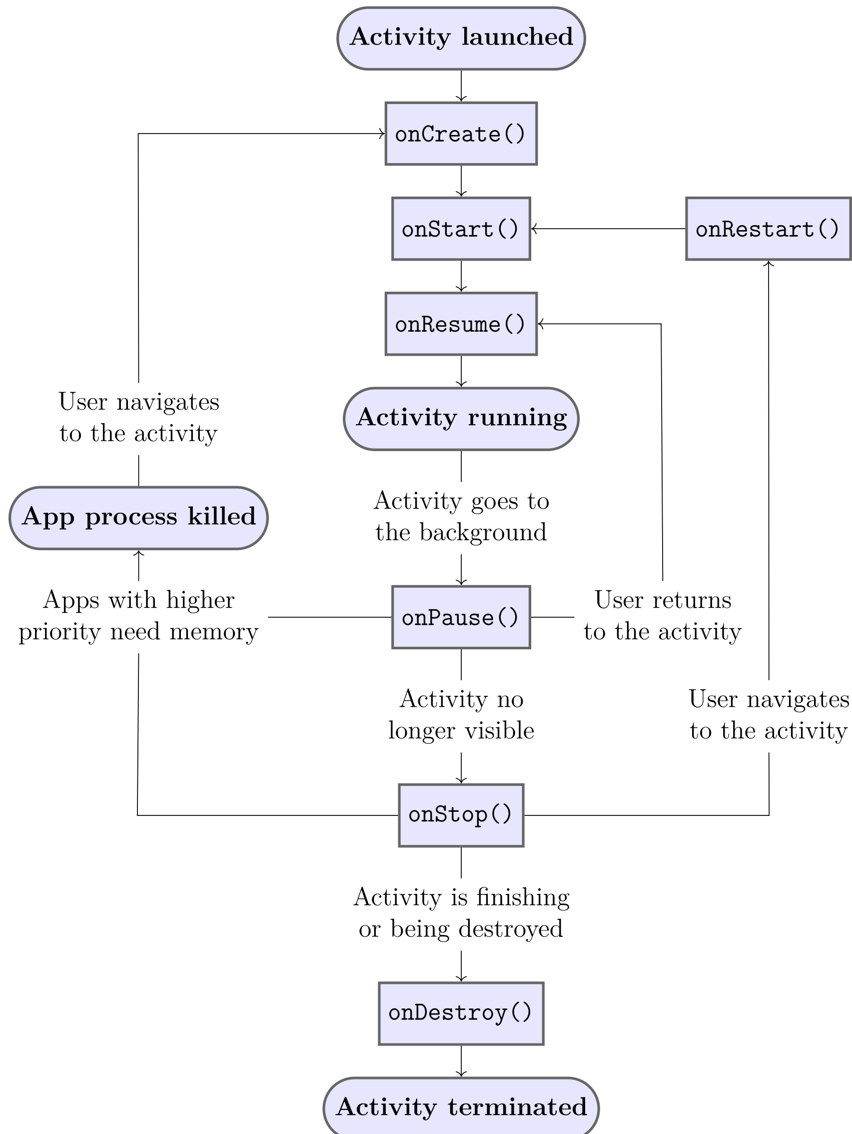

Figure 3.1: Graph of the lifecycle of an Android `Activity`, taken from the official Android documentation [63].

be launched every time the related activity was created.

Because of this, we wanted to ensure that all coroutines started in the scope of a given lifecycle component would be terminated together with that component. This behaviour is compliant with and, in fact, recommended by, the definition of structured concurrency used by Kotlin [42].

The destruction of a lifecycle component is not something that can be inferred after an examination of the code: it can be triggered by factors unrelated to the target application, as mentioned earlier and detailed in 3.1. Static analysis can verify whether a task contains references to e.g. instances of the `Activity` or `ViewModel` class but, in order to verify that the task is terminated together with its related component, we need to employ Runtime Verification.

3.1.2 Properties of tasks that return a value

As mentioned in section 2.2.3, Kotlin discourages the use of exceptions as return types and recommends instead that the user define their own (or use predefined ones such as the `Option` type defined in the Λrrow library [39]). Unfortunately, with coroutines this is not always possible since exceptions are closely tied to their intercommunication: the `CancellationException` class is used as a notification of termination. For this we identified three possible use cases:

1. a task is executed without any failures and returns a given value (which will be `Unit` for `launch`ed operations);
2. a task is cancelled at some point during its execution and therefore cannot return any value;
3. a task fails at some point during its execution and throws a given exception

Scenarios 1 and 3 can easily be associated with “ideal” and “exceptional” behaviour, respectively; the second case is, however, unclear: can this be counted as an “exceptional” case? Different types of coroutines have different treatments for this scenario: those started with the `launch` command simply ignore any `CancellationExceptions`, while those started with the `async` command will rethrow such an exception when the method `await` tries to retrieve their return value.

This behaviour can be best monitored while the app is running, which makes properties of tasks returning a value a good case for the employment of RV. We decided to examine the two clearer scenarios, which are 1 and 3, in order to express them as properties that we could monitor with our tool.

3.1.2.1 Property 2: NormalAsync

If a given block of code is executed without any failures, it will yield a certain return value depending on the type of coroutine on which it was running:

- `launch`ed tasks will yield `Unit`;
- `async` tasks will yield a `Deferred<T>` value, i.e. a “future” result that eventually evaluates to a value of type `T`.

3.1.2.2 Property 3: ExceptionalAsync and Property 4: NeedHandler

If a given block of code fails during its execution, it usually throws a `Throwable` subclass detailing the kind of failure it triggered and, possibly, the line of code that caused it.

As mentioned earlier in this document, any exception being thrown inside a, `async` coroutine (except for `CancellationExceptions`) will “taint” the current `CoroutineContext` and mark it for termination. Meanwhile, `launch` coroutines will rethrow the exception to their parent task all the way until the top level, at which point the `CoroutineExceptionHandler` inside the context is given the task to “handle” the failure [60].

We decided therefore to monitor that, in both cases, any exception is either rethrown or wrapped inside another `Throwable` object that is, in turn, thrown. This behaviour

was called **ExceptionalAsync** for `async`, in opposition to the previous one, and **NeedHandler** for `launch`.

3.1.3 Property 5: NoBlockUI and Property 6: UpdateUI

On a general note, Android apps are recommended to leave the main execution thread as lightweight as possible and avoid blocking it [59] because it would both cause a poor user experience and risk triggering an OS alert as seen in Figure 3.2.
There are some scenarios that are, however, downright forbidden and they can be summarised as "using the wrong thread" for certain operations:

- when a I/O operation is executed on the UI thread, the Android runtime launches a `NetworkOnMainThreadException`;
- likewise, when a background thread tries to access UI elements, the Android runtime throws a `CalledFromWrongThreadException`.

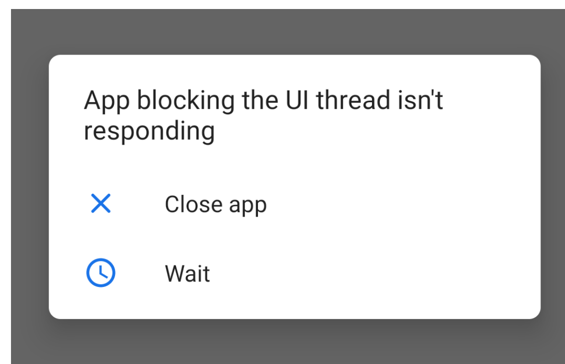

Figure 3.2: Notification from the Android OS that the UI thread is blocked for the application aptly named "App blocking the UI thread".

This behaviour is clearly considered harmful and we decided that it was worth monitoring in coroutines, since one of their biggest features is exactly the ability to "jump" from one thread to another, making it difficult to establish through static analysis where a task may be running at a given time. Using the scenario described above, we identified the properties:

- "NoBlockUI", according to which an I/O operation is always executed in a background thread;
- "UpdateUI" which states the opposite, i.e. that an update of the UI needs to be carried out in the main thread.

3.1.4 Property 7: ResumeIfNeeded

Under the hood coroutines are a sequence of callbacks that are suspended at one or more points in their execution. These **suspend points** can be traced back to any invocation of a `suspend` function inside a coroutine code block.
The code inside a coroutine is executed until a suspend point is reached: here the coroutine returns a special value to "warn" its dispatcher that its execution is not finished. Later, the dispatcher checks whether the coroutine is suspended, complete

Figure 3.3: Suspend points marked on lines 55, 58, 61 and 63.

or cancelled and, in the first case, it resumes the coroutine; the execution will start from right after the last suspend point.
In cases where a computation takes a large amount of time to complete, however, there might not be a chance for the dispatcher to check for cancellation [61]. A good practice is generally to check the flag `isActive` which returns `false` whenever the current coroutine is not supposed to execute anymore; there is also an `ensureActive` method that throws a `CancellationException` unless the `isActive` flag is `true`. These checks are executed at runtime and, therefore, we felt it would be appropriate to ensure at runtime that a task is only completed if necessary.

3.1.5 List of Monitors

From the scenarios described above we could isolate four events that corresponded to the same amount of Kotlin instructions:

- **component.destruction()** refers to the destruction of some "component": our assumption is that the component is lifecycle-aware and, as such, its method `onDestroy` is called as the component is on its way to termination;
- **launch(context, task)** corresponds to an invocation of the `launch` method with the arguments:
  - **context**, a representation of a `CoroutineContext`, i.e. a map that may contain a `CoroutineDispatcher` and a `CoroutineExceptionHandler`, among other things;
  - **task**, a representation of a block of Kotlin code;
- **async(context, task)** represents an invocation of the `async` method with the same argument constraints as described for **launch**;
- **await(deferredTask)** represents the invocation of the `await` method with an argument **deferredTask** of generic type `Deferred<T>`, where `T` can be any Kotlin type; since only the `async` method returns values of type `Deferred`, this event implies that a task had previously been started by means if `async`.

Our monitors were therefore defined as follows:

- one monitor would detect the **launch(context, task)** event and verify the **NeedHandler** property;

Table 3.1: Summary of the properties that we wanted to verify and which of the monitors defined in section 3.1.5 would carry out the necessary controls.

Property	**Monitors**			
	component.destruction()	**launch(context, task)**	**async(context, task)**	**await(deferredTask)**
DestroyedWithOwner	Yes	No	No	No
NormalAsync	No	No	No	Yes
ExceptionalAsync	No	No	No	Yes
NeedHandler	No	Yes	No	No
NoBlockUI	No	Yes	Yes	No
UpdateUI	No	Yes	Yes	No
ResumeIfNeeded	No	Yes	Yes	No

- one monitor would detect the **async(context, task)** event;
- one monitor would detect the **await(deferredTask)** event and verify the properties **NormalAsync** and **ExceptionalAsync**;
- every instance of a lifecycle-aware class (e.g. `ViewModel`) would contain a monitor to detect that object's **this.destruction()** event and verify the **DestroyedWithOwner** property.

Properties that were not tied to a specific type of coroutine, such as **NoBlockUI**, **UpdateUI** and **ResumeIfNeeded** would be verified by monitors for both builder functions, i.e. **launch(context, task)** and **async(context, task)**.
Each of our monitor should be able to intercept actions and decide, based on a detected event, whether an action was compliant with the "correct" behaviour or not. An action disrupting our desired behaviour could be reported or even cancelled, depending on the approach we would follow.
A summary of the properties and monitors defined so far can be found on Table 3.1.

3.2 First Approach: Hardcoding Monitors into the Application

As a first way of implementing monitors we opted for a rough, small-scale approach: we created a rudimentary app with both network activity and background computations.
Our app asks the user to submit a tag and searches images containing that tag

Figure 3.4: The image browsing app developed for testing hard-coded monitors.

on the image hosting website Flickr [52]. This operation uses the Flickr "search" feed [53] to receive the metadata for up to twenty user-submitted images in JSON format. As soon as the reading operation is completed, the app parses the response and extracts the URL for each image, which is then loaded and displayed on screen as shown in Figure 3.4.

3.2.1 Definition of tasks to execute on coroutines

The operations of reading from Flickr and parsing the JSON response are carried out with coroutines. We tried to implement a hard-coded monitor for the previously mentioned properties. All monitors were intended to activate an error state upon detecting a violation of either property.
We also identified two more properties specific for this application, which are presented below.

3.2.1.1 Property 8: AlwaysOneJob

We wanted the app to only carry out one search operation at a time. Since the operation is started whenever the user taps a button on the screen, an undefined

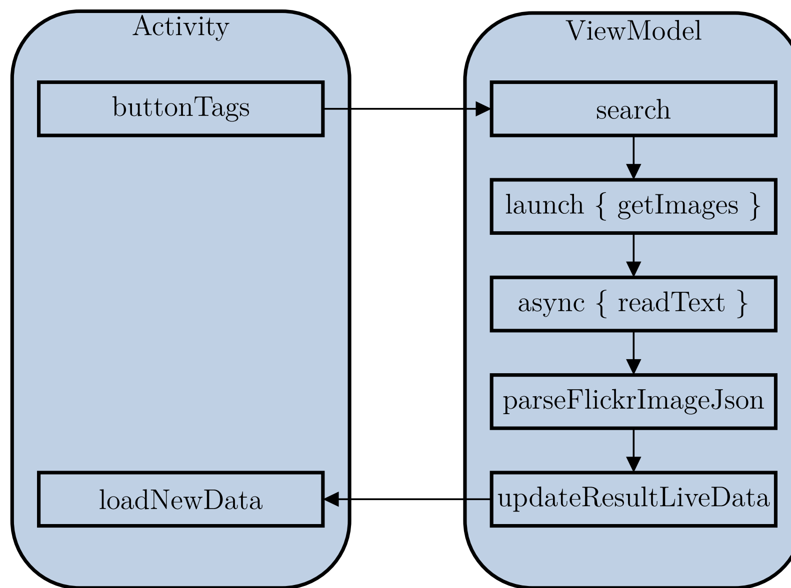

Figure 3.5: Simplified representation of the app's architecture: once the `buttonTags` UI element is tapped by the user, it triggers the `search` method which, in turn, launches the `getImages` method on a coroutine. This coroutine launches an asynchronous task to read bytes from the remote endpoint (in the `readText` method) and parses the result with the method `parseFlickrImageJson` to obtain one or more images. The images are used by the `updateResultLiveData` method to notify the activity, which receives the new data and displays in the method `loadNewData`.

amount of tasks might be launched in a matter of seconds and all images retrieved through the Flickr feed would be competing for access to the UI.
We decided to avoid this scenario by **limiting the highest amount of active search operations to one**: if the user were to tap the button while a search was ongoing, nothing would happen.
This property should be enforced at the moment of starting a search operation, so we chose to monitor with property when detecting a **launch(context, task)** event: as can be seen in Figure 3.5, the webservice is indeed invoked inside the method `launch`.

3.2.1.2 Property 9: SuccessWithJSON

The Flickr feed is supposed to return a list of up to 20 images, expressed in JSON format. Unforeseen problems with the endpoint (such as e.g. errors on the remote server) might, however, happen, and this would result in the application receiving an invalid JSON response.
This was added to the possible "exceptional" scenarios but, since it didn't entail an exception being thrown, we felt it wouldn't fit the **ExceptionalAsync** property: the parsing operation was executed outside of the `async` method invocation (see Figure 3.5). We therefore rationalised that the `async` method should, instead, *return* a **string containing a valid JSON**. For this reason we had the property monitored at each occurrence of the event **await(deferredTask)**, the same event that triggers

Table 3.2: Updated version of Table 3.1 containing properties 8 and 9.

Property	**Monitors**			
	component.destruction()	**launch(context, task)**	**async(context, task)**	**await(deferredTask)**
DestroyedWithOwner	Yes	No	No	No
NormalAsync	No	No	No	Yes
ExceptionalAsync	No	No	No	Yes
NeedHandler	No	Yes	No	No
NoBlockUI	No	Yes	Yes	No
UpdateUI	No	Yes	Yes	No
ResumeIfNeeded	No	Yes	Yes	No
AlwaysOneJob	No	Yes	No	No
SuccessWithJSON	No	No	No	Yes

the monitoring of **ExceptionalAsync**.

Table 3.2 shows a summary of the properties to verify in this first approach, as well as monitors carrying out the necessary controls for each property.

3.2.2 Class implementation of monitors

Trying to translate our notation for properties from paper to code, we used a notation similar to the automaton approach described in section 2.1.1. We defined the following classes and data structures:

- a **trigger** is a `String` that we send to the monitor to notify an event taking place;
- a **condition** is a function that reads the current state and outputs a `Boolean`: `true` when the condition is met and `false` otherwise;
- an **action** is a function that reads the current state and outputs a new state with updated values.

The **state** itself is defined as a map of entries where the keys are in `String` format; some notable keys are `state` to distinguish between "starting", "OK" and "error" states and `result` to carry the result of the last computation, as well as `error` to carry the error raised in the last unsuccessful operation. The state is updated by the monitor's **action** function but can also be changed directly by modifying the entries for the aforementioned keys, or adding new ones.
Finally, the **monitor** itself is a class containing the state and a list of **properties**,

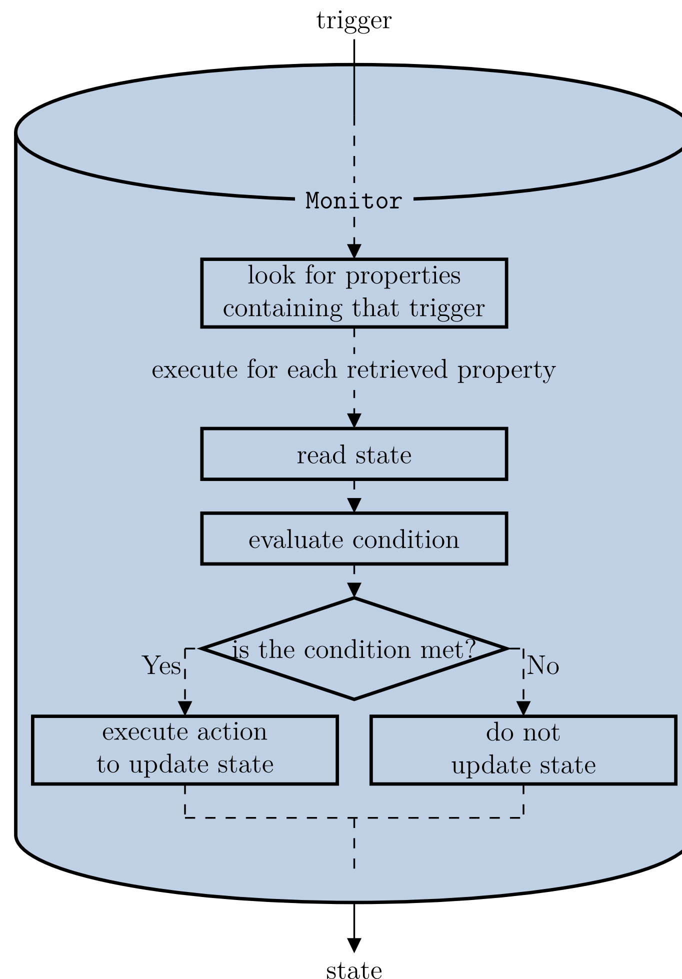

Figure 3.6: Representation of the inner workings of the `Monitor` class described in Figure 3.7. The method `check` reads an input trigger and starts a series of internal operations that may or may not cause a change in the monitor's state. The method yields a state that is either an "OK state" or an "error state".

defined as triplets of a trigger, a condition and an action. The class diagram is shown in Figure 3.7 and the overall architecture can be seen in Figure 3.6.
With the monitor being defined, we needed to find a way to insert it into our app.

3.2.3 Running the monitors in the `ViewModel`

By design Android recommends using the Model-View-Viewmodel (or MVVM) architecture [54], where the `ViewModel` class takes care of the logic, so we decided to focus our efforts in instrumenting it.
We defined an abstract `MonitoredViewModel` class containing:

- a centralised `Monitor` observing a list of app-independent properties;
- an API for notifying the monitor before or after an event is triggered (exemplified in Figures 3.8 and 3.9):
  - `doNext` reads a trigger and feeds it to the monitor, updating the current state, and then executes a given block of code saving the result (or the

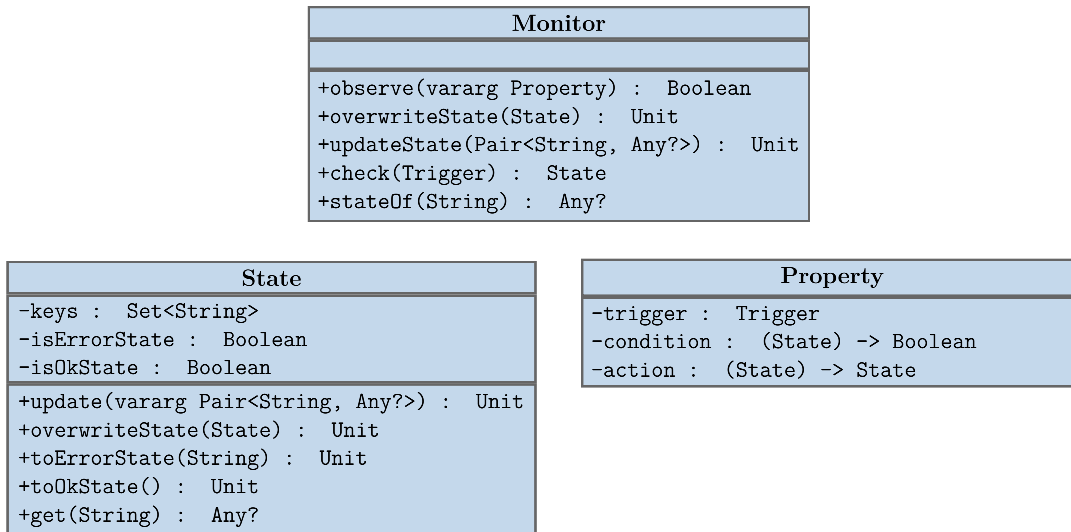

Figure 3.7: UML representation of the `State`, `Property` and `Monitor` classes. The `Trigger` class is a type alias for `String`.

error) in the state;
  - `afterResult` behaves in a similar way but executes the block of code before having the monitor check the trigger;
  - `maybeDoNext` is an enforcement-oriented version of `doNext` that only executes the block of code when the monitor outputs an "OK" state and reverts the state otherwise;
- an API for running coroutines in a controlled environment:
  - `launch` has the same effect as `launch`ing a coroutine normally, but it also saves the resulting job in the viewmodel for future reference;
  - `await` calls the `await` function for a given `Deferred` value and then checks the monitor for properties defined for `async` tasks.

We decided to use a single, centralised monitor in order to only have one block of business logic running at one time. This `Monitor` instance would receive a `Trigger` from the API and, one by one, check all properties that trigger was relevant for. Any such properties would be, sequentially, examined: if the given `Condition`, applied to the internal `State`, were to yield `true`, then the monitor would update the `State` as specified in its own `Action`.

We set up the viewmodel to extend the `MonitoredViewModel` and defined the properties from the list in 3.2, translating them to `Property` instances as shown in Listing 6.

The first property, **AlwaysOneJob** was checked using the `maybeDoNext` function and we noticed that it successfully prevented the viewmodel from starting any search operations if another one was already processing. To test the **SuccessWithJSON** property we set the reading task to randomly return an invalid result and verified that the state of the monitor was correctly set to "error".

We replaced all invocations of the `launch` and `await` coroutine functions with the ones defined in our class. This way we could easily verify the other, non-app specific properties.

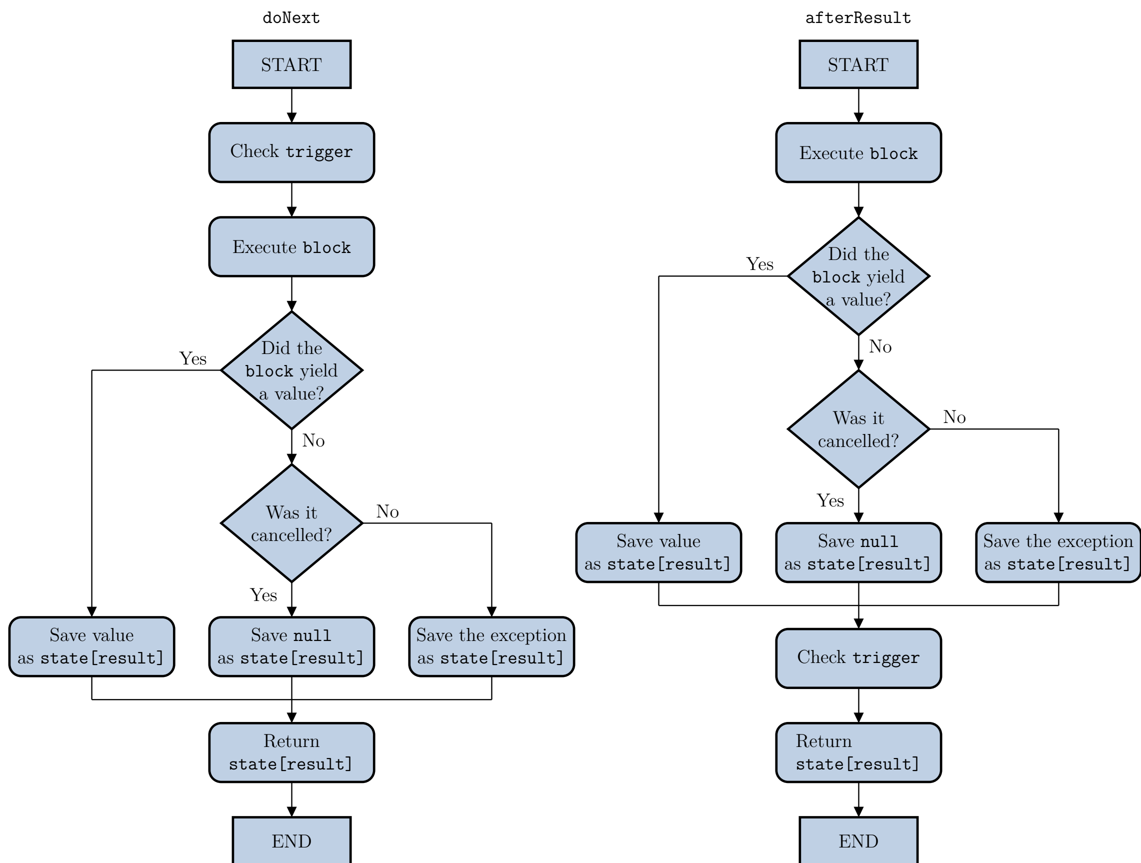

Figure 3.8: Flow chart detailing the logic for the functions `doNext` and `afterResult` of the `MonitoredViewModel` class.

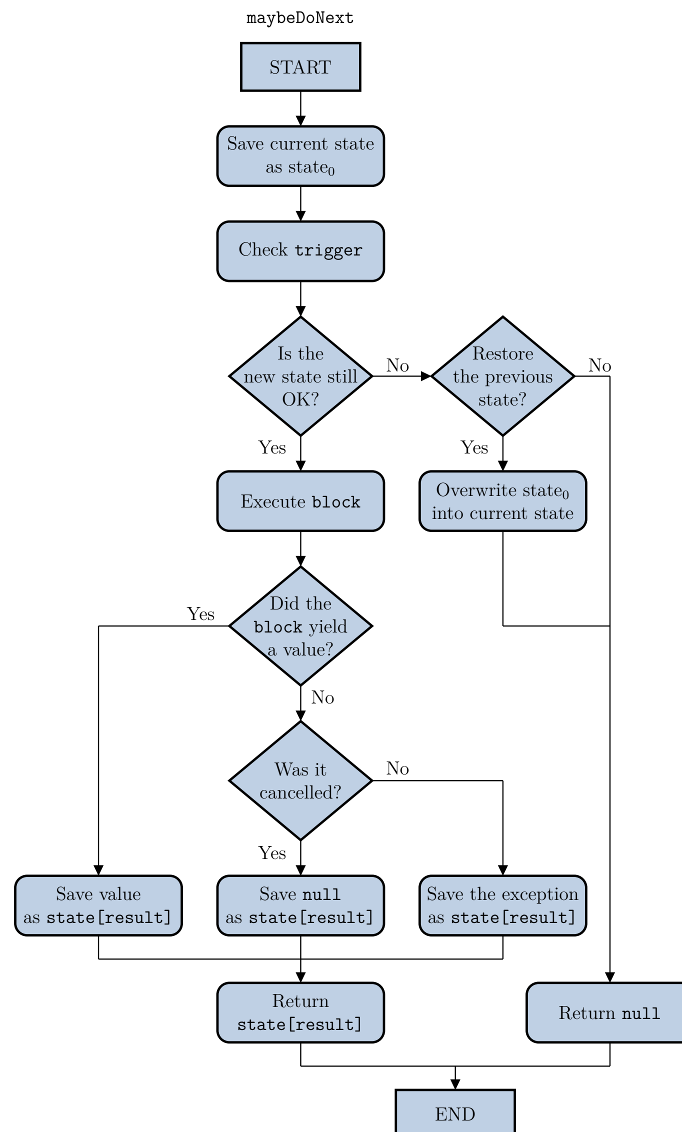

Figure 3.9: Flow chart for the function `maybeDoNext` of the `MonitoredViewModel` class.

```
private val alwaysOneJob = Property(
    trigger = TRIGGER_SEARCH,
    condition = { state ->
        state[KEY_ACTIVE_SEARCHES]?.let { it as Int > 0 } ?: false
    },
    action = { state ->
        state.apply {
            toErrorState("One search job must be running at most")
        }
    }
)
```

```
private val successWithJson = Property(
    trigger = TRIGGER_FINISHED_READING,
    condition = { state ->
        if (state.isOkState) false
        else try {
            JSONObject(state[State.KEY_RESULT] as String)
            false
        } catch (e: JSONException) {
            // The result is not a valid JSON object.
            true
        } catch (e: ClassCastException) {
            // The result is not a String.
            true
        }
    },
    action = { state ->
        state.apply {
            toErrorState("Successful read but invalid JSON")
        }
    }
)
```

Listing 6: Implementations of the **AlwaysOneJob** and **SuccessWithJSON** properties expressed using the `Property` class.

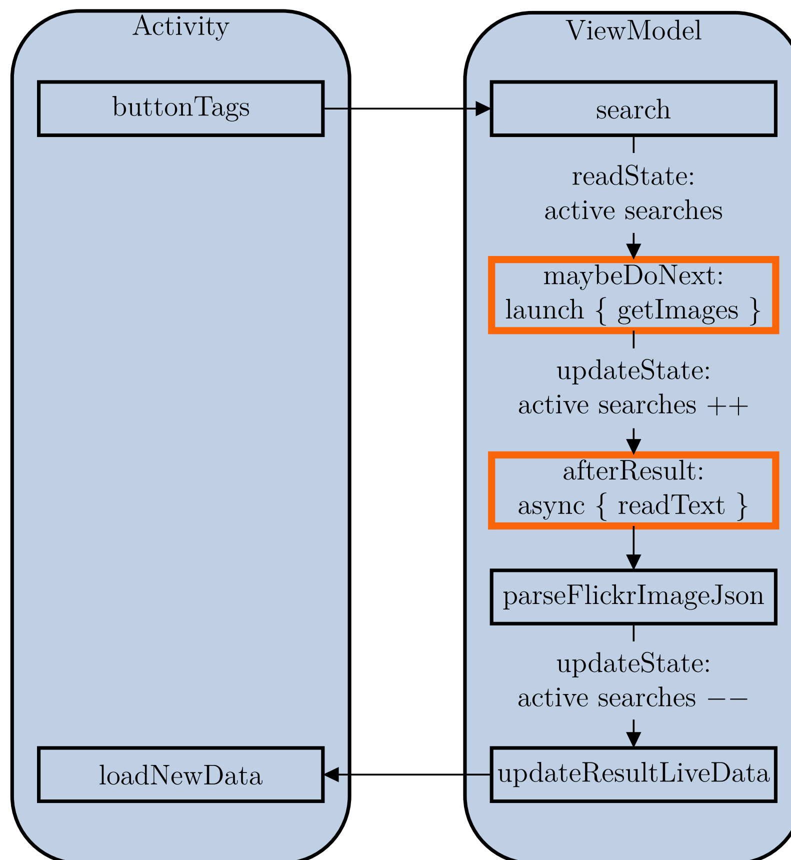

Figure 3.10: Updated version of the architecture in Figure 3.5 with added information on instrumentation: the coroutine builder methods `launch` and `async`, displayed in a different colour, are wrapped inside the `maybeDoNext` and `afterResult` methods (shown respectively in Figures 3.9 and 3.8). Throughout the execution of each method inside the viewmodel, the internal state of the `Monitor` instance is updated with the number of search operations currently ongoing ("active searches ++" signifying an increment by 1 and "active searches −−" a decrement by 1).

3.2.3.1 Considerations

This implementation proved effective to a degree but, obviously, came with its share of disadvantages. The first drawback was in the nature of the `Monitor`: it was a unique instance "containing" a list of properties, a big difference to our planned approach with several monitors defined for coroutine- and component-specific events. Another shortcoming was the reliance on being called by the `doNext`, `maybeDoNext` and `afterResult` methods, with the invocation procedure being fairly intrusive as shown in the Listings 7 and 8 that feature the same method, respectively with and without the boilerplate code.
We wanted to make a second attempt at a basic implementation, while trying to keep the boilerplate code to a minimum and, if possible, move it away from the viewmodel to a separate structure. This would help split what concerned the execution of the app and what concerned its behaviour throughout the execution.

```
fun search(tags: String) {
    val uri = createFlickrUri(tags)
    launch(coroutineExceptionHandler) {
        val runningJobs =
            readState(KEY_ACTIVE_SEARCHES) as Int? ?: 0
        maybeDoNext(TRIGGER_SEARCH, true) {
            updateState(KEY_ACTIVE_SEARCHES to runningJobs + 1)
            getImages(uri)
        }
    }
}
```

Listing 7: Implementation of the `search` method: the code on lines 5-8 is only used to manually inform the viewmodel of a new search operation being initiated, with invocations of the methods `readState` and `updateState` to respectively read and change the internal state of the monitor.

```
fun search(tags: String) {
    val uri = createFlickrUri(tags)
    launch(coroutineExceptionHandler) {
        getImages(uri)
    }
}
```

Listing 8: "Ideal" implementation of the `search` method where the boilerplate code featured in Listing 7 is hidden.

3.3 Second Approach: Using Annotations to Generate Compile-time Monitors

One of the most popular ways to instrument classes in Android is, as mentioned, the Aspect-Oriented paradigm. Kotlin, however, introduces a functional approach to writing Android code which can sometimes be used instead of AOP [55].
We decided to investigate this situation to determine in which cases a functional approach can help us better define and monitor a given property.
Kotlin annotations are, like in Java, used for attaching metadata to blocks of code: this is then processed by the compiler which, based on the annotation, can run additional checks (such as the Java `@Override` ensuring that a method is actually overriding something), relax some constraints (e.g. `@SuppressWarnings` hiding some warning messages) or generate some specific code, as is the case with the AspectJ compiler AJC.
With these premises, we looked into *what* we could actually employ annotations for:

- generate boilerplate instead of needing the user to move through the cumbersome API of the hardcoded implementation seen earlier: this, however, was not something we strictly needed annotations to do as the previously mentioned

functional approach could abstract away most of the unnecessary code;
- employing AJC to weave monitors in our Kotlin code: despite sounding promising, this solution did not really employ the unique features of the language and used instead its interoperability with Java; this essentially meant that we would be weaving monitors into the compiled JVM classes as opposed to the actual code written by the user, so we opted not to use this solution, either.

We decided to forego this approach and determined two other possible implementations for our tool:
(a) extending the `CoroutineScope` to include monitors;
(b) extending the lifecycle components (such as the `MonitoredViewModel` of the initial implementation) and carry out the instrumentation inside of them.

Solution (a) seemed like an elegant solution as that would allow us to execute any verification in a way that would be transparent to the end user. Unfortunately this solution came with the drawback of making it harder for us to spawn a coroutine with a block of code of type

```
suspend MonitoredCoroutineScope.() -> T
```

(where `T` can be any Kotlin type) using the existing infrastructure.
Solution (b) was, on the other hand, in contrast with the Kotlin guideline of defining every coroutine builder as an extension of `CoroutineScope`, because it would put a different class (be it `LifecycleOwner` or `ViewModel`) between the scope and the builder function.
Our final solution was therefore a blend of the two ideas: we implemented a `MonitoredComponent` interface, containing the monitoring API, and then overloaded the builder functions `launch` and `async` to include the component as an argument. It is described in the next chapter.

4

Implementation of an API for Monitoring Kotlin Coroutines

Our experiences with the approaches presented in chapter 3 made it clear for us what we wanted to achieve: an API that would take away the boilerplate and still monitor the points specified in 3.1.5, following the Kotlin coding guidelines of tying coroutine builders to the `CoroutineScope` class.
Our API should not replace the existing one for coroutines, so making new functions (e.g. "monitoredAsync" or "launchWithMonitor") was out of scope. Similarly, we did not want to add a number of extra steps that the programmer should take when defining monitors, since that was one of the drawbacks of the hardcoded model as discussed in section 3.2.3.1.
The result was a new interface called `MonitoredComponent`, which is examined in this chapter.
Section 4.1 describes our implementation for the interface and all classes related to it; section 4.2 details how we refactored the implementation to improve its API and move it closer to our research goal.

4.1 The Interface `MonitoredComponent` and its API

We thought that it would be best to stay as close as possible to the existing tools: overloading [64] the existing methods `launch` and `async` was deemed the most appropriate course of action. As shown in Listing 9, we added the `MonitoredComponent` class as an argument so as to use it "behind the curtains" to observe all points described earlier in the processes of starting and terminating tasks on coroutines.

4.1.1 First Version of the API

Our first goal was to move away as much as possible from the hardcoded architecture. We thus decided to employ class inheritance to abstract away from the previous concept of a viewmodel handling the states of both the UI and of the monitoring system: it was definitely confusing, since two different concepts of state (one referred to the Android app and one to its compliance with our properties) could easily be misinterpreted, and it prevented us from working on the `Activity` class, which is a lifecycle-aware component in and of itself, with a different lifecycle than the `ViewModel`'s (see Figure 3.1 for reference).

```
fun CoroutineScope.launch(
    component: MonitoredComponent,
    context: CoroutineContext = EmptyCoroutineContext,
    block: suspend CoroutineScope.() -> Unit
): Job

fun <T> CoroutineScope.async(
    component: MonitoredComponent,
    context: CoroutineContext = EmptyCoroutineContext,
    block: suspend CoroutineScope.() -> Unit
): Deferred<T>
```

Listing 9: Our version of the coroutine builder methods: the signature is thus the same as the standard one for `CoroutineScope.launch` [56] and `CoroutineScope.async` [57] with the sole addition of the `MonitoredComponent` as an argument.

```
val launchedTasks: ArrayList<LaunchableTask>
val recommendedDispatchers: HashMap<String, CoroutineDispatcher>
val defaultHandler: CoroutineExceptionHandler
val monitoredApplication: MonitoredApplication?
```

Listing 10: Declarations for the internal data structures in the `MonitoredComponent` interface.

Our `MonitoredComponent` interface was therefore supposed to be implemented by both the `Activity` and `ViewModel` classes, not to mention any other class the developer should see fit.

It needed to hold records of the tasks that had started, as well as their dispatchers and exception handlers. We defined the following data structures:

- `launchedTasks`, an `ArrayList` that kept track of all the tasks launched with the current `MonitoredComponent` as argument;
- `recommendedDispatchers`, a `HashMap` storing the "best" coroutine dispatcher to use with each task;
- `defaultHandler`, a `CoroutineExceptionHandler` that should be inserted into "unhandled" coroutine contexts according to the *NeedHandler* property;
- `monitoredApplication`, an accessor providing communication between the component and the `MonitoredApplication` instance: this class will be expanded upon in a dedicated section.

The code declaration for each of these data structures can be seen in Listing 10. Some of them have types that are not immediately recognisable, like `ArrayList<LaunchableTask>`, and will be explained below.

4.1.1.1 Keeping Track of Tasks

According to the property **DestroyedWithOwner** we needed the `MonitoredComponent` instance to know which tasks should be cancelled in the case of termination. We therefore needed to store a reference to each task started inside our component.
The Kotlin library "AndroidX lifecycle" [58] already provides a possible solution to this: instances of `CoroutineScope` are defined for the `LifecycleOwner` and `ViewModel` classes (respectively `lifecycleScope` and `viewModelScope`). It's guaranteed that all tasks spawned within these scopes will be cancelled when the related instance is destroyed. `LifecycleOwner` is a superclass of `FragmentActivity`, which means `Activity` and `Fragment` instances have access to its lifecycle-aware methods and fields.
Despite these `CoroutineScope` instances being readily available for both of the classes that we were going to work with, we decided to define our own implementation for this. Our reasoning was that the methods in the API could be overridden and, therefore, the user would always be able to define additional operations. This is not possible by employing the AndroidX solution by itself.
In order to record tasks, we introduced the `launchedTasks` array. This data structure was supposed to contain references to tasks, so we decided to focus on how tasks are usually carried out:

1. a task is initialised but not started yet, identifiable as the time frame between the start of the coroutine builder method and the actual execution of the block of code containing the task;
2. the initialisation is completed and the task is running;
3. the task is terminated.

The `launchedTasks` array needed therefore to follow these three states: we translated this into code by defining the class `LaunchableTask` and its three subtypes:

- a `Reserved` slot was waiting for a task to be initialised;
- a `LaunchedTask` contained a reference to a `Job` that was currently running;
- an `Empty` slot used to contain a task but was "freed" once the task terminated.

Each entry in the array would therefore have to switch between these three types. We defined the following methods to apply this logic to the `launchedTasks` data structure.

- `reserveTask`: when a task is being initialised, one `Empty` entry will become `Reserved`;
- `newTask`: when a task is running, the previous entry will no longer be `Reserved` and will instead identify a `LaunchedTask`;
- `onComplete`: when a task ends its execution, the `LaunchedTask` entry referencing it will become obsolete and change into an `Empty` position.

The types of tasks that we could record were only limited to the three cases detailed above; for this reason we implemented `LaunchableTask` as a `sealed` class. This particular kind of class is used for representing restricted class hierarchies and is similar to the Java `enum` type [65].
On this basis we wrote the methods as seen in Listing 11. The methods were all synchronised blocks of code where the content of the `launchedTasks` array could not be modified by any other thread. The methods could, of course, be

```
/**
 * Reserves the first available position in [launchedTasks]
 * to be filled with a task.
 * @return Pair<Int, Int> containing a unique reservation key
 * and the position to be filled (0..lastIndex)
 */
fun reserveTask(): Pair<Int, Int>

/**
 * Inserts the given [task] in the position marked with [index].
 * @throws ConcurrentModificationException if the [key] does not
 * match with what's in the position, or if the position had not
 * been reserved earlier.
 */
fun newTask(index: Int, task: Job, key: Int): Unit

/**
 * Removes the task in the position marked with [index].
 * @throws ConcurrentModificationException if the position is
 * not holding either a LaunchedTask or a reservation to the
 * same task.
 */
fun onComplete(index: Int, key: Int): Unit
```

Listing 11: Formal definition of the methods with which the `MonitoredComponent` interface handled the initialisation, start and termination of tasks.

overridden by the developer; we decided therefore to integrate this behaviour with `ConcurrentModificationException` instances being thrown in the case of concurrent access to the data structure.

The signature for the method `onComplete` did not feature a `Job` instance among its arguments: there was a possibility that a task could be cancelled before being launched (which could happen, for example, if the component were to be destroyed while the task was being started). In order to handle this edge case as well as the other three we defined a handshake mechanism:

1. the coroutine scope warned the `MonitoredComponent` instance that a task was being started by calling the method `reserveTask`;
2. the `MonitoredComponent` instance generated a random `Int` key and used it to "reserve" the first available position in the `launchedTasks` array, then returned both the key and the position to the coroutine scope;
3. the coroutine scope generated a task running the given block of code and, before termination, called the `onComplete` method of the `MonitoredComponent` instance;
4. once the task was created, the coroutine scope would call the method `newTask` to register the task into the `MonitoredComponent` instance;
5. as soon as the task terminated and the `onComplete` method was invoked,

```
1  fun CoroutineScope.launch(
2      component: MonitoredComponent,
3      context: CoroutineContext = EmptyCoroutineContext,
4      start: CoroutineStart = CoroutineStart.DEFAULT,
5      block: suspend CoroutineScope.() -> Unit
6  ): Job {
7      val (key, index) = component.reserveTask()
8      val task = launch {
9          try {
10             block()
11         } finally {
12             component.onComplete(index, key)
13         }
14     }
15     component.newTask(index, task, key)
16     return task
17 }
```

Listing 12: Implementation of our version of the `launch` method in order to save references to the launched tasks. As can be seen, our version of `launch` was actually a "wrapper" function calling the default `launch` (line 8) after performing some control operations. The call on line 12 was inside a `finally` block, meaning it would be executed whenever the task was cancelled. The task might actually be cancelled before the call on line 15, which would result in the `launchedTasks` array *not* holding any references to the task: for this reason we used the `key` argument in the method `onComplete` rather than the `task` itself. The same applied to our version of the `async` method, which performed the same operations and then call the default version of the same method.

the `MonitoredComponent` instance would remove any reference to it inside its `launchedTasks` array.

The translation of the above into Kotlin code is shown in Listing 12.

4.1.1.2 Handling Exceptions

The properties **NormalAsync**, **ExceptionalAsync** and **NeedHandler** described earlier required coroutines to follow a precise behaviour:

- for coroutines created with the `launch` method there needed to be a handler;
- for ones created with the `async` method there needed to be some sort of exception handling.

Since tasks started via the `launch` method will only employ the handler at top level, we opted to make sure that an object of type `CoroutineExceptionHandler` be present at all times: the `MonitoredComponent` instance had no way of knowing whether the task currently being started was in the top level or not.

The `launch` method we defined for the `MonitoredComponent` interface checked the given coroutine context and looked for a handler inside it; if no handler was to

```
1  fun CoroutineScope.launch(
2      component: MonitoredComponent,
3      context: CoroutineContext = EmptyCoroutineContext,
4      block: suspend CoroutineScope.() -> Unit
5  ): Job {
6      val handler = context[CoroutineExceptionHandler.Key]
7          ?: component.defaultHandler
8      return launch (context + handler) {
9          try {
10             block()
11         } catch (c: CancellationException) {
12             // Ignore this exception.
13         } catch (e: Exception) {
14             throw e
15         }
16     }
17 }
```

Listing 13: Implementation of our version of the `launch` method in order to ensure that one `CoroutineExceptionHandler` instance should always be in the context. The assignment at lines 6-7 ensured that the handler we would use be either the one already in the context or, should no such object exist in the context, the `defaultHandler` data structure defined inside the component. By employing the expression `context[CoroutineExceptionHandler.Key]` we queried the coroutine context for its current handler (receiving a `null` if no handler was defined). It should be noted that the lines 11-15 add technically nothing to the function: ignoring `CancellationExceptions` is standard coroutine behaviour since they are just a notice of cancellation, and any `Exception` thrown inside a block of code is automatically rethrown (as stated in 2.2.3).

be found, the `defaultHandler` data structure would be added as shown in Listing 13. The `async` method ignores any handlers in its context by design, therefore we did not need to check the coroutine context and only really had to rethrow any exceptions.
Adding the code in Listing 13 to the handshake mentioned in Listing 12 we ended up with the implementations featured in Listings 14 and 15, respectively for the `launch` and the `async` methods.
So far we had covered the properties **DestroyedWithOwner**, **NormalAsync**, **ExceptionalAsync** and **NeedHandler** by implementing monitors for the events **launch(context, task)**, **async(context, task)** and **component.destruction()** (for each **component** defined as an instance of `MonitoredComponent`).

```
fun CoroutineScope.launch(
    component: MonitoredComponent,
    context: CoroutineContext = EmptyCoroutineContext,
    block: suspend CoroutineScope.() -> Unit
): Job {
    val handler = context[CoroutineExceptionHandler.Key]
        ?: component.defaultHandler
    val (key, index) = component.reserveTask()
    val task = launch(context + handler) {
        try {
            block()
        } catch (c: CancellationException) {
            // Ignore this exception.
        } catch (e: Exception) {
            throw exception
        } finally {
            component.onComplete(index, key)
        }
    }
    component.newTask(index, task, key)
    return task
}
```

Listing 14: Our implementation of the `launch` method in order to both keep record of running tasks and ensure the presence of a `CoroutineExceptionHandler` instance in the coroutine context.

4.1.1.3 Remembering Failed Tasks

The properties **SlowDownUI** and **UpdateUI** required any given instance of `MonitoredComponent` to be able to perform two checks:

1. query the coroutine context for the current `CoroutineDispatcher` instance;
2. know whether a given block of code was going to update a UI element or perform a costly operation.

Unfortunately, we have no way of knowing in advance what a block of code will do once executed.

Fortunately we can instead *remember* what a given block of code did during its last execution. We decided to detect exceptions of types `CalledFromWrongThreadException` and `NetworkOnMainThreadException` thrown inside a previously spawned coroutine and use them to infer what thread we may want to use the next time the same block of code were to be executed.

This operation was, however, risky: we needed to recognise a block of code. We decided to solve this by inspecting the signature of the caller methods in the stack-trace.

By examining stacktraces at the beginning of our `launch` and `async` methods, we

```
fun <T> CoroutineScope.async(
    component: MonitoredComponent,
    context: CoroutineContext = EmptyCoroutineContext,
    block: suspend CoroutineScope.() -> Unit
): Deferred<T> {
    val (key, index) = component.reserveTask()
    val task = launch(context + handler) {
        try {
            block()
        } catch (e: Exception) {
            throw exception
        } finally {
            component.onComplete(index, key)
        }
    }
    component.newTask(index, task, key)
    return task
}
```

Listing 15: Our implementation of the `async` method in order to both keep record of running tasks and rethrow any exceptions that should be raised during the task's execution.

found that:

- the first element (`stacktrace[0]`) referred obviously to the current method;
- the second element referred to the "default" version of the current method: this is the extra method created by the Kotlin compiler to pre-fill any arguments with a default value (previously mentioned in section 2.2.2);
- the third element is the line of code where the current method is invoked: it refers to a call of `invokeSuspend`, which is how Kotlin translates coroutine builders to JVM bytecode;
- elements after the third one are different based on whether the current coroutine is being started or resumed.

We decided to use the third element (i.e. `stacktrace[2]`) as our "identifier" for each given task. Whether this is a valid course of action or not, we decided to run with it for the time being.
The arbitrary identifier we assigned to each block of code was therefore a `String` containing class and method name, as well as line number, of the third stacktrace element.
Our next challenge was then defining how we should remember which tasks fail because of their dispatcher. We defined the data structure `recommendedDispatchers` as a `HashMap` inside our `MonitoredComponent` exactly for this purpose. The map would use the identifier as the key to store each instance of `CoroutineDispatcher`. We considered the possibility of merging this structure with the previously defined `launchedTasks` but we decided to keep them as separate entities for three reasons:

1. we wanted to have the tracking of task and dispatcher as two separate features

for the end user to override separately, if needed;
2. we tracked tasks by their stacktrace-based identifier in this case and accommodating the `launchedTasks` to follow this convention would add unnecessary complexity;
3. the `launchedTasks` were only relevant while the component was running, while we wanted the application to "remember" the correct dispatchers even after the component's termination: this would come in handy in the case that an exception caused the current Android activity to crash.

The `recommendedDispatchers` were defined as a `Map` where each task's id acted as a key and each task's recommended dispatcher was the value. As for what dispatcher was the recommended one for a given task, we assumed that:

- if the task raised a `NetworkOnMainThreadException`, we should recommend `Dispatchers.IO`;
- if the task raised a `CalledFromWrongThreadException`, we should recommend `Dispatchers.Main`.

4.1.1.4 Referencing the Correct Task

This presented us with another obstacle: how to make sure the dispatcher was associated with the right task and not, for example, a parent task that received an exception from a child (or the child of a child and so on)?
Our idea was to introduce a new exceptional type intended to replace both `CalledFromWrongThreadException` and `NetworkOnMainThreadException`: the failing task would handle the actual exception and then throw the "new" exception instead of rethrowing the real one.
We introduced a new subclass of the `Exception` type to carry out this task. The new class, called `WrongDispatcherException`, stores the original exception instance as its cause; it also carries a reference to the `CoroutineDispatcher` instance that should be used the next time the same task is launched. Its implementation can be seen in Listing 16.
In order for this to be effective, we needed to update the exception handling in the methods `launch` and `async`. For the latter it was enough to change the `try-catch` block but, as mentioned earlier, the former requires its own dedicated handler: for this reason we created a new subclass of `CoroutineExceptionHandler` that would automatically carry out the conversion from classes `NetworkOnMainThreadException` and `CalledFromWrongThreadException` to the new `WrongDispatcherException` type, while handling any other exception normally.
The new handler, which we called `WrongDispatcherExceptionHandler`, needed to

```
class WrongDispatcherException(
    cause: Throwable,
    val recommendedDispatcher: CoroutineDispatcher
) : Exception(cause)
```

Listing 16: Declaration of the `WrongDispatcherException` class.

actually process whatever exception it would receive, so we gave it a “normal” `CoroutineExceptionHandler` instance as a private field.
The updated `launch` would now follow the logic illustrated in Figure 4.1.

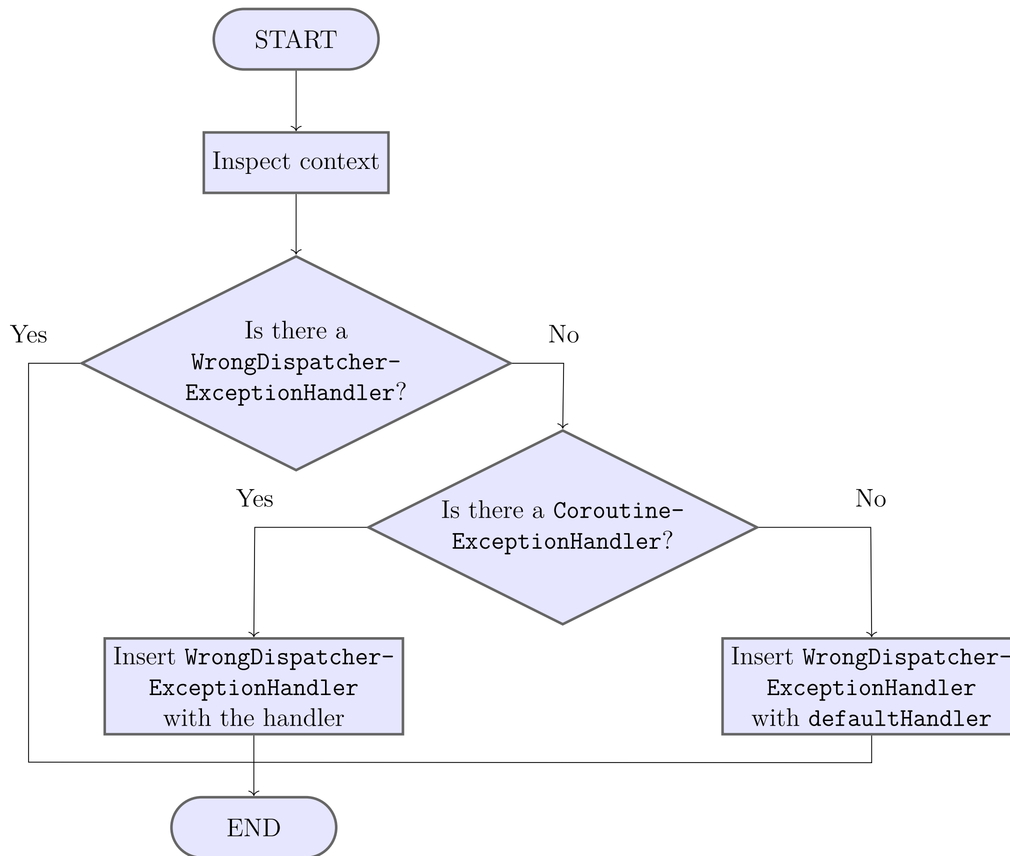

Figure 4.1: Representation of the logic followed by the updated `launch` method for saving a `CoroutineExceptionHandler` inside the context.

When handling exceptions the `WrongDispatcherExceptionHandler` would check whether the exception was an instance of either relevant classes: if it was, it would be converted to `WrongDispatcherException`. Either way it would then be processed by the internal handler.
This solution presented a complication: we needed to distinguish the relevant exceptional types. In the case of `NetworkOnMainThreadException` instances this operation was straightforward, as all we had to do was employ the `is` operator (which is the Kotlin equivalent to Java’s `instanceof`); with instances of `CalledFromWrongThreadException` this was not as simple, since the exception type is not actually exposed. It is instead defined inside the `android.view.ViewRootImpl` class which is also not exposed.
We used Kotlin’s extension functions to create a fake accessor for the `Throwable` class that would act as a surrogate type check for this kind of exception. Its implementation can be seen in Listing 17.
Of course, in addition to propagating the `WrongDispatcherException` we needed to record the actual exception and register a recommended `CoroutineDispatcher` instance for the failing task. This part was also straightforward as all we needed

```
val Throwable.isCalledFromWrongThreadException
    get() = javaClass.name ==
        "android.view.ViewRootImpl\$CalledFromWrongThreadException"
```

Listing 17: The extension function that we implemented as a fake accessor for the `Throwable` class. It returns `true` when called on an instance of `CalledFromWrongThreadException`, allowing us to distinguish objects of this type by matching their class name.

was to operate in a `try-catch` block inside each of our methods.

The final part was then the detection of any recommended dispatchers for a given task. By looking in the given coroutine context it is possible to retrieve the registered dispatcher by using the expression

`context[ContinuationInterceptor.Key]`

which evaluates either to `null` or to an object of type `ContinuationInterceptor`, a superclass of `CoroutineDispatcher`.
The `MonitoredComponent` instance would be queried for a recommended dispatcher based on the task's id; as mentioned before, this id was extracted from the third stacktrace element. Based on the result yielded by the query, one of the following would happen:

- `null`: this result means that there is no recommended dispatcher, either because the task has never failed before or because it's been launched for the first time; in this case we used the dispatcher currently in the context (or `Dispatchers.Default` in the unlikely case that no dispatcher was already present);
- a `CoroutineDispatcher` instance that is the same as the one in the context: in this case we would not need to change anything;
- a dispatcher that is different from the one in the context: we forced the recommended dispatcher instead.

4.1.2 Implementation of the Methods `launch` and `async`

The only property left to explore was now **ResumeIfNeeded**. Unfortunately, as mentioned earlier, we had no way of knowing what a task would do before executing it.
Fortunately the `CoroutineScope` provides us with a flag indicating whether a given task is active (i.e. running) or not: we could add an explicit check for this flag before launching the task but we had no way of injecting it at set intervals during a block of code. An alternative would be starting a task with a given timeout value, forcing its termination after a set amount of milliseconds had passed: this would not work on long computations such as the example mentioned in the official Kotlin guide for coroutines [61] but, more importantly, we felt it would not actually "check that the task is still needed" but, rather, it would terminate the task in the case that it should take too long to complete.
We decided therefore to downsize this property to just reading the `isActive` flag

before running the block of code.
The final version of the code for the customised method `launch` is shown in Listing 18. Some remarks about the implementation:

- on line 8 we combined the input context with the one already present in the `CoroutineScope` using the built-in `plus` operator: this generates a new `CoroutineContext` where the properties of the left operand are overwritten by the non-null properties of the right operand;
- on lines 31-32 we used the `plus` operator again by adding `newHandler` and `newDispatcher` to the context, effectively overwriting its previous handler and dispatcher.
- on line 37 we used the `isActive` flag to ensure that the task has not been cancelled while we were operating on the coroutine context, which was our way of verifying the property **ResumeIfNeeded**.
- on line 44 we employed `Throwable.toWrongDispatcherException`, an extension function that we defined to filter the exception types that we were interested in: this function returns a `WrongDispatcherException` only for instances of either `NetworkOnMainThreadException` or `CalledFromWrongThreadException`, returning the unaltered object in any other case; this allowed us to perform the check on line 45.

The code for the `async` method, shown in Listing 19, was similar but presented some key differences:

- there was no need for a `CoroutineExceptionHandler` so we skipped that part and only looked at the dispatcher on lines 12-21;
- on line 27 we would throw a `CancellationException` if the task was no longer needed: this was necessary because we had no value to return; we felt that this course of action would not violate the property **NormalAsync** because it matched the second case that we could not fit in our definitions for the "good" or "bad" behaviour;
- there was no `catch` block for `CancellationException` since this type was only ignored by `launch`.

Compared to our definition of monitors in Section 3.1.5 we only really used three of the four of them: the monitor **await(deferredTask)** was not actually employed, as we could fit the verification for the properties **NormalAsync** and **ExceptionalAsync** inside the **async(context, task)**. This was made possible by the fact that `async` defines a coroutine to be built, and `await` actually starts the execution and waits for the result to be returned: by moving the appropriate checks to the overriden `async` method we effectively ensured they would be executed once `await` was called.

```
fun CoroutineScope.launch(
    component: MonitoredComponent,
    context: CoroutineContext = EmptyCoroutineContext,
    start: CoroutineStart = CoroutineStart.DEFAULT,
    block: suspend CoroutineScope.() -> Unit
): Job {
    // Extract CoroutineContext data.
    val combinedContext = coroutineContext + context
    val dispatcher = combinedContext[ContinuationInterceptor.Key]
        as CoroutineDispatcher?
    val handler = combinedContext[CoroutineExceptionHandler.Key]
    // Extract utility data.
    val id = Exception().stackTrace[2].let {
        "${it.className}.${it.methodName}:${it.lineNumber}"
    }
    // Assign exception handler and dispatcher.
    val newHandler =
        if (handler is WrongDispatcherExceptionHandler) handler
        else WrongDispatcherExceptionHandler(
            handler ?: component.defaultHandler
        )
    val newDispatcher = when (component.getRecommendedDispatcher(id)) {
            null,
            dispatcher -> dispatcher ?: Dispatchers.Default
            else -> component.getRecommendedDispatcher(id)!!
    }
    val newContext = combinedContext + newHandler + newDispatcher
    // Save and begin task.
    val (key, index) = component.reserveTask()
    val task = launch(newContext, start) {
        try {
            if (isActive) block()
        } catch (c: CancellationException) {
            // Ignore this exception.
        } catch (w: WrongDispatcherException) {
            // Not much to do but rethrow.
            throw w
        } catch (e: Exception) {
            val exception = e.toWrongDispatcherException()
            if (exception is WrongDispatcherException) {
                component.saveRecommendedDispatcher(
                    id,
                    exception.recommendedDispatcher
                )
            }
            throw exception
        } finally {
            component.onComplete(index, key)
        }
    }
    component.newTask(index, task, key)
    return task
}
```

Listing 18: Complete implementation for the `launch` method.

```
fun <T> CoroutineScope.async(
    component: MonitoredComponent,
    context: CoroutineContext = EmptyCoroutineContext,
    start: CoroutineStart = CoroutineStart.DEFAULT,
    block: suspend CoroutineScope.() -> T
): Deferred<T> {
    // Extract utility data.
    val id = Exception().stackTrace[2].let {
        "${it.className}.${it.methodName}:${it.lineNumber}"
    }
    // Extract CoroutineContext data and assign dispatcher.
    val newContext = (coroutineContext + context).let {
        val dispatcher = it[ContinuationInterceptor.Key]
            as CoroutineDispatcher?
        it + when (component.getRecommendedDispatcher(id)) {
            null,
            dispatcher -> dispatcher ?: Dispatchers.Default
            else -> component.getRecommendedDispatcher(id)!!
        }
    }
    // Save and begin task.
    val (key, index) = component.reserveTask()
    val task = async(newContext, start) {
        try {
            if (isActive) block()
            else throw CancellationException()
        } catch (w: WrongDispatcherException) {
            throw w
        } catch (e: Exception) {
            val exception = e.toWrongDispatcherException()
            if (exception is WrongDispatcherException)
                component.saveRecommendedDispatcher(
                    id,
                    exception.recommendedDispatcher
                )
            throw exception
        } finally {
            component.onComplete(index, key)
        }
    }
    component.newTask(index, task, key)
    return task
}
```

Listing 19: Complete implementation for the `async` method.

4.1.3 The `MonitoredApplication` Class

One of the main concerns about the approach to properties **UpdateUI** and **SlowDownUI** was the persistence of any recommended dispatchers: if a `WrongDispatcherException` were to crash the current activity, we would lose every record and that would make our efforts moot.
Our solution was to save our records somewhere. We considered two possibilities:

1. save the dispatchers in memory, making sure that they exist so long as the app itself is not terminated;
2. save the dispatchers in a file, making it accessible even after the app is terminated.

We decided to keep our records inside the application memory so as to avoid taking a potentially large space in the device's internal memory and to counter the possible slowdowns caused by frequent operations of file reading/writing.
The class `MonitoredApplication`, a subclass of `Application`, was our approach to this: it holds a collection containing the `recommendedDispatchers` map for each instance of `MonitoredComponent`. These maps are stored using as a key the class name of their owner, as can be seen in Listing 20.
Two methods, `loadRecommendedDispatchers` and `saveRecommendedDispatchers`, are exposed for the `MonitoredComponent` instances to load and save any records.
Our custom application class includes one added feature: right before the app is terminated, a service prints the content of each saved `recommendedDispatchers` map. This string is visible on the device log, which we thought was a good compromise between storing everything in memory and creating an output file.
The printout looks like this:

```
2020-01-15 12:06:38.035 12315-12315/com.android.rv D/Report:
Post-execution report for app com.android.rv.KotlinRV:
Component: com.android.rv.properties.BrowsePicturesViewModel
 com.android.rv.ViewKt$loadFrom$2$1.invokeSuspend:51 =>
     LimitingDispatcher@849932d[dispatcher = DefaultDispatcher]
 com.android.rv.ViewKt.loadFrom:47 => Main
Component: com.android.rv.properties.BrowsePicturesActivity
```

The output above signifies that the coroutine launched on line 51 of the `loadFrom` function in the source file "View.kt" should use a dispatcher for background threads while the invocation on line 47 of the same function should use the UI thread.
Of course this deduction is not immediately comprehensible by the formatting but we believed it to be somewhat readable by any Kotlin programmer; there is always the option of reworking the way the output is formatted in order to make it more explicit.

```kotlin
internal val recommendedDispatchers =
    hashMapOf<String, HashMap<String, CoroutineDispatcher>>()
```

Listing 20: Declaration and initialisation of the data structure we used to keep record of the recommended coroutine dispatchers.

Before our application was fully operational, we needed to define how it should be accessed by objects implementing the MonitoredComponent interface: they are supposed to be at least instances of Activity and ViewModel, so we defined separate implementations for either case.
The only "common grounds" between implementations were that:

- the method loadRecommendedDispatchers should be called as soon as the component is created;
- the method saveRecommendedDispatchers should be called right before the component is destroyed;
- the accessor monitoredApplication should return the current application if it is an instance of MonitoredApplication or null otherwise.

4.1.4 The MonitoredActivity Class

The Activity class extends LifecycleOwner and therefore inherits its lifecycle awareness. It also holds a reference to the Application instance. For these reasons its implementation was quite straightforward:

- in the onCreate method we initialise launchedTasks and, unless the monitoredApplication yields null, we load the recommendedDispatchers;
- in the onDestroy method we check that all elements of launchedTasks are Empty (and proceed to stop any tasks still running) and then we save all recommendedDispatchers if the monitoredApplication is not null.

The implementation for this class can be seen in Listing 21. The class was defined as abstract in order to force the user to extend it and create their own defaultHandler.

4.1.5 The MonitoredViewModel Class

The ViewModel class is not as simple as the Activity when it comes to lifecycle. While it does come with a onCleared method that is called right before destruction, it has no method equivalent to the Activity's onCreate. Furthermore, it stores no references to the Application by default.
For our MonitoredViewModel we decided therefore to extend not the standard ViewModel class but, rather, the AndroidViewModel: it's a special subclass with a getApplication method. While this allowed us to link the class to the MonitoredApplication, we still had no way of loading the recommendedDispatchers at the moment of creation.
The viewmodel is never created automatically by the application but, rather, it is explicitly requested by its matching Activity. We took advantage of this and defined an extension function for the MonitoredActivity class. This function, shown in Listing 22, requests a MonitoredViewModel instance (or rather the instance of a subclass, since MonitoredViewModel is an abstract class much like the MonitoredActivity) and automatically calls its init method, which takes care of the aforementioned initialisation.
The code for the MonitoredViewModel can be seen in Listing 23.

```
abstract class MonitoredActivity : AppCompatActivity(),
    MonitoredComponent {
    override val launchedTasks = arrayListOf<LaunchableTask>()
    override val recommendedDispatchers =
        hashMapOf<String, CoroutineDispatcher>()

    override val monitoredApplication: MonitoredApplication?
        get() = application.monitored

    override fun onCreate(savedInstanceState: Bundle?) =
        super.onCreate(savedInstanceState).also {
            init()
            monitoredApplication?
                .loadRecommendedDispatchers(javaClass.name)?
                    .let { entries ->
                        recommendedDispatchers.putAll(entries)
                    }
        }

    override fun onDestroy() = super.onDestroy().also {
        destroy()
        monitoredApplication?.saveRecommendedDispatchers(
            javaClass.name,
            recommendedDispatchers
        )
    }
}
```

Listing 21: Code of the `MonitoredActivity`, subclass of `Activity` that implements the `MonitoredComponent` interface.

```
inline fun <reified T : MonitoredViewModel>
    MonitoredActivity.getMonitoredViewModel() =
        ViewModelProviders.of(this).get(T::class.java)
            .also { it.init() }
```

Listing 22: Implementation of the extension function `getMonitoredViewModel` that provides any instance of `MonitoredActivity` with a matching viewmodel and initialises it automatically.

```
abstract class MonitoredViewModel(
    application: Application
) : AndroidViewModel(application), MonitoredComponent {
    override val launchedTasks = arrayListOf<LaunchableTask>()
    override val recommendedDispatchers =
        hashMapOf<String, CoroutineDispatcher>()

    override val monitoredApplication: MonitoredApplication?
        get() = getApplication<Application>().let {
            if (it is MonitoredApplication) it
            else null
        }

    override fun init() = super.init().also {
        monitoredApplication?
            .loadRecommendedDispatchers(javaClass.name)?
                .let { entries ->
                    recommendedDispatchers.putAll(entries)
                }
    }

    override fun onCleared() = super.onCleared().also {
        destroy()
        monitoredApplication?
            .saveRecommendedDispatchers(
                javaClass.name,
                recommendedDispatchers
            )
    }
}
```

Listing 23: Code of the `MonitoredViewModel`, subclass of `ViewModel` that implements the `MonitoredComponent` interface.

4.2 The Finalised API

The API produced so far was an improvement over the previous hardcoded solution but it still came with two major shortcomings:

1. it was tied too tightly to a very specific set of properties;
2. it required the user to manually translate any properties to Kotlin code.

We tried therefore to address both issues and update our API such that it would allow monitoring a more diverse range of properties, solving the first problem and improving the abstraction so as to facilitate addressing the second problem at a later time.

The overall architecture was left untouched and we focused our efforts instead on improving the user's control over the operations carried out in the builder functions.

4.2.1 The Methods `launch` and `async`

Both methods gave little freedom to a developer, allowing any custom `MonitoredComponent` subclass to customise only:

- the implementations of `reserveTask`, `onComplete` and `newTask` when starting a task;
- the methods for saving and loading recommended `CoroutineDispatcher`s;
- what the `defaultHandler` object would do in case of exceptional behaviour, not to mention that the handler would still be wrapped inside a `WrongDispatcherExceptionHandler` instance.

The names of each method in the `MonitoredComponent` instance were also extremely specific and we thought they could incorrectly suggest the programmer to perform a limited range of operations.

We updated these names with the idea that they should not suggest an action but, rather, the moment when one or more actions are executed, leaving the developer to decide what they should do.

In the case of the `launch` method, the execution was updated as follows:

- any operations executed before starting the given task were replaced by two invocations:
  - `beforeTask` is a method that carries out controls shared between `launch` and `async`, such as checking the context for a `CoroutineDispatcher` instance;
  - `beforeLaunch` is a method carries out controls specific of `launch`, such as checking that the context contains an instance of `CoroutineExceptionHandler`;

  these methods update the coroutine context as seen in figure 4.2, effectively giving the user control over the context instead of the sole internal fields of the component;
- the task is spawned on the `coroutineScope` field defined for the component;
- when an exception is detected it will be handled by a `handleException` method, save for `CancellationException` objects that get their own `handleCancellation` method;

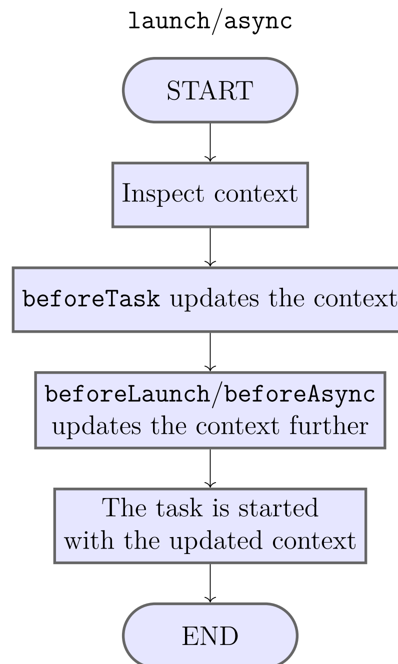

Figure 4.2: New behaviour of the `launch` and `async` methods with regard to the `CoroutineContext`: the methods `beforeTask` and `beforeLaunch/Async` progressively update the context, leaving the user free to define *how* it should be updated.

- the method `onComplete` was renamed to `afterTaskEnd` and, likewise, the method `newTask` to `afterTaskStart` as we aimed for an API more inspired by AspectJ's "before" and "after" advices.

The same changes went into the `async` method, with the obvious difference that it calls a `beforeAsync` method instead of a `beforeLaunch`, as seen in figure 4.2.
These changes needed of course to be reflected onto the `MonitoredComponent` and its subclasses.

4.2.2 The New `MonitoredComponent` Interface

As mentioned in the previous section, the `MonitoredComponent` interface was updated to feature its own `CoroutineScope` instance. We took the decision of removing the `launchedTasks` data structure: it added a handshaking mechanism that we deemed unnecessarily complicated and that scaled poorly with parallel executions due to its heavy use of locks. Locks impact execution speed by forcing processes to access a shared variable one at a time, and the entire operation of saving a reference to a task was tied to a single property rather than to the whole architecture. We opted to let the developer decide whether this operation should be carried out or not. We introduced a `coroutineScope` field to replace the `launchedTasks` data structure so as to tie any task to the `MonitoredComponent` instance and ensure that, upon the destruction of the component, all tied tasks be terminated.
The `recommendedDispatchers` data structure, which existed for a similar reason, was instead preserved. The rationale is that everything related to dispatchers adds a layer of information that the `launchedTasks` did not deliver.

```kotlin
/**
 * Generates an id used for recognising a task throughout its lifecycle.
 */
fun buildTaskId(): Unit

/**
 * Generic method called before a task (be it launch or async) is started.
 */
fun beforeTask(context: CoroutineContext, taskId: String): CoroutineContext

/**
 * Method called after beforeTask when starting a task with the [launch] method;
 * handles any operations that are specific to launch but not [async].
 */
fun beforeLaunch(context: CoroutineContext, taskId: String): CoroutineContext

/**
 * Method called after beforeTask when starting a task with the [async] method;
 * handles any operations that are specific to launch and not [launch].
 */
fun beforeAsync(context: CoroutineContext, taskId: String): CoroutineContext

/**
 * Method for handling a CancellationException. These exceptions are special
 * cases in coroutines and may need more specific behaviour.
 */
fun handleCancellation(
    cancellationException: CancellationException,
    taskId: String
): Nothing

/**
 * Method for handling a generic Exception.
 */
fun handleException(exception: Exception, taskId: String): Nothing

/**
 * Method called when the [task] has been started
 * with either [launch] or [async].
 */
fun afterTaskStart(task: Job, taskId: String): Unit

/**
 * Method called when the task identified by [taskId]
 * is about to be terminated.
 */
fun afterTaskEnd(taskId: String): Unit
```

Listing 24: Signatures of the new methods replacing the old API.

The interface's exposed methods were refactored into the ones shown in Listing 24. As mentioned earlier, in the current implementation of `MonitoredComponent` the following applies:

- the method `beforeTask` checks for a recommended dispatcher and adds it to the context if it exists in the `recommendedDispatchers` map;
- the method `beforeLaunch` saves a `WrongDispatcherExceptionHandler` inside the context unless one is already present;
- the method `beforeAsync` performs no operations on the context since none of the properties relevant to us require it;
- the methods `afterTaskStart/End` do nothing, since the `launchedTasks` data structure has been removed.

The methods `handleException` and `handleCancellation` use the `Nothing` return type [36], thus forcing the user to throw an exception. This ensures that no exceptional behaviour is left unchecked.
All methods feature now an extra argument in `taskId` (with the obvious exception of `buildTaskId`), making it easier to keep track of a certain task throughout all the controls performed before and after its execution.

4.2.2.1 The New `MonitoredActivity` and `MonitoredViewModel` Classes

Since the `MonitoredComponent` now required a `coroutineScope`, we needed to implement this for both subclasses. Fortunately, the *AndroidX Lifecycle* library provided everything we needed:

- the `MonitoredActivity`'s scope is the `lifecycleScope`;
- the `MonitoredViewModel`'s is the `viewModelScope`.

This allowed us to effectively tie each component to its intended `CoroutineScope`.
A top-level declaration for a coroutine using this API would now look like

```
component.coroutineScope.launch(component) { ... }
```

which we felt looked a little too long, with the added redundancy of `component` being used both as receiver and argument. We defined thus a new object, called `MonitoredScope`, to act as syntactic sugar: as can be seen in Listing 25, calling a coroutine builder on this object yields the same result as the invocation above.
It should be noted that `MonitoredScope` is **not** a subclass of `CoroutineScope` but rather a singleton that follows the same naming convention.
As a final modification, we added an internal check in the `beforeTask` method of the `MonitoredViewModel` class: since it is the first method to be called during the execution of either `launch` or `async`, it now checks whether the `MonitoredViewModel` has been initialised and, if not, it proceeds to call the `init` method before continuing.

This final version of our API was employed to run our tests with regards to performance overhead and memory footprint. The next chapter will cover our tests and what results they yielded.

```kotlin
/**
 * Empty object that serves as syntactic sugar for calling an
 * instrumented [launch] or [async] on a MonitoredComponent
 * using the component's own scope.
 */
object MonitoredScope

fun <ComponentType : MonitoredComponent> MonitoredScope.launch(
    component: ComponentType,
    context: CoroutineContext = EmptyCoroutineContext,
    start: CoroutineStart = CoroutineStart.DEFAULT,
    block: suspend CoroutineScope.() -> Unit
) = component.coroutineScope.launch(component, context, start, block)

fun <ComponentType : MonitoredComponent> MonitoredScope.async(
    component: ComponentType,
    context: CoroutineContext = EmptyCoroutineContext,
    start: CoroutineStart = CoroutineStart.DEFAULT,
    block: suspend CoroutineScope.() -> Unit
) = component.coroutineScope.async(component, context, start, block)
```

Listing 25: Implementation of the `MonitoredScope` object and its two extension functions as short-hands for invoking the method `launch` or `async` on the component's `coroutineScope` field.

5

Experimental Validation

After finalising our API we needed to verify the correctness of its behaviour: this meant ensuring that it would work and also that it would not compromise the execution of the target app either by introducing heavy performance loss or by taking up too much memory.
In this chapter, we go through the results of our validation, detailing what we did and what behaviour we recorded. Section 5.1 describes the app that we used to test the behaviour of our API, section 5.2 describes what tests we ran with regard to performance overhead and section 5.3 describes the tests we carried out on what impact our API had on the allocation of objects in the application memory. In the next chapter, we will draw our final conclusions on the project.

5.1 Testing the API on a Proof-of-concept App

We stated earlier that we would eventually look for a candidate app to run our tests on and, should we find nothing suitable, we would have to create our own Android application.
As we could not find a suitable target application, we expanded upon the initial app developed in-house (mentioned in section 3.2) and built a proof-of-concept to run our tests.
The previous iteration of the app only had a single activity and all instrumentation was carried out in the viewmodel. In this new version, we made both activity and viewmodel implement the `MonitoredComponent` interface by extending, respectively, the `MonitoredActivity` and `MonitoredViewModel` classes.
For the newly defined `BrowsePicturesActivity` class we added a `defaultHandler` that would simply log any exception, while for the class `BrowsePicturesViewModel` we defined a handler that would display the exception on the main page.
Inside the viewmodel we used coroutines to invoke the webservice, receive its bytes and parse them into a custom data class; inside the activity we defined a custom `RecyclerViewAdapter` that would download bytes from the `link` field of each `Image` object, convert the bytes to a bitmap picture and apply that picture to the layout on screen.
For both cases we purposefully employed risky operations such as:

- instantiating a new `CoroutineContext` instead of using the one defined in the AndroidX Lifecycle library, with the risk of having any coroutine survive after the related component being destroyed (a clear breach of the **Destroyed-WithOwner** property);

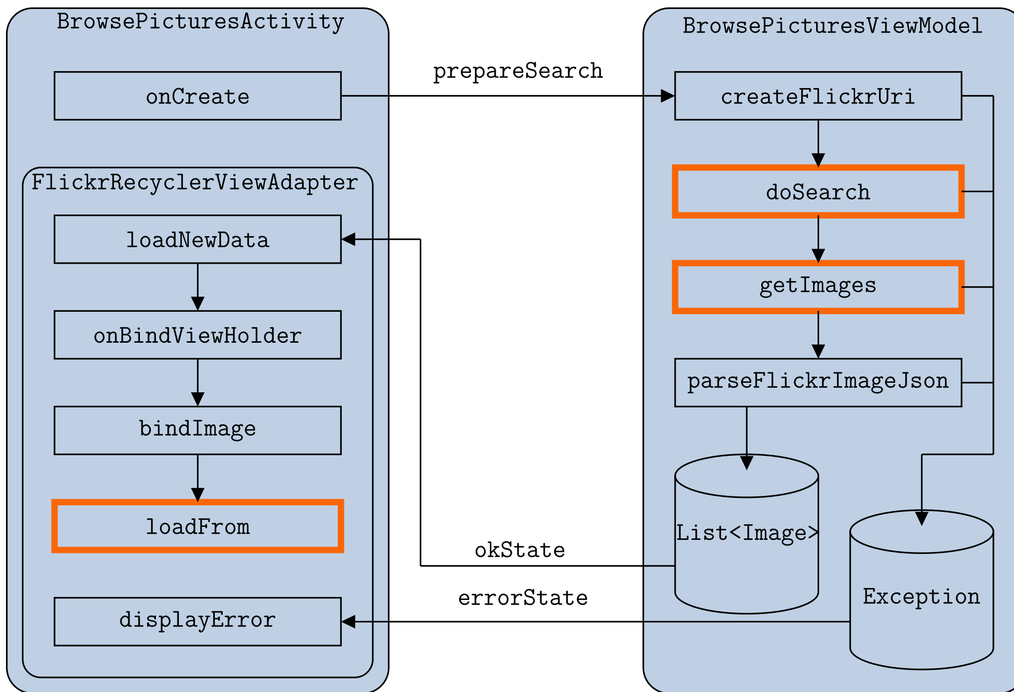

Figure 5.1: Simplified call diagram for the proof-of-concept app. Each rounded rectangle represents a class and each rectangle inside it represents its methods, with the orange rectangles being the one employing coroutines and, therefore, that we were interested in monitoring. The arrows represent the call flow starting from the activity. Note that inside the `BrowsePicturesViewModel` class there is a second flow that ends in error and updates the UI accordingly.

- using a random dispatcher whenever starting a coroutine, so as to leave the target thread for each task entirely up to chance: this was, of course, an open violation of the two properties **UpdateUI** and **NoBlockUI**.

Of course, the rest of the app was built following Android guidelines when coroutines were not involved, so as to isolate coroutine-related failures.

5.1.1 Test results

We ran our tests in the following way:

1. we started the app, entered some tags and started the search task;
2. as the results page loaded, we scrolled downward until the app had displayed all pictures;
3. we flipped the device so as to destroy the activity and repeat the search task;
4. we repeated points 2 and 3;
5. we navigated back to the first page so as to destroy the viewmodel;
6. we repeated points 1 through 5 taking notes of any errors we might run into.

The procedure described above was executed over twenty times using different combinations of tags to ensure that our app would be made to load images of varying sizes.

One of the first issues we had was with the launch of coroutines on a ran-

dom dispatcher: while we had defined the `WrongDispatcherExceptionHandler` class to only handle exceptions of type `NetworkOnMainThreadException` or `CalledFromWrongThreadException`, **we were receiving a third kind of exception** caused by the employment of the "wrong" thread:

```
java.lang.IllegalStateException:
Cannot invoke setValue on a background thread
```

This exception was triggered whenever we would try and update the viewmodel's `LiveData` fields from a background thread. This data is the means by which the viewmodel updates the UI: by observing the `LiveData` objects, the activity is aware of any new data being computed inside the viewmodel. The failure was caused by a `LiveData` object being updated from a background thread, thus preventing the activity from detecting the update.

While we could have simply "added" this exception to our list of failures handled by the `WrongDispatcherExceptionHandler`, we decided against it for the reason that the exception was of type `IllegalStateException` (which is a generic exception thrown when a method is invoked at an illegal or inappropriate time [66], and not an exception specific to this kind of behaviour).

On one hand we did detect any `CalledFromWrongThreadExceptions` based on their class name, so we could adopt a similar strategy and detect any such failures from their message "Cannot invoke setValue on a background thread" but, on the other hand, this course of action would have felt more like a last-minute solution than an actual enhancement of our API: we also had no idea whether any more exception could be triggered in a similar way, so we decided to leave this issue as a future case of study.

Any instances of the other two exceptions caused by an inappropriate dispatcher were, however, handled efficiently: as soon as we fixed the above issue by enforcing the update of `LiveData` on the UI thread, we started seeing a pattern of **no exceptions being lifted after upwards of four search operations**. An inspection of the `recommendedDispatchers` data structure for the `MonitoredApplication` instance revealed that several tasks had been associated with a preferred dispatcher:

```
com.android.rv.properties.BrowsePicturesViewModel.doSearch:44 =>
    LimitingDispatcher@1ef4893[dispatcher = DefaultDispatcher]
com.android.rv.ViewKt.loadFrom:48 =>
    Main
com.android.rv.ViewKt$loadFrom$2$1.invokeSuspend:52 =>
    LimitingDispatcher@1ef4893[dispatcher = DefaultDispatcher]
com.android.rv.properties.BrowsePicturesViewModel$getImages$2
    .invokeSuspend:49 =>
    LimitingDispatcher@1ef4893[dispatcher = DefaultDispatcher]
```

These entries identified four invocations of coroutine builders, whose code snippets can be seen in Listing 26. The entries can be described as follows:

- the first one referred to the method `doSearch` which, at line 44, invoked the `launch` with a random dispatcher;
- in turn, the `getImages` method at line 49 (the fourth entry in our structure) was trying to read bytes from a remote endpoint;
- the remaining two entries referred to the method `loadFrom`: this method starts

File *BrowsePicturesViewModel.kt*

```
44 CoroutineScope(randomDispatcher()).launch(this) {
45     getImages(uri, settings)
46 }
```

```
49 async(this@BrowsePicturesViewModel) {
50     URL(uri).readText()
51 }
```

File *View.kt*

```
48 CoroutineScope(randomDispatcher()).launch(component) {
49     setPlaceholder()
50     async(component, randomDispatcher()) {
51         BitmapFactory.decodeStream(URL(url).openStream())
52     }.let { setImageBitmap(it.await()) }
53 }
```

Listing 26: Snippets of the code referenced by the `recommendedDispatchers` data structure upon inspection.

```
1 launch(component) {
2     setPlaceholder()
3     setImageBitmap(
4         BitmapFactory.decodeStream(URL(url).openStream())
5     )
6 }
```

Listing 27: An example of a coroutine that bypasses our monitoring

a task at line 48 to update the UI with a bitmap which it then reads at line 52 from a remote URL.

As can be seen in Figure 5.1, all intercepted coroutine builders are called inside the methods we had instrumented.

Another weakness we discovered in our implementation was that using an I/O operation and a UI update in the same task would bypass our monitors; a block of code written as shown in Listing 27 would generate a false negative. In the example code, the monitors would first save `Dispatchers.Default` as the recommended dispatcher due to the I/O operation on line 4 but then the coroutine would fail due to the UI update on line 3 requiring `Dispatchers.Main`.

This told us that we needed to consider a third case in our handling of the properties **UpdateUI** and **NoBlockUI**: the case in which the recommended dispatcher was still faulty. As already mentioned for the `IllegalStateException` occurrence, we needed to strengthen our handling of the "correct dispatcher" behaviour.

Aside from the above problem the `WrongDispatcherExceptionHandler` worked, as **we always received a `WrongDispatcherException` after any dispatcher-related failures**, even in the case of starting a coroutine without any handler, a sign that our `beforeLaunch` and `beforeTask` methods were effectively monitoring our

```
CoroutineScope(randomDispatcher()).launch(viewModel) {
    launch(viewModel, randomDispatcher()) {
        launch(viewModel, randomDispatcher()) {
            // ...
        }
    }
}
```

Listing 28: An example of nested coroutine builders, each using a randomised dispatcher and lacking an exception handler.

invocations. This increased our confidence that, while we did run into unforeseen issues, the overall behaviour of our API was compliant to our expectations.

5.2 Benchmarking the Application

After finalising our API we proceeded to run tests to verify its performance and its impact on the device memory. Our first test involved running benchmarks.
We opted to run four different tests:

1. a `launch` test: this would spawn one task using the `launch` methods (the standard Kotlin implementation and our version involving the `MonitoredComponent`) and wait for its completion;
2. an `async` test: same as the above test but using the `async` methods instead;
3. a second `async` test that would instead spawn a set number of tasks in parallel, then wait for their completion;
4. a simulation of the standard execution of the `ViewModel` for the POC app, which means downloading bitmap images from the same Flickr API used in the application.

For tests 1 through 3 we decided to employ a task called `runSomething` that would only suspend the current coroutine for 100 milliseconds. This allowed us to pinpoint what the expected duration should be for the execution of the task and how much our API would extend it.
We prepared a second activity, called `BrowsePicturesUnmonitoredActivity`, complete with matching `ViewModel`, that would carry out the same operations as the `BrowsePicturesActivity` but without using our API. This means that:

- the class `BrowsePicturesUnmonitoredActivity` extends the class `AppCompatActivity` rather than `MonitoredActivity`;
- the class `BrowsePicturesUnmonitoredViewModel`, in a similar fashion, extends the class `ViewModel` rather than `MonitoredViewModel`.

For the benchmarks involving the methods `launch` and `async` we spawned our tasks depending on the presence, or absence, of monitors.
For the non-monitored version, we started the task using the `lifecycleScope` field defined for the `Activity` class, using an empty `CoroutineContext`. This means that we had no handlers or dispatchers. There was no actual need for either object since:

Table 5.1: Technical specifications of both devices employed in our benchmarks.

Model name	**CPU**	**RAM**	**OS**
Huawei ATU-L21	quad core 1.4 GHz	2 GB	Android 8.0.0 Oreo
Sony H8324	octa core 4x 2.7 GHz 4x 1.7 GHz	4 GB	Android 10

1. the `runSomething` function did not have any exceptional behaviour;
2. no UI or I/O operations were involved, making the use of a specific dispatcher unnecessary.

For the monitored version, we used the `MonitoredScope` syntax mentioned in section 4.2.2.1, which is syntactic sugar for calling the instrumented `launch` (or `async`) method in the `coroutineScope` field defined for the `MonitoredActivity` class. Just like for the non-monitored test, we defined no handlers or dispatchers; in this case, however, our API would proceed to add them automatically.
We ran our benchmarks on two different smartphones, a lower-end and a medium-end one. The former was a Huawei Y6 ATU-L21 and the latter a Sony Xperia XZ2 Compact H8324, both produced in the year 2018; more details about each of them can be seen in table 5.1.

5.2.1 Benchmarking Tool

For our tests we used the benchmarking tools provided by the Android Jetpack library [67]: they launch a dedicated activity that uses the main module as a library. The activity runs all the tests defined by the user while maintaining foreground state on the target device, ensuring that the Android OS does not allocate resources elsewhere.
We defined four test classes, one for each test case. Each test class contained two methods, one for benchmarking the “standard”, non-instrumented behaviour and one benchmarking the monitored behaviour.
The Jetpack library provides a `BenchmarkRule` type that allows us to define which blocks of code need to be measured and which are to be skipped: namely, any code inside the `BenchmarkRule.measureRepeated` method will be tracked. Using this tool we managed to define exactly what we wanted to measure.

5.2.2 Results

Each of the benchmarks was executed ten times. For each test we identified the slowest and fastest runs and computed the average running time among the ten values we had recorded. We subtracted the average non-monitored time from the average monitored time to determine the average overhead.
Table 5.2 contains the results for the **first benchmark**. The values are expressed in nanoseconds which is the standard measure unit for the Android Jetpack benchmarking tool.
Since $1ms = 1,000,000ns$, we can see that the $100ms$ delay comprises most of the execution time and the average overhead is of roughly half a millisecond.

Table 5.2: Benchmark results for the `launch` method listing the fastest, slowest and average times for the "standard" version and the instrumented one, with the overheads in the rightmost columns. All values are in nanoseconds (*ns*) save for the last column.

Model	Launch							
	No monitors			Monitored			Overhead	
	Fastest	Slowest	Average	Fastest	Slowest	Average	Average	%
Huawei	100,479,698	100,940,948	100,792,120	101,019,801	101,482,979	101,257,927	465,807	0.46
Sony	101,002,761	103,085,416	102,199,974	103,283,125	105,066,615	103,833,698	1,633,724	1.60

Table 5.3: Benchmark results for the `async` method listing the fastest, slowest and average times for the "standard" version and the instrumented one, with the overheads in the rightmost columns. All values are in nanoseconds (*ns*) save for the last column.

Model	Async (single call)							
	No monitors			Monitored			Overhead	
	Fastest	Slowest	Average	Fastest	Slowest	Average	Average	%
Huawei	100,457,458	101,086,469	100,817,698	101,050,322	101,426,208	100,817,698	451,537	0.45
Sony	102,000,156	103,150,000	102,611,005	101,629,218	104,671,406	103,490,036	879,031	0.86

Table 5.3 contains the results for the **second benchmark**, the one involving the `async` method being called only once. Its results are similar to the ones for the `launch` method as far as the Huawei device is concerned; the Sony device presents a `launch` overhead almost twice as big as the `async` overhead. We suspected that this difference may have to do with the `beforeLaunch` method taking longer than the `beforeAsync`, so we performed one more round of benchmarks where we altered the `beforeLaunch` method so as to not do anything. For this round of test we only considered the Sony device and obtained the following results:

- for the non-monitored `launch` method, a fastest running time of $101,375,885ns$, a slowest of $104,816,198ns$ with an average of $102,620,823ns$;
- for the monitored version, a fastest running time of $101,902,761ns$, a slowest of $104,371,771ns$ with an average of $103,435,630ns$;
- an average overhead of $814,807ns$ (equal to a 0.99% increase over the non-monitored average time), much closer to the `async` overhead of $879,031ns$.

We assume therefore that the Sony device is simply not as performant as the Huawei device for this operation.
Table 5.4 shows the results for the **third benchmark**. The same `runSomething`

Table 5.4: Benchmark results for the `async` method, called a thousand times in parallel. The table lists the fastest, slowest and average times for the "standard" version and the instrumented one, with the overheads in the rightmost columns. All values are in nanoseconds (*ns*) save for the last column.

Model	Async (massive)							
	No monitors			Monitored			Overhead	
	Fastest	Slowest	Average	Fastest	Slowest	Average	Average	%
Huawei	140,391,941	152,338,870	144,162,770	339,490,920	361,831,599	352,463,374	208,300,604	144.49
Sony	116,279,011	123,808,594	119,748,896	180,976,042	193,502,240	187,324,891	67,575,995	56.43

```
viewModel.viewModelScope.launch(Dispatchers.IO + handler) {
    // Read JSON data from Flickr
    val json = URL(flickrUri).readText()
    val parsedData = parseFlickrImageJson(json, false)
    // Get each picture as a bitmap
    parsedData.map {
        it.getOrThrow().link.let {
            async {
                BitmapFactory.decodeStream(URL(it).openStream())
            }
        }
    }.awaitAll()
}
```

Listing 29: Implementation of the test for simulating the non-monitored execution.

function was called by one thousand `async` tasks running in parallel with the following results:

- due to the tasks running in parallel, the total execution time for the non-monitored method is not much longer compared to the single call: we have a 43% increase on the Huawei device and a 16% increase on the Sony device;
- the execution time for the monitored method displays a dramatic increase on the Huawei device compared to the single call: the duration shows an increase of over 3 times; this becomes a tamer 81% increase on the Sony device, most likely due to its greater amount of CPUs.

For the **fourth benchmark** we used the matching `ViewModel` for either activity. The code used for testing can be seen on Listing 29 for the non-monitored simulation and on Listing 30 for the monitored one. The latter is quite similar to the former, with a couple of differences:

- the coroutine builder used on lines 1 and 8 is the standard one for the first test and the instrumented one for the second;
- since the code invokes an external web service, which may trigger an exception, on line 1 the non-monitored test includes a `CoroutineExceptionHandler` defined outside of the benchmark function;
- always on line 1 both tasks are spawned on the `IO` thread dispatcher, since the operation is entirely I/O based.

The variable `flickrUri` on line 3 for both code snippets is a pre-generated `String` containing a search request for popular tag ("lamborghini") to ensure that the webservice retrieves the highest number of pictures. This number is 20 for the webservice we use, so every invocation will return a JSON containing 20 entries, each with its own image that will be downloaded as a stream of bytes by an asynchronous task.
Table 5.5 shows that, in the absence of a delay function, the difference between the instrumented and non-instrumented implementation is dramatic, with an increase of almost 7 times (695.69%) on the Huawei device and of 18 times (1,799.45%) on the Sony device.

```
MonitoredScope.launch(viewModel, Dispatchers.IO) {
    // Read JSON data from Flickr
    val json = URL(flickrUri).readText()
    val parsedData = parseFlickrImageJson(json, false)
    // Get each picture as a bitmap
    parsedData.map {
        it.getOrThrow().link.let {
            async(viewModel) {
                BitmapFactory.decodeStream(URL(it).openStream())
            }
        }
    }.awaitAll()
}
```

Listing 30: Implementation of the test for simulating the monitored execution.

Table 5.5: Benchmark results for the `ViewModel` simulation listing the average time for the "standard" version and the instrumented one, with the overheads in the rightmost column. All values are in nanoseconds (ns).

Model	Simulation							
	No monitors			Monitored			Overhead	
	Fastest	Slowest	Average	Fastest	Slowest	Average	Average	%
Huawei	13,742	82,218	30,427	203,463	325,886	272,534	242,107	795.00
Sony	1,626	5,770	2,914	53,921	57,083	55,351	52,436	1799.22

5.3 Memory Footprint

Android Studio comes with a built-in profiler allowing us to monitor the memory usage of the application that is currently running. We used this tool to check the memory footprint of our `MonitoredApplication` containing a non-empty map of recommended dispatchers.
Our course of action was to start the proof-of-concept app using the standard Android `Application` instance, measure its overall size in memory, reinstall the app with the `MonitoredApplication` enabled and measure its size after having saved some entries into its `recommendedDispatchers` structure.
The results can be seen in Table 5.6 and they mark an expected increase in the memory footprint. With each entry containing only `String` instances and `Dispatcher` references, we assume that each entry will take approximately 150-200 bytes; the space is proportional to the class name for the `MonitoredComponent` subclass as well as the combined length of class, method name and line number for the registered task.
The registered size was of course quite small for our simplistic app but it would definitely increase in a bigger one. Furthermore, we did not like how the class names were directly responsible for the size of each entry; an alternative way of recording tasks could definitely solve this issue.

Table 5.6: Memory reserved by the `Application` instance, not instrumented (second column) and then instrumented (third and fourth columns). All values are in bytes.

Model	**Application**	**MonitoredApplication**	
		Empty	**Two entries**
Huawei	128	172	413
Sony	128	172	544

6
Conclusion

After our tests, we draw our conclusions and evaluate the project in this chapter. In section 6.1 we discuss what we gained from this project as well as its impact on the state of the art and, lastly, in section 6.2 we address the limitations of the project and what could be done in the future to correct them.

6.1 Considerations

Our overall judgement of the API is that it is an interesting starting point for the introduction of RV on an Android development platform but, at the same time, it requires a way for the user to define any number of properties and monitor them without translating them manually to Kotlin code.
Our approach in making a syntactically similar framework to AspectJ made it easier for us to implement monitors and have them observe specific events within the execution of the app.
We can use our experience with this project to provide answers to our research questions from section 1.2.

What are the meaningful properties of Kotlin coroutines that we can verify by employing RV?

We identified a handful of properties that have to do with creation and termination of coroutines at runtime. This number will likely rise as the currently experimental coroutine functionalities (e.g. the builder methods `actor` and `produce` mentioned in section 2.2.4.1) are given a stable implementation. The properties related to what dispatcher is associated with a certain task are also certainly interesting.

Can we use the Kotlin language to monitor properties of the more specific Kotlin features?

We could define an API close to AspectJ in its purpose but we couldn't achieve a coverage as wide as we would have wanted. We will elaborate more on this in the section below.

How can we push further the current situation for RV of Android applications by using Kotlin?

We feel that there is definitely something to be gained from applying Runtime Verification to an app still in development: this can be used to detect unexpected behaviour, especially when originated from language-specific features with which the developer may not be familiar.
Kotlin has recently been adopted as a main language for Android development, which means its user base is likely going to see a marked increment in the coming years: the monitoring of its unique features, while not necessarily relevant for released apps (as explained at the start of Section 3), can be of assistance when migrating old Java code or refactoring classes using an outdated approach.
The concept of applying Runtime Verification to an application during its development phase, as opposed to after its release, doesn't seem to have been explored in the past. Our API made it possible to monitor the behaviour of an Android app from an early stage and we believe that this approach, if adopted at an industrial level, will come with two advantages:

- making developers more confident in fully using the features of Kotlin when writing applications;
- making RV more widespread in the industry, growing from a tool used after release to one also employed from the start to the end of the development process.

In this project we see a potential to grow the scope of Runtime Verification and push its usage to multiple design stages of an Android application.

6.2 Perspectives and Future Work

There are several changes that could be made to enhance this final API even further but that we opted not to do as we did not feel the need to carry them out at this time.
Like we mentioned earlier, during our final redesign we tried to make our API more similar (at least in name) to AspectJ: we now have several "before" and "after" methods and, to provide a more complete range of methods, we would need:

- some sort of "around" method that decides whether a given task should be started or not and, in the latter case, what should be returned instead;
- an "afterReturning" method for the `async` tasks: this should be executed after the given block of code provided that no exception was thrown.

These methods could prove an asset to a future version of the monitoring API.
Another enhancement would be a rework of the `MonitoredApplication` class: as of now it only keeps track of the `recommendedDispatchers` maps defined for each instance of `MonitoredComponent` and its methods are very transparent about it. We feel that the class should be given a treatment similar to the `MonitoredComponent` interface with a rehaul of its exposed methods and the opportunity to save a more varied range of information. Of course, such a modification would need to be tested for its impact on performance and memory consumption.

As emerged from our test of the memory footprint (see section 5.3), the current way of saving records of dispatcher data employs the class name of each

`MonitoredComponent` instance. This makes the combined length of class and package name influence the amount of memory occupied by our application and its data structures. The class name is, however, currently used by the `TerminationService` class to print a human-readable list of all recommended coroutine dispatchers grouped by their lifecycle components; we need therefore to find an alternate way to store records of each component without using their class name but at the same time keeping them recognisable by a human programmer.

Despite all this, the main drawback in our implementation was the aforementioned need to define properties as blocks of Kotlin code. **Our API needs an interpreter to translate properties to Kotlin code** instead of forcing this task onto the developer: a solution would be the employment of an already existing specification language like DATEs [30] with the creation of an intermediate layer carrying out the parsing and interpretation.
Another important limitation of our tool is its coverage of the sole source code, since its instrumented versions of `launch` and `async` do not fully affect the code used in libraries.

Bibliography

[1] K. Havelund and A. Goldberg: Verify Your Runs. *Working Conference on Verified Software: Theories, Tools, and Experiments*, Springer 2005: pp 374-383

[2] M. Leucker and C. Schallhart: A brief account of runtime verification. *The Journal of Logic and Algebraic Programming* 78(5): pp 293-303 (2009)

[3] Y. Falcone, K. Havelund and G. Reger: A Tutorial on Runtime Verification. *Engineering Dependable Software Systems* 2013: pp 141-175

[4] E. Bartocci, Y. Falcone, A. Francalanza and G. Reger: Introduction to Runtime Verification. *Lectures on Runtime Verification* 2018: pp 1-33

[5] E. Bartocci and Y. Falcone: Lectures on Runtime Verification: Introductory and Advanced Topics. *Lecture Notes in Computer Science Vol. 10457*, Springer 2018, ISBN 978-3-319-75631-8

[6] Y. Falcone, S. Currea and M. Jaber: Runtime Verification and Enforcement for Android Applications with RV-Droid. *International Conference on Runtime Verification*, Springer 2012: pp 88-95

[7] H. Sun, A. Rosà, O. Javed and W. Binder: ADRENALIN-RV: Android Runtime Verification using Load-time Weaving. *IEEE International Conference on Software Testing, Verification and Validation*. IEEE 2017: pp 532-539

[8] Android 10's official home page. `https://www.android.com/android-10/`

[9] Kotlin on Android FAQ. `https://developer.android.com/kotlin/faq`

[10] J. Buttigieg, M. Vella and C. Colombo: BYOD for Android - Just add Java (2015)

[11] M. D'Amorim and K. Havelund: Event-Based Runtime Verification of Java Programs. *ACM SIGSOFT software engineering notes* 30.4 (2005): pp 1-7

[12] AspectJ's reference page on the Eclipse Foundation's website. `https://www.eclipse.org/aspectj`

[13] P.Daian, Y. Falcone, P. Meredith, T. F. Şerbănuţă, S. Shiriashi, A. Iwai, G. Rosu: RV-Android: Efficient Parametric Android Runtime Verification, a Brief Tutorial. *Runtime Verification*, Springer 2015: pp 342-357

[14] A. R. Hevner, S. T. March, J. Park and S. Ram: Design science in information systems research. *MIS quarterly* 2004: pp 75-105

[15] dex2jar GitHub repository. `https://github.com/pxb1988/dex2jar`

[16] M. E. Conway: Design of a Separable Transition-diagram Compiler. *Communications of the ACM* 6.7 (1963): pp 396-408

[17] Definition of a callback function, answer on StackOverflow. `https://stackoverflow.com/questions/9596276#9652434`

[18] "Managing exceptions in nested coroutine scopes", blog post. `https://proandroiddev.com/9f23fd85e61`

[19] Opera Software's website. https://www.opera.com/about
[20] Opera Touch's homepage. https://www.opera.com/mobile/touch
[21] Opera News application page on Google's Play Store. https://play.google.com/store/apps/details?id=com.opera.app.news
[22] OPay's homepage. https://operapay.com/
[23] Short explanation of how the AspectJ compiler works on StackOverflow. https://stackoverflow.com/a/41980558/4074036
[24] "Introduction to Android", snapshot from 21st of March, 2017. https://web.archive.org/web/20170321155658/https://developer.android.com/guide/index.html
[25] A. Bauer, J. C. Küster and G. Vegliach: Runtime Verification meets Android Security. *NASA Formal Methods Symposium*, Springer 2012: pp 174-180
[26] A. Francalanza, A. Gauci, G. J. Pace: Distributed system contract monitoring. *The Journal of Logic and Algebraic Programming* 82.5-7 (2013): pp 186-215
[27] L. Chircop, C. Colombo and G. J. Pace: Device-Centric Monitoring for Mobile Device Management. *EPTCS* 205, 2016: pp 31-44
[28] W. Zhou, O. Sokolsky, B. T. Loo, I. Lee: DMaC: Distributed Monitoring and Checking. *International Workshop on Runtime Verification*, Springer 2009: pp 184-201
[29] A. Villazón, H. Sun, W. Binder: Capturing Inter-process Communication for Runtime Verification on Android. *International Symposium on Leveraging Applications of Formal Methods*, Springer 2018: pp 25-31
[30] C. Colombo, G. J. Pace and G. Schneider: LARVA— Safer Monitoring of Real-Time Java Programs (Tool Paper). *2009 Seventh IEEE International Conference on Software Engineering and Formal Methods*. IEEE 2009: pp 33-37
[31] "Null References: The Billion Dollar Mistake", talk held in 2009 by Tony Hoare about null references. https://www.infoq.com/presentations/Null-References-The-Billion-Dollar-Mistake-Tony-Hoare/
[32] "The Trouble with Checked Exceptions", transcript of a discussion between Anders Hejlsberg, Bill Venners and Bruce Eckel. https://www.artima.com/intv/handcuffs.html
[33] "Java's checked exceptions were a mistake (and here's what I would like to do about it)", blog post by Rod Waldhoff. http://radio-weblogs.com/0122027/stories/2003/04/01/JavasCheckedExceptionsWereAMistake.html
[34] "Bruce Eckel is Wrong", blog post by Elliotte Rusty Harold. https://cafe.elharo.com/programming/bruce-eckel-is-wrong/
[35] "The case against checked exceptions", question on StackOverflow. https://stackoverflow.com/questions/613954/the-case-against-checked-exceptions
[36] Kotlin documentation for the Nothing type. https://kotlinlang.org/docs/reference/exceptions.html#the-nothing-type
[37] Kotlin tutorial for throwing and catching exceptions. https://kotlinlang.org/docs/tutorials/kotlin-for-py/exceptions.html#throwing-and-catching
[38] Documentation for the `Either` data type in Arrow, external library for Kotlin. https://arrow-kt.io/docs/arrow/core/either/

[39] Documentation for the `Option` data type in Arrow, external library for Kotlin. https://arrow-kt.io/docs/arrow/core/option/
[40] Documentation for the `Try` data type in Arrow, external library for Kotlin. https://arrow-kt.io/docs/arrow/core/try/
[41] Procyon metaprogramming Java tool, repository on BitBucket. https://bitbucket.org/mstrobel/procyon
[42] Blog post by Martin Sústrik detailing structured concurrency. http://250bpm.com/blog:71
[43] Debug mode for the coroutines Kotlin library. https://github.com/Kotlin/kotlinx.coroutines/blob/master/docs/debugging.md#debugging-coroutines
[44] Debug agent for Kotlin coroutines on Android. https://github.com/Kotlin/kotlinx.coroutines/blob/master/docs/debugging.md#debug-agent-and-android
[45] Android developers guide, "Improve app performance with Kotlin coroutines". https://developer.android.com/kotlin/coroutines
[46] Android developers guide, "Use Kotlin coroutines with Architecture components". https://developer.android.com/topic/libraries/architecture/coroutines
[47] Blog post "Why I stopped using Coroutines in Kotlin". https://dev.to/martinhaeusler/why-i-stopped-using-coroutines-in-kotlin-kg0
[48] Blog post "Why the Kotlin/Native memory model cannot hold." https://itnext.io/why-the-kotlin-native-memory-model-cannot-hold-ae1631d80cf6
[49] `kotlinx.coroutines` issue #462: Support multi-threaded coroutines on Kotlin/Native. https://github.com/Kotlin/kotlinx.coroutines/issues/462
[50] Kotlin documentation for platform types. https://kotlinlang.org/docs/reference/java-interop.html#null-safety-and-platform-types
[51] Blog post "Coroutines in pure Java". https://medium.com/@esocogmbh/coroutines-in-pure-java-65661a379c85
[52] Flickr home page. https://www.flickr.com/
[53] Home page for the Flickr feed services. https://www.flickr.com/services/feeds/
[54] Model-View-Viewmodel architecture on Android Jetpack's documentation. https://developer.android.com/jetpack/docs/guide#recommended-app-arch
[55] Blog post "AOP vs functions" by Roman Elizarov, Team Lead at JetBrains. https://medium.com/@elizarov/aop-vs-functions-2dc66ae3c260
[56] Documentation for the `CoroutineScope.launch` coroutine builder method. https://kotlin.github.io/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/launch.html
[57] Documentation for the `CoroutineScope.async` coroutine builder method. https://kotlin.github.io/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/async.html

[58] Description of the AndroidX Lifecycle library, containing lifecycle-aware structures and coroutine scopes. https://developer.android.com/jetpack/androidx/releases/lifecycle
[59] Overview of Android's threading model. https://developer.android.com/guide/components/processes-and-threads#Threads
[60] Blog post "Advanced Kotlin Coroutines tips and tricks". https://proandroiddev.com/coroutines-snags-6bf6fb53a3d1
[61] Approaches to making Kotlin computation code in coroutines cancellable. https://kotlinlang.org/docs/reference/coroutines/cancellation-and-timeouts.html#making-computation-code-cancellable
[62] Configuration changes on Android and their impact on activities. https://developer.android.com/guide/components/activities/state-changes#cco
[63] Description of the lifecycle of an Android activity. https://developer.android.com/guide/components/activities/activity-lifecycle#alc
[64] "Overloading" in computer programming. https://en.wikibooks.org/wiki/Computer_Programming/Function_overloading
[65] Definition of sealed class in the official Kotlin documentation. https://kotlinlang.org/docs/reference/sealed-classes.html
[66] Documentation for the IllegalStateException class. https://docs.oracle.com/javase/8/docs/api/java/lang/IllegalStateException.html
[67] Introduction and quick start guide for the Android Jetpack benchmarking tools. https://developer.android.com/studio/profile/benchmark