# Static Deadlock Analysis for Kotlin Coroutines

*Version of October 10, 2024*

Bob Brockbernd

# Static Deadlock Analysis for Kotlin Coroutines

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Bob Brockbernd
born in Zeist, the Netherlands

Software Engineering Research Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
`www.ewi.tudelft.nl`

JetBrains
Concurrent Computing Lab
Gelrestraat 16
Amsterdam, the Netherlands
`www.jetbrains.com`

©2024 Bob Brockbernd. All rights reserved.

# Static Deadlock Analysis for Kotlin Coroutines

Author: Bob Brockbernd
Student id: 4594665

**Abstract**

Asynchronous programming is often a difficult and non-trivial task. To make asynchronous programming more straightforward, languages are continuously introducing new syntax and patterns, making it easier to think about and develop solutions for concurrent problems. JetBrains introduced coroutines for Kotlin in 2018. Although Kotlin coroutines promise safe execution and concise code, it is not immune to concurrency bugs such as deadlocks. Particular *runBlocking* deadlocks are common when working with Kotlin coroutines. While other languages have made various advancements in detecting deadlocks, Kotlin lags behind.

In this work, we present two static analysis techniques that help developers detect and prevent deadlocks. The first technique, focused on the *runBlocking* problem, successfully identified dangerous patterns in open source repositories, leading to their resolution. Additionally, this technique has been integrated into JetBrains flagship IDE: IntelliJ IDEA. The second technique, aimed at general deadlock detection, has been developed and tested as a prototype. By using existing modeling techniques combined with novel approaches we have been able to accurately predict deadlocks in a controlled environment. Overall, this study tackled a common problem in Kotlin coroutines and made the first steps toward general deadlock detection.

Thesis Committee:

| | |
|---|---|
| Chair: | Prof. Dr. A. van Deursen, Faculty EEMCS, TU Delft |
| University supervisor: | Prof. Dr. B. Ozkan, Faculty EEMCS, TU Delft |
| Company supervisor: | Drs. N. Koval, JetBrains |
| Committee Member: | Prof. Dr. C. Poulsen, Faculty EEMCS, TU Delft |

# Preface

I would like to thank Nikita and Burcu for their invaluable guidance throughout the thesis, for their confidence in my success and for their constructive criticism when I needed to hear it. Lastly, my endless gratitude goes out to Carlotta for taking care of me when I had to take care of this thesis.

Bob Brockbernd
Delft, the Netherlands
October 10, 2024

# Contents

**Preface** iii

**Contents** v

**List of Figures** vii

**1 Introduction** 1

**2 Related Work** 3
- 2.1 Deadlock detection for JVM . . . 3
- 2.2 Deadlock detection for C# . . . 4
- 2.3 Deadlock detection for Go . . . 4

**3 Background** 5
- 3.1 Coroutine builders . . . 5
- 3.2 Suspend function . . . 6
- 3.3 Coroutine scope . . . 7
- 3.4 Dispatchers . . . 7
- 3.5 Channels . . . 9

**4 RunBlocking analysis** 11
- 4.1 The runBlocking problem . . . 11
- 4.2 Solution . . . 14
- 4.3 Implementation of runBlocking analysis . . . 18
- 4.4 Validation . . . 19
- 4.5 Discussion and future work . . . 21

**5 Static Deadlock Detection** 23
- 5.1 Channel Deadlocks . . . 24
- 5.2 Expressing Kotlin in Promela . . . 25
- 5.3 Translating Kotlin to Promela . . . 32

5.4 SPIN model verification . . . . . . . . . . 37
5.5 Validation: Accuracy . . . . . . . . . . 38
5.6 Validation: Scalability . . . . . . . . . . 41
5.7 Discussion and Future work . . . . . . . . . . 42

**6 Conclusion** **43**

**Bibliography** **45**

**A Prompt** **49**

# List of Figures

4.1 Visualization of a deadlock when calling *runBlocking* from a coroutine dispatched in a single-threaded environment. . . . . . . . . . . . . . . . . . . . . . . 12
4.2 Call graph representing all coroutine builders and functions they can reach from Listing 4.4 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 15
4.3 Call graph representing the code in Listing 4.5. The *Included* area is the resulting Coroutine Call Graph. . . . . . . . . . . . . . . . . . . . . . . . . . . . . 17
4.4 What the end user would see for an analysis of Listing 4.4. . . . . . . . . . . . 19
4.5 How an end user would change the settings of the analysis. . . . . . . . . . . . 19

5.1 A visual illustration of the wait for relations that result in the deadlock of Listing 5.1. Each arrow shows what is waited for e.g. `ch1.send()` waits for `ch1.receive()`. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 24
5.2 Execution order of Listing 5.2 that does not lead to a deadlock. . . . . . . . . . 25
5.3 Execution order of Listing 5.2 leads to a deadlock. The last send in coroutine B will never be executed. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 25

# Chapter 1

# Introduction

Asynchronous programming is often a difficult and non-trivial task. Due to the non-deterministic nature of scheduling, it can introduce bugs that are difficult to detect and, thus, challenging to solve. To make asynchronous programming more straightforward, languages are continuously introducing new syntax and patterns, making it easier to think about and develop solutions for concurrent problems. Recently, JetBrains introduced coroutines for Kotlin (October 2018). Although Kotlin coroutines promise safer execution due to structured concurrency [30, 14], it is not immune to concurrency bugs. An empirical study [6] shows that blocking and non-blocking concurrency bugs are possible and occur in real-world repositories.

#### Research gap and Impact

Concurrency bugs are notoriously hard to detect and debug. Therefore, good tooling is required to help developers make better code and provide better understanding. Currently, existing tools can check for linearizability [20]. Additionally, race detection can be performed with platform-specific tools like DRD [31] for JVM. However, while other languages have made various advancements in detecting deadlocks (chapter 2), Kotlin lags behind.

According to recent work, [6] a certain *runBlocking* mistake which leads to deadlocks, is often made across multiple repositories. This shows that a study in deadlock detection for Kotlin is not only novel but might even be necessary. Unfortunately, the *runBlocking* deadlock differs significantly from the classical deadlocks. Therefore, in addition to studying the specific *runBlocking* problem, we believe that a more general technique capable of detecting communication deadlocks [13] and deadlocks with mutexes can benefit Kotlin as well. Granted that deadlocks do not occur as often in Kotlin as they occur in Go, it is important to note that Kotlin is growing in popularity and further adoption of coroutines in concurrent backends should be expected. Consequently, this study can help set up Kotlin for the future and allow developers to write concurrent code with more confidence.

### Research questions

This thesis will embark in two directions: tackle the urgent *runBlocking* problem, and make the first steps towards a general deadlock detection technique. In this study we aim to answer the following questions:

- What code pattern leads to the common *runBlocking* problem and how can it be detected?
- How can we accurately model and verify asynchronous behaviour of a Kotlin coroutine program, enabling the prediction of deadlocks?

### Scope and Goal

The scope of this work is in twofold. For the *runBlocking* detection we aim to provide an effective tool that can help developers solve the current problems. For the general deadlock detection we focus on creating an initial prototype that supports only one synchronisation primitive and basic language features. Proving that the proposed solution has potential and encouraging future research.

### Structure

This thesis will be presented as follows: the related deadlock detection techniques are discussed in chapter 2, a comprehensive introduction into Kotlin coroutines is given in chapter 3, the *runBlocking* part of the thesis is discussed in chapter 4, the general deadlock detection study is presented in chapter 5, and lastly we conclude in chapter 6.

# Chapter 2

# Related Work

In this chapter, we will discuss the deadlock detection techniques that are most related to Kotlin deadlock detection. Detection techniques for the JVM will be discussed, since that is Kotlin's main platform. But also techniques for C# and GO since these languages have similar concurrency primitives.

## 2.1 Deadlock detection for JVM

In this section, the existing deadlock detection techniques for JVM will be discussed. Since Kotlin is primarily a JVM language, one might expect that one of these techniques works for Kotlin coroutines. However, Kotlin's concurrency system is vastly different from the other JVM languages. All current available deadlock detection techniques for JVM focus on thread operations, while deadlocks in Kotlin can manifest on coroutine level.

Stalemate [29] and JADE [23] are static deadlock detection techniques that rely on the creation of lock order graphs. JADE only focuses on two-thread deadlocks but is more scalable than Stalemate. Both techniques, however, produce many false positives. Another downside of these techniques in the context of Kotlin coroutines is that these techniques focus on locking primitives that work on the thread level. Whereas Kotlin coroutines come with locking mechanisms that do not block the thread but work on coroutine level. Therefore, deadlocks created with these locks will stay under the radar.

INFER Starvation [7] is a recent development in the static deadlock detection field for JVM. It favours false negatives over false positives, aiming to make reported deadlocks more reliable. Furthermore, it addresses scalability by focussing on code changes instead of reanalysing the complete code base after each commit. However, just as with the previously discussed methods, it only focuses on thread level deadlocks.

Some dynamic analysis techniques that detect deadlocks in the JVM are Dl-check [18], Sherlock [15] and Omen [27]. Dl-check creates and maintains a lock order graph during runtime. It, however, has no control over the execution. Sherlock, on the other hand, attempts to steer execution towards a deadlock with concolic execution. Lastly, Omen synthesises specific tests that potentially lead to a deadlock; these tests are then executed with a dynamic deadlock detector to validate the tests.

Another work [26] proposes a technique where Stalemate [29] is used to find lock order violations. Which are then used to steer Omen [27] towards test generation that potentially trigger the deadlocks found by Stalemate.

## 2.2 Deadlock detection for C#

This section discusses deadlock detection for C#. Kotlin and C# are quite similar in how one deals with concurrency. Both languages provide async/await syntax in some form and use continuation passing style with function colouring.

DeadWait [28] is a static deadlock detection tool for C# that creates a continuation scheduling graph to detect potential deadlocks. By keeping track of which thread might execute which continuation and how continuations are sequenced and relate to each other, it can deduce whether there is a possibility for a deadlock. This technique has been proven to work on code bases up to 30000 lines of code. However, this technique does not detect deadlocks that occur with mutexes and channels.

## 2.3 Deadlock detection for Go

Similar to Kotlin, in Go one can spawn coroutines to do concurrent work. Both have primitives to non-blockingly, wait on coroutines and lock resources. And both allow for communication over channels. Static deadlock detection techniques for go are often based on behavioural types [17].

The first static deadlock detection based on behavioural types for go is, dingo-hunter [24]. In this work, local session types are inferred from coroutines and channels. These session types can be thought of as communicating state machines. By attempting to create a global graph from these small state machines, dingo-hunter can test for deadlocks.

A newer deadlock detection technique based on behavioural types is Godel Checker [22]. This technique is able to represent more of the go language, resulting in higher precision and recall. While inferring session types from go source code is a similar approach to the previous technique, the analysis differs. It uses the mCLR2 model checker [9] to check for deadlocks. This not only allows for more flexibility in properties that can be checked, but also takes advantage of the scalability a model checker can provide.

A recent advancement made, based on Godel Checker [22], is a technique called GOMELA [10, 11]. The first iteration of this technique solved a limitation of Godel Checker, namely the lack of support for coroutines spawned in loops and channels with limited capacity. This tool asks the user to bound parameters for loop count and channel capacity where needed. The Promela language is used to encode the behavioural types and the SPIN model checker to verify behaviour. Promela has native support for channels and is more alike Go, making it easier to report deadlocks. The second iteration of GOMELA added support for mutexes and waitgroups. This technique has been tested on large codebases, implying scalability.

# Chapter 3

# Background

In this section, we introduce Kotlin coroutines [14]. A coroutine is a special kind of function or code block that enables a program to multitask. By suspending and resuming at predefined points, they can yield computing resources to other coroutines and coordinate their execution. This kind of multitasking is called cooperative multitasking.

Coroutines are similar to threads in the sense that they execute code concurrently from the rest of the program. Where they differ is that threads are scheduled preemptively, meaning that they can be interrupted at any point. Whereas coroutines are scheduled cooperatively, meaning that the developer has control over when coroutines suspend and resume. In fact, a coroutine runs on a thread and can resume in a different thread after suspension.

Cooperative multitasking can be beneficial, for instance, when working with the network. Network requests can take a relatively long time; therefore, keeping computing resources occupied while waiting for the response would be wasteful. What coroutines allow is for other code to execute while the request is in progress. And this can significantly increase efficiency.

This section will introduce the following parts of Kotlin coroutines: the coroutine builders, suspend functions, the coroutine scope, coroutine dispatchers, and channels.

## 3.1 Coroutine builders

Coroutines are created and launched by a coroutine builder, which is a function that takes a suspendable code block and returns the result of that block either directly or wrapped. This code block runs asynchronously in the created coroutine. The different kinds of builders are discussed below.

### 3.1.1 runBlocking

The `runBlocking` coroutine builder creates a new coroutine that blocks the current thread until the coroutine completes. Since it launches the coroutine in a blocking fashion, the result will be returned in a normal matter. This builder is mainly used to bridge the synchronous and asynchronous world. In the synchronous world, the code doesn't run in a coroutine and thus cannot suspend. `runBlocking` creates a coroutine that allows for code

Listing 3.1: runBlocking coroutine builder

```
// synchronous world
var asyncResult = runBlocking {
    // asynchronous world
}
```

Listing 3.2: launch and async coroutine builders

```
// parent coroutine
var job: Job = launch {
    // child coroutine
}
job.join() // wait for child coroutine to finish

var deferred: Deferred = async {
    // another child coroutine
}
var asyncResult = deferred.await() // wait for result
```

to be suspended and, therefore, bridges to the asynchronous world. Listing 3.1 shows the syntax for runBlocking builder.

### 3.1.2 Launch

The launch builder is used to create a coroutine that does not return a value. The builder itself, however, returns directly a `Job` object that can be used to control the launched coroutine (see Listing 3.2). Note that the `launch` builder can only be called from a `CoroutineScope` (see 3.3) and therefore can only be used in an asynchronous context.

### 3.1.3 Async

The `async` builder is used to create a coroutine that does return a value. Listing 3.2 shows that the builder returns an instance of the `Deferred` object. This object can control the coroutine just like the `Job` object but also wraps the return value of the code block running in the coroutine. Similar to the `launch` builder, the `async` builder is called upon a `CoroutineScope`.

## 3.2 Suspend function

The suspend function is an important building block of Kotlin's concurrency model. As the name suggests, it is a function that can suspend (and resume) execution. This property allows coroutines to pause and resume and, therefore, share threads. A suspend function

Listing 3.3: suspension points and suspend fun

```
1 fun main() {
2     runBlocking {                    // Spawn coroutine 1
3         var job = launch { ... } // Spawn coroutine 2
4         foo()                        // Suspension point
5         job.join()                   // Suspension point
6     }
7 }
8
9 suspend fun foo() { ... }
```

can only be invoked in an asynchronous context so, either from a coroutine code block or from another suspend function (which runs by definition also in a coroutine).

Listing 3.3 shows `runBlocking` spawning the first coroutine and `launch` spawning the second. The call to suspend function `Foo()` does create a suspension point, allowing coroutine 1 to potentially yield resources when this call is reached. The `job.join()` call creates another suspension point in coroutine 1. This call suspends coroutine 1 until coroutine 2 has finished execution.

## 3.3 Coroutine scope

Coroutines follow a principle of structured concurrency, which means that new coroutines can only be launched in a specific scope, delimiting the lifetime of the coroutine. Structured concurrency ensures that they are not lost and do not leak. An outer scope cannot be completed until all its children's coroutines are complete.

In Listing 3.3, `runBlocking` launches a new coroutine and establishes a coroutine scope (accessible by `this` in the code block), so any coroutine launched within this block will cause this `runBlocking` call to wait until the launched coroutine finishes, even without explicit join calls.

One may also specify a custom `CoroutineScope` to ensure that launched coroutines do not get lost and do not leak. Specifically, the scope finishes when all the coroutines launched within it are completed. Listing 3.4 contains the `printHelloWorld()` suspending function that, launches a new coroutine and prints `"Hello"`, while the launched coroutine suspends for one second and prints `"World!"`. The coroutine scope here ensures that `printHelloWorld()` finishes only when the launched coroutine finishes.

## 3.4 Dispatchers

The dispatcher determines the thread on which a coroutine runs. Like the OS dispatcher allows threads to run on physical cores, a coroutine dispatcher allows coroutines to run on a thread. The dispatcher of a coroutine can be defined during launch or switched with the `withContext` method. See Listing 3.5. The following dispatchers are available.

Listing 3.4: `CoroutineScope` in Kotlin.

```
suspend fun printHelloWorld() = coroutineScope {
    launch {
        delay(1000L)
        println("World!")
    }
    println("Hello")
}
```

Listing 3.5: coroutine context

```
runBlocking (Dispatchers.Default) {
    // This code runs on the default thread pool
    launch (Dispatchers.Main) {
        // This code runs on UI thread
    }
}
```

### 3.4.1 BlockingEventLoop

When no dispatcher is provided to the current coroutine context, a single-threaded event loop is used that blocks the current thread. The `runBlocking` builder uses a `BlockingEventLoop` and all coroutines that inherit from this context.

### 3.4.2 Dispatchers.Default

This dispatcher is designed for CPU-bound work. It uses a shared background thread pool, and the number of threads is typically equal to the number of CPU cores. In contrast to what the name might suggest, this dispatcher is not used by default.

### 3.4.3 Dispatchers.IO

This dispatcher is optimized for I/O-bound work, such as reading and writing files, making network requests, or accessing a database. It's backed by a thread pool, but the number of threads is larger than in `Dispatchers.Default` because I/O operations are often slower and may involve waiting.

### 3.4.4 Dispatcher.Main

This dispatcher is designed for UI-related work in Android applications. It ensures that the coroutine runs on the main (UI) thread. This single UI thread is necessary since the Android UI toolkit is not thread-safe [16]. Working with a single thread to do UI operations prevents concurrency issues.

### 3.4.5 Dispatchers.Unconfined

This dispatcher is not confined to any specific thread. It deploys the coroutine on the thread from where it is started or resumed. In other words, if a suspend call is made from this coroutine and that runs on a different thread, then the resumed coroutine will also run on that thread.

### 3.4.6 singleThreadContext

All coroutines using this context will run on the same thread. Since this reserves a native resource (thread or worker), it is considered expensive and is, therefore, part of the delicate API.

## 3.5 Channels

A channel can pass information from one coroutine to another and/or can be used to synchronize two coroutines. Where a Deferred transfers one value between coroutines a channel can stream values. It has a `send` and a `receive` method and a capacity (buffer size) that determines the number of pending messages it can hold at a time. [19]

### 3.5.1 Rendezvous channel

The rendezvous channel has a buffer size of zero. When a message is sent over the channel, the sending coroutine is suspended until a `receive` call has been made. When a `receive` call is made before `send` the receiving coroutine is suspended until something is sent over the channel. This way, each `send` is matched to a `receive`, where the channel resumes both coroutines when the *rendezvous* happens.

### 3.5.2 Buffered channel

A buffered channel has a buffer size larger than zero. Now, the `send` method only suspends when the buffer is full. The `receive` method suspends when the buffer is empty. The buffer can be seen as a queue where the `send` method adds a message to it, and the `receive` method takes one.

### 3.5.3 Multiple senders or receivers

When multiple coroutines `send` a message over the channel but the `receive` method has not been called, the sending coroutines are put in a queue. The same holds for receiving coroutines when multiple `receive` calls have been made before a message is sent. This queue should not be confused with the message buffer since all messages in the buffer did not suspend the `send` call, but all messages in the send queue did suspend.

# Chapter 4

# RunBlocking analysis

This chapter discusses a common problem, found in Kotlin coroutine programs, that leads to deadlocks and performance issues: nested *runBlocking*. First, some theory and intuition about the problem are given in section 4.1. Then, section 4.2 discusses the proposed solution to the problem. After which the implementation of the discussed solution is described in section 4.3. Next, we will discuss the performance and limitations of the provided implementation in section 4.4. Lastly, the discussion and future work recommendations are in section 4.5.

## 4.1 The runBlocking problem

The *runBlocking* builder bridges the synchronous and asynchronous world. It should function as a starting point from where coroutines can be spawned. Once inside a coroutine, we should use the `launch` or `async` builder. However, calling the runBlocking builder from a coroutine can lead to problems.

In the following sections I will explain why calling the runBlocking builder from a coroutine is: risky (section 4.1.1), easily overlooked (section 4.1.2), solvable (section 4.1.3), and most importantly avoidable (section 4.2).

### 4.1.1 Why avoid runBlocking from coroutine?

As the name suggests, the *runBlocking* builder blocks the function calling it. This blocking happens on thread level. So when the *runBlocking* builder is called from a coroutine the underlying thread gets blocked. Where coroutines are designed to share threads for performance gains, in this case we block the thread which was supposed to be a shared resource.

While a potential loss of performance is undesired, a far more severe problem might occur when *runBlocking* builders keep getting called from the wrong places. Listing 4.1 provides the code and the deadlock illustration. The program launches Coroutine *A* on the `Dispatcher.Main` dispatcher (line 2), dispatching the coroutine onto the UI thread. Then, coroutine *A* calls `nonSuspendingFunction()`. In turn, this function launches a new coroutine *B* via *runBlocking*, scheduling it on `Dispatcher.Main` (line 8). The *runBlocking* builder blocks the UI thread until coroutine *B* finishes. However, coroutine *B* cannot be

dispatched until the UI thread is free. In other words, coroutine *A* blocks the thread that needs to execute coroutine *B*, while coroutine *A* also waits for coroutine *B* to complete, which results in a deadlock.

Listing 4.1: A program with a deadlock due to a *runBlocking* call from a coroutine on a single-threaded dispatcher.

```
fun main() = runBlocking {
  launch(Dispatchers.Main) { // launch coroutine A
    nonSuspendingFunction() // call that leads to runBlocking
  }
}

fun nonSuspendingFunction() {
  runBlocking(Dispatchers.Main) { // launch coroutine B
    println("Done")
  }
}
```

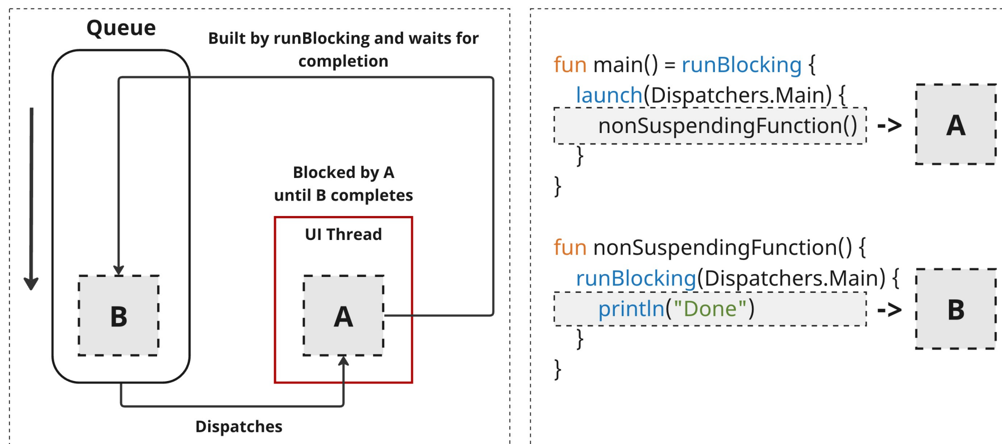


Figure 4.1: Visualization of a deadlock when calling *runBlocking* from a coroutine dispatched in a single-threaded environment.

It might seem like this is a limitation of a single-threaded dispatcher. However, it can happen in multithreaded scenarios as well. The same deadlock can occur not only in a single-thread case but also in multi-threaded scenarios. Listing 4.2 provides such an example. This program uses the `Default` multi-threaded dispatcher, which enables parallel computations and typically bounds the number of threads to the number of CPU cores. A program that spawns coroutines in a loop (e.g., spawning coroutines *A* on line 3) suffers from a similar problem as in Listing 4.1. The program reaches deadlock when all the threads of the `Default` dispatcher execute the copies of coroutine *A*, which are all waiting for coroutine *B* to be scheduled. It is only a matter of time before all threads of the `Default`

Listing 4.2: A program with a deadlock due to a *runBlocking* call from a coroutine on a multithreaded dispatcher.

```
fun main() = runBlocking {
  for(i in 1..1000) { // 1000 > max number of scheduler threads
    launch(Dispatchers.Default) {
      nonSuspendingFunction()
    }
  }
}

fun nonSuspendingFunction() {
  runBlocking(Dispatchers.Default) {
    println("Done")
  }
}
```

dispatcher are executing a coroutine $A$, which are all waiting for their coroutine $B$ to be scheduled.

### 4.1.2 Why is this mistake easily made?

A common misconception is that the *runBlocking* builder can safely be used in non-suspending functions. However, a non-suspending function can be called from an asynchronous context; there is no guarantee that it runs outside a coroutine. Even if developers know they are working inside a coroutine, they might be unaware that the function they call contains a *runBlocking* builder. It is not always trivial to determine whether a piece of code runs inside a coroutine or whether a function call reaches a *runBlocking*, especially when the call stack is large.

When developers need to call a suspending function from a non-suspending function, they tend to call *runBlocking*, especially when the developer is unaware that this synchronous function actually runs in a coroutine. Another situation is where a developer is sure that the current function is not reachable from a coroutine and in that case the only valid option would be to call the *runBlocking* builder. However, in the future another developer might need to call this function without having knowledge of the internal workings (which in larger teams is very probable) and calls it from a coroutine. The promise that any normal function can be called from a suspend function wihtout worry doesn't seem to hold.

A situation where it is nontrivial to identify parts of the codebase that run in coroutines and they may introduce unintended nested *runBlocking* calls might occur when the codebase is gradually migrated to use coroutines [8].

### 4.1.3 How to solve this mistake?

In the example program, turning `nonSuspendingFunction` into a `suspendingFunction` by adding the `suspend` keyword does the trick, as given as a potential solution in Listing 4.3.

Listing 4.3: A potential solution to the bug with nested *runBlocking* calls.

```
fun main() = runBlocking {
  launch(Dispatchers.Main) { // launch coroutine A
    suspendingFunction() // safe to call
  }
}

suspend fun suspendingFunction() {
  coroutineScope { // suspends execution until coroutine B is done
    launch(Dispatchers.Main) { // launch coroutine B
      println("Done")
    }
  }
}
```

By turning the function into a suspend function, the developer can call `coroutineScope`, which allows for a normal `launch`. Note that this requires all functions calling `suspendingFunction` to be suspending as well.

Ultimately, both *runBlocking* and `coroutineScope` will pause the execution of the function calling it. The difference is that *runBlocking* does this by blocking the underlying thread and `coroutineScope` by suspending the coroutine, which releases the thread in the meantime.

In other cases, one might prefer to acquire a coroutine scope created from somewhere else. This scope, however, comes with its own set of challenges [6]. Therefore, the right course of action is not always clear and requires careful consideration by the developer.

## 4.2 Solution

From section 4.1 it is clear that the *runBlocking* builder should not be called from a coroutine. The core of the problem seems to be the challenge of keeping track of code that potentially runs in coroutines. Therefore, we propose a solution that not only aids developers in fixing current occasions of nested *runBlocking* but might prevent the issue altogether: a static analysis that finds and reports on *runBlocking* calls made from coroutines. When implemented correctly and incorporated in an IDE this can potentially prevent dangerous code changes that result in such *runBlocking* cases by warning the developer before a code change is made final.

Sections 4.2.1 and 4.2.2 explain the theory and approach of the analysis, where sections 4.2.3, 4.2.4 and 4.2.5 reveal the difficulties and how they are addressed.

### 4.2.1 Coroutine Call Graph

To determine whether a *runBlocking* possibly runs in a coroutine or not, we need to know what part of the codebase runs in coroutines. A function runs in a coroutine when that

function is reachable from the body of a coroutine builder. Therefore, we need to identify all sites where coroutines are created. Those sites can than be the starting points of a call graph that represents the part of the codebase that runs in coroutines. As discussed in section 3.1 a coroutine is created with one of the following builders: `launch`, `async`, and *runBlocking*. In Listing 4.4 there are three coroutine builders: *runBlocking* builder *R1* on line 1, `launch` builder *L1* on line 5, and another *runBlocking* builder on line 19. These builders are represented in Figure 4.2 as red nodes in the graph. Then, by exploring function calls (like on line 2 or line 6) and adding those to the graph we get a complete picture of what functions run in coroutines.

Listing 4.4: Example Kotlin program with function calls and coroutine builders.

```
1  fun main() = runBlocking {// launch R1
2      foo()                 // call 1
3      bar()                 // call 2
4
5      launch {              // launch L1
6          bar()             // call 3
7      }
8  }
9
10 fun foo() {
11     badFun()              // call 4
12 }
13
14 fun bar() {
15     badFun()              // call 5
16 }
17
18 fun badFun() {
19     runBlocking {         // launch R2
20         println("Hello")
21     }
22 }
```

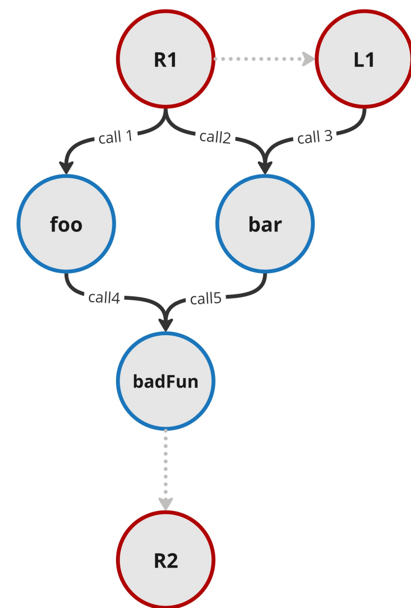


Figure 4.2: Call graph representing all coroutine builders and functions they can reach from Listing 4.4

### 4.2.2 Trace from coroutine creation to runBlocking

While it is clear that a *runBlocking* builder called from the call graph runs in a coroutine, we would like to provide the developer additional info about the origin of the problem. A potential call stack from coroutine origin to the *runBlocking* builder in question can help the developer understand the problem at hand in more detail. By traversing the graph backwards from a function node we can deduce what sequence of calls resulted in this function running in a coroutine. From the graph in Figure 4.2 we can easily see that there are three paths from a coroutine builder to the *runBlocking* on line 19 (Listing 4.4). Namely:

- $R1 \rightarrow foo \rightarrow badFun \rightarrow R2$
- $R1 \rightarrow bar \rightarrow badFun \rightarrow R2$
- $L1 \rightarrow foo \rightarrow badFun \rightarrow R2$

Note that enumerating all possible paths in larger call graphs can become infeasible. Therefore, the choice is made to only compute (one of) the shortest path(s). Also important to note that this graph can contain cycles (recursion). These should be skipped when searching for coroutine origin.

### 4.2.3 Lambdas

A challenging situation arises when we encounter lambda functions. First of all, for these functions it can be unclear when and where they are executed. Secondly, when calling a lambda function it can be unclear which one is actually invoked. For example, in Listing 4.5 there is a class `LambdaWrapper` defined (line 13) which has a variable lambda property (line 14). This lambda function can be set by calling `setFun` (line 15), and can be invoked later by calling `invoke`. It might look like there is a case of nested *runBlocking* calls (*R1* on line 3 and *R2* on line 5), however *R2* is actually invoked on line 10. This is represented in the graph in Figure 4.3. From this graph it is clear that the `R2` runBlocking does not run from a coroutine, given that `foo` doesn't un in a coroutine. In this example it might seem trivial to determine where `R2` is executed but with larger code bases where multiple coroutines can call `setFun` it becomes increasingly more difficult to determine.

However, ignoring lambdas all together would make our analysis weak especially since lambdas are a first class citizen in Kotlin and can be found in abundance. For example, looping commonly happens with `forEach {}` or `repeat(n) {}`. Luckily, in Kotlin most functions that take lambda arguments are defined as `inline`. An `inline` function is a function that in reality is not called but where all callsites are replaced with the body of the function. Listing 4.6 is an example of an inline function `runFun` that takes a lambda argument and is called on line 2. Listing 4.7 shows what the code would look like after the *inlining* step of the compiler. Semantically, the code achieves the same, but in Listing 4.7 the function call and the lambda function have been merged into the body of `main`. More importantly, from inline functions we know that their lambda arguments are invoked at the callsite. And therefore, the body of the lambda function can be treated as the body of the function which calls the inline function. In the case of our example (Listing 4.6 and 4.7) this means that the `print("world ")` call can be added directly as a child to the `main` function node.

Lastly, suspend lambdas run per definition in a coroutine and can therefore be safely explored as well, for the coroutine call graph creation. So in conclusion, the analysis skips the exploration of lambda arguments unless they are an argument for an `inline` function or the argument itself is a `suspend` lambda.

Listing 4.5: Example Kotlin program with where a lambda function is passed but invoked from different execution context.

```
fun foo() {
    val lw = LambdaWrapper()
    runBlocking { //R1 blocking coroutine
        lw.setFun {
            runBlocking { //R2 looks nested
                println("Blocking")
            }
        }
    }
    lw.invoke() //actually invokes runBlocking
}

class LambdaWrapper {
    private var changableFun: () -> Unit = {}
    fun setFun(lam: () -> Unit) {
        changableFun = lam
    }
    fun invoke() {
        changableFun()
    }
}
```

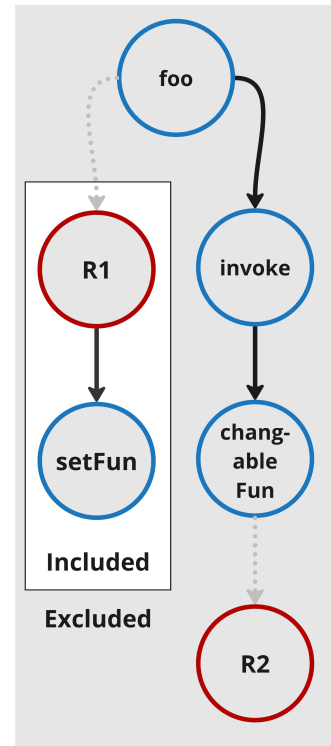


Figure 4.3: Call graph representing the code in Listing 4.5. The *Included* area is the resulting Coroutine Call Graph.

### 4.2.4 Functions overrides

Since Kotlin is object oriented and thus supports inheritance, it might not always be clear which implementation of a function is actually invoked during runtime. To address this issue, we explore only the function of the declared receiver type by default. However, we can give the user more control by providing different strategy options. We provide two other options, one is to not explore functions with overrides at all, this might have a big impact on the number of true positives. The other is to explore all overrides.

### 4.2.5 Other limitations

There are other control flow constructs that are not modelled using the proposed call graph. For example, conditional branching and shortcicuiting. By assuming that all branches can be taken, might lead to false positives. However, one could argue that it would be bad design when a nested *runBlocking* case is dependent on conditional branching.

Another limitation of this approach is that it does not take library or java code into

Listing 4.6: Example Kotlin program with where a lambda function is passed but invoked from different execution context.

```
fun main() {
   runFun {
       print("World ")
   }
}

inline fun runFun(fn: ()->Unit){
    print("Hello ")
    fn()
    print("!")
}
```

Listing 4.7: Example Kotlin program with where a lambda function is passed but invoked from different execution context.

```
fun main() {
    print("Hello ")
    print("World ")
    print("!")
}
```

account.

## 4.3 Implementation of runBlocking analysis

To implement the discussed analysis approach, we would need some tools that help parse and traverse Kotlin code. Also, to make the analysis effective for developers, it should be made available as part of a common developer tool used for Kotlin development.

In the case of Kotlin, the IntelliJ platform is the obvious (and only) choice. The IntelliJ platform SDK offers a powerful toolset for creating code analyses. For example the PSI (Program Structure Interface) system to traverse Kotlin's AST and resolving references to method calls. But also APIs to present the analysis results in the appropriate format and location. Additionally, most Kotlin developers use IntelliJ IDEA (or a variant like Android Studio) to develop, which supports the choice for the IntelliJ platform even further.

Figure 4.4 shows what the end user would see after analysing a code base containing Listing 4.4. On the left side a list of files is presented that contain runBlocking(s) called from a coroutine. And per file the *runBlocking* problems. When a *runBlocking* problem is selected, it presents a potential call stack from coroutine origin to the *runBlocking* in question. Each call in this stack is clickable and opens the corresponding file in the editor while moving the cursor to the right code element.

To choose the exploration strategy discussed in section 4.2.4 the user can access the settings of this particular inspection. This is shown in Figure 4.5.

### 4.3.1 Released into production

The implementation has been reviewed and approved by JetBrains and can be found at [5]. The runBlocking inspection is available as a standard inspection from IntelliJ IDEA 2024.2 onwards. To find the inspection, one would have to go to the *Problems* tool window, open the *Project Errors* tab and click on *Inspect Code*. A window will reveal itself which allows

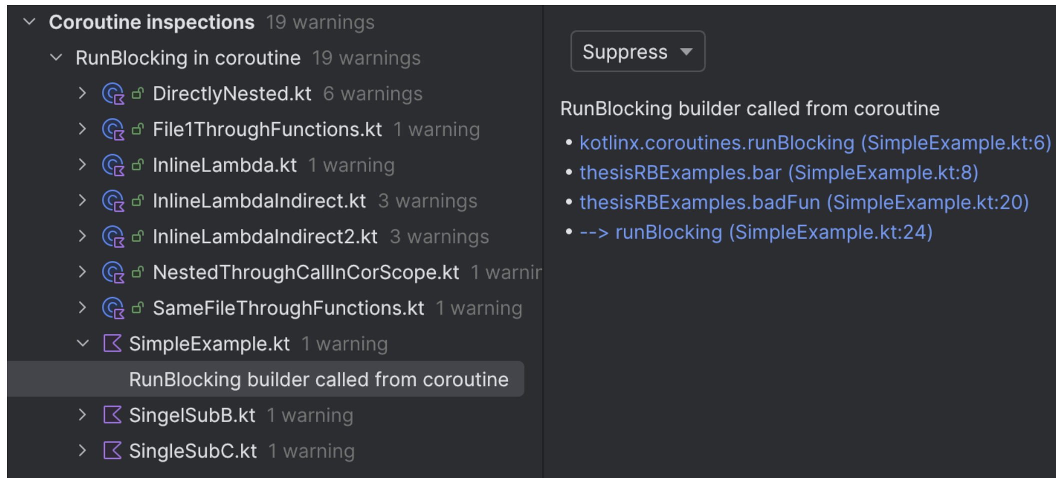


Figure 4.4: What the end user would see for an analysis of Listing 4.4.

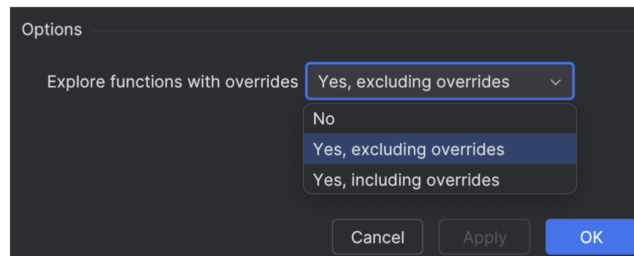


Figure 4.5: How an end user would change the settings of the analysis.

the user to select the scope of the analysis and what inspections to include. This window also allows the user to select one of the exploration strategies.

## 4.4 Validation

This section presents the validation results of the *runBlocking* analysis implementation. To give some notion about accuracy and scalability, the algorithm has been tested on several open source projects. Eight Android repositories have been selected since Kotlin is the primary language for android app development. Additionally, the Ktor backend framework and the IntelliJ-Community repository have been selected as well.

For each of these repositories and every exploration strategy, the analysis results have been evaluated to count the true and false positives. During development, it became clear that a second run of the algorithm usually runs much faster. Hence, for each strategy and repository the execution time is measured twice. Once with a fresh boot of the IDE and a second directly after.

Table 4.1: True and false positive counts per repository per exploration level.

| Repository | Strict TP | Strict FP | Default TP | Default FP | All TP | All FP |
|---|---|---|---|---|---|---|
| Woocommerce | 0 | 0 | 0 | 0 | 0 | 0 |
| DuckDuckGo | 1 | 0 | 1 | 0 | 5 | 2 |
| Tachiyomi | 3 | 0 | 3 | 0 | 3 | 1 |
| Wordpress | 0 | 0 | 0 | 0 | 0 | 0 |
| Ktor | 0 | 0 | 0 | 0 | 1 | 1 |
| Home-Assistant | 3 | 0 | 3 | 0 | 4 | 4 |
| Google-hfir | 2 | 0 | 2 | 0 | 2 | 0 |
| Firefox | 3 | 0 | 3 | 0 | 3 | 0 |
| IntelliJ | 8 | 0 | 8 | 0 | 9 | 1 |
| Total | 20 | 0 | 20 | 0 | 27 | 9 |

### 4.4.1 runBlocking analysis precision

The true and false positive results are shown in table 4.1. The three exploration strategies are *Strict* for not exploring overridable functions, *Default* for only exploring the overridable function itself, and *All* for exploring the overridable function and all its sub implementations. These strategies are discussed in more detail in section 4.2.4. What immediately stands out is that the *Strict* and *Default* modes produce exactly the same results and that both do not have false positives. For the *All* mode, however, there is a gain of seven true positives while also having nine additional false positives. Although the results of *Strict* and *Default* are satisfactory, it is important to note that false positives are still technically possible. Similarly, there might exist repositories where the *Default* setting finds more than the *Strict* setting. Nevertheless, these results support the choice to make the *Default* setting the default.

### 4.4.2 Execution time

The execution time is displayed in table 4.2. For each repository and each strategy, the execution times are displayed. The first execution is with a fresh IDE, and the second execution is directly after. It became clear that a second run is often faster. This is probably caused by some caching mechanism to load PSI trees quicker.

### 4.4.3 Resolution of real world cases

A few *runBlocking* cases of the DuckDuckGo and Home-Assistant repositories have been communicated to their corresponding development teams. These cases have been acknowledged and some of these have been solved [3, 2].

Table 4.2: Averaged execution times for a first (fresh) execution and a second (cached) execution.

| Repository | Strict 1 | Strict 2 | Default 1 | Default 2 | All 1 | All 2 | kLOC |
|---|---|---|---|---|---|---|---|
| Home-Assistant | 0:21 | 0:03 | 0:26 | 0:02 | 0:33 | 0:03 | 64.2 |
| Tachiyomi | 0:12 | 0:08 | 0:20 | 0:15 | 0:26 | 0:10 | 78.0 |
| Google-hfir | 0:32 | 0:13 | 0:12 | 0:05 | 0:17 | 0:02 | 100.3 |
| Ktor | 0:42 | 0:44 | 0:42 | 0:59 | 1:02 | 0:15 | 180.5 |
| Wordpress | 0:18 | 0:21 | 0:20 | 0:24 | 0:27 | 0:24 | 306.2 |
| Woocommerce | 0:36 | 0:03 | 0:32 | 0:05 | 0:26 | 0:05 | 327.7 |
| DuckDuckGo | 0:38 | 0:04 | 0:36 | 0:04 | 0:41 | 0:05 | 351.5 |
| Firefox | 1:09 | 0:21 | 1:12 | 0:09 | 1:37 | 0:20 | 1178.6 |
| IntelliJ | 5:50 | 4:55 | 4:18 | 3:48 | 5:55 | 5:04 | 4222.8 |

## 4.5 Discussion and future work

Although this inspection has proven to be effective and scalable, it leaves room for improvement in two areas.

First, while a runtime of 1 or a few minutes for a large repository is acceptable, a tool is only useful when it is actually used. A minute of waiting after each commit for probably no warning seems like a lot. To leave it to the developer to manually trigger this analysis every once in a while might result in a nested *runBlocking* case that is deeper rooted in the code and therefore harder to solve. Assuming that the developer remembers to use this inspection. One simple solution would be to add this inspection as a step to a CICD pipeline which can run in parallel with other steps. A better but more involved solution would be to make this analysis incremental where we only need to recalculate parts of the callgraph that have changed. This way the developer can be notified while working on the project just like most other (fast) inspections in the IDE.

Second, while it is true that inline lambdas are more common than regular lambdas. Higher order functions are first-class citizens in Kotlin and regular old lambdas are therefore not rare. This potentially leads to missing important *runBlocking* cases since the current state of the analysis skips regular lambdas. In order to find which call sites call what lambdas, we need to find where a reference originates from and how it is passed from function to function. However, keeping track of references during static analysis can be very challenging. This is discussed further in chapter 5.

# Chapter 5

# Static Deadlock Detection

In the previous chapter, we discussed the detection of a possible anti-pattern that, in some cases, can lead to a deadlock. This chapter will focus on detecting deadlocks that are the result of the interaction between coroutines. These interactions can happen through channels, mutexes, or joins (or a combination thereof). Since there exists a plethora of techniques that detect mutex deadlocks, the choice has been made to focus on channel-related deadlocks. Moreover, channels are flexible enough to function as a mutex or a join. Therefore, once we can detect channel-based deadlocks, it should be relatively easy to support mutexes and joins as well.

As discussed in chapter 3 Go has similar features to Kotlin in terms of concurrency. There already exist several successful works that describe static deadlock analysis techniques for channels in Go. These techniques model concurrent behaviour and leverage the power of existing model checkers to validate that model. From these model languages, Promela (Process Meta Language) [21] seems well suited for our use case for the following reasons: Promela has been designed to model distributed and concurrent behaviour, Promela has native support for channels, Promela is more readable than other model languages like mCLR2. Additionally, most recent works use Promela as well. Therefore, using this language to model Kotlin's concurrent behaviour seemed like the best strategy.

The goal of this thesis is not to develop a comprehensive tool that fully models every language feature and deadlock. What this work will focus on is developing a prototype to evaluate the mentioned analysis's ability to detect deadlocks and assess its scalability. Therefore, this prototype will only support channel deadlocks and a subset of Kotlin's features. This will be discussed in greater detail in section 5.2.

But first, section 5.1 will give an introduction to such channel deadlocks in Kotlin. Then, section 5.2 discusses how Kotlin's concurrent behaviour can be expressed as a Promela model. After which, section 5.3 will explain how the translation works. Section 5.4 will briefly discuss how this model is verified by SPIN. Section 5.5 will go into detail about the accuracy of the analysis technique. Section 5.6 discusses the scalability of the proposed technique. Lastly, the findings will be discussed in detail, and some recommendations for future work are given in section 5.7.

## 5.1 Channel Deadlocks

The `channel` primitive in Kotlin coroutines does not only function as a message passing mechanism but also allows for synchronization. As discussed in section 3.5, a channel can suspend a coroutine when the `send` or `receive` functions are called. The coroutine is resumed again when a corresponding `receive` or `send` call is made. However, incorrect use of channels might lead to deadlocks. In the example of Listing 5.1 there are two coroutines spawned: coroutine *A* (line 5) and coroutine *B* (line 10). Also, two channels are initialized: *ch1* and *ch2* on lines 2 and 3, respectively. Coroutine *A* sends a message over both channels, and coroutine *B* receives over both channels. However, they do these operations in reversed order. Coroutine *A* is suspended after it sends over channel 1 and waits for a receive on channel 1. While coroutine *B* is suspended after the receive on channel 2 and waits for a send on channel 2. Neither of these coroutines can continue. As Figure 5.1 shows: all four operations wait on each other to finish, resulting in a deadlock.

Listing 5.1: Example Kotlin program that leads to a channel deadlock.

```
1  fun main() = runBlocking {
2    val ch1 = Channel<String>()
3    val ch2 = Channel<String>()
4
5    launch { // coroutine A
6      ch1.send("Hello")
7      ch2.send("World")
8    }
9
10   launch { // coroutine B
11     val world = ch2.receive()
12     val hello = ch1.receive()
13     println(hello + world)
14   }
15 }
```

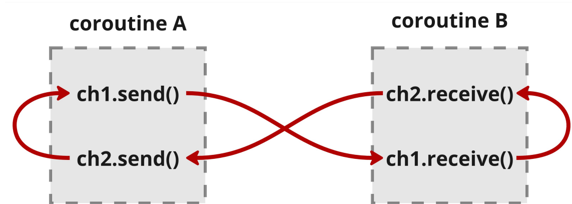


Figure 5.1: A visual illustration of the wait for relations that result in the deadlock of Listing 5.1. Each arrow shows what is waited for e.g. `ch1.send()` waits for `ch1.receive()`.

The channel deadlock shown in the last example consistently goes into a deadlock. However, there exist programs that non-deterministically end up in a deadlock as well. In other words, a program where in one execution it behaves as expected and another execution it ends up in a deadlock. This is possible due to the non-deterministic nature of scheduling coroutines. Listing 5.2 gives an example of such a bug. This bug works with one channel created on line 3. Note that this channel has a capacity of one and is therefore a buffered channel. As discussed in section 3.5: a buffered channel only suspends a `send` call when its buffer is full and suspends a `receive` when the buffer is empty. Coroutine *A* and *B* are spawned on lines 5 and 10, respectively. Figure 5.2 shows an execution order of the two coroutines that does not lead to a deadlock. As expected, coroutine *A* sends a message which is received by coroutine *B*. And afterward, coroutine *A* receives a message sent by coroutine *B*. Figure 5.3, on the other hand, shows an execution order that does lead to a

deadlock. Coroutine *A* sends and receives its own message, which is possible since the channel has a buffer. Then coroutine *B* calls the `receive` method but, since the buffer is empty, suspends. The only other `send` operation is after the current suspending `receive` and is therefore never reached. This results in a deadlock.

Listing 5.2: Example Kotlin program that, depending on execution order, can leads to a channel deadlock.

```
fun main() = runBlocking {
  // Channel of capacity 1
  val ch = Channel<Int>(1)

  launch { // coroutine A
    ch.send(0)
    ch.receive()
  }

  launch { // coroutine B
    ch.receive()
    ch.send(1)
  }
}
```

| coroutine A | coroutine B |
|---|---|
| ch.send(0) | |
| | ch.receive() |
| ch.receive | |
| | ch.send(1) |

Figure 5.2: Execution order of Listing 5.2 that does not lead to a deadlock.

| coroutine A | coroutine B |
|---|---|
| ch.send(0) | |
| ch.receive | |
| | ch.receive() |
| | ~~ch.send(1)~~ |

Figure 5.3: Execution order of Listing 5.2 leads to a deadlock. The last send in coroutine B will never be executed.

## 5.2 Expressing Kotlin in Promela

This section will elaborate on what language features will be supported and how these features can be modelled in Promela.

### 5.2.1 Kotlin requirements

As the goal is to design a prototype, and Kotlin is a far more advanced language than Promela, there will be some features that will not be supported during the translation from Kotlin to Promela. However, as this will be tested on working code samples and real world repositories, we need to find the right subset of Kotlin to support.

To detect deadlocks involving coroutines and channels, it is necessary to model coroutines along with the initialization, sending, and receiving actions of channels. As demonstrated in the examples provided in Section 5.1, the order of operations on a channel is crucial. Therefore, it is essential to account for Kotlin's control flow, including statement order, functions, and calls, as well as coroutines and asynchronous calls.

To determine which channel a send or receive operation targets, we must track references and data flow, including function parameters, call arguments, and local variables. Given that real-world applications often store channels as class properties, it is important to support these as well. Consequently, we must also support classes to effectively model class

properties. Therefore, the model should include compositional types (structs), constructors, function return values, and call receivers.

Some language features that will not be supported in the prototype are:

- Conditional branching. For example, `if (A) B; else C;` will be treated as `A; B; C;`.
- Lambdas. They are treated as inlined, meaning that they are modeled as if they run at the location where they are defined.
- Mutability and nullability.
- Inheritance.

### 5.2.2 Writing Promela

Using a series of examples, we will explain how Promela models Kotlin. During this exploration, we will encounter certain limitations of Promela and demonstrate how these challenges can be addressed.

#### Coroutines and Channels

Listing 5.3 is the first deadlock example from section 5.1. Next to it, listing 5.4 shows what this code sample would look like in Promela after translation.

As discussed before, the Kotlin example creates two channels (lines 2 and 3), and launches two coroutines (lines 5 and 10) that perform send and receive operations on those channels (lines 6, 7, 11, and 12).

The Promela example shows the creation of channels on lines 2 and 3. It shows a send operation `ch1!0` on line 10 and line 11. Note that the send operator is an exclamation mark and that an integer 0 is sent. Since this analysis does not care about the content sent over channels we default to send the integer 0. Line 15 and 16 are receive operations (?).

To model a coroutine Promela has a similar construct called a process. Processes like coroutines run concurrently with each other and can be spawned dynamically. To define a coroutine or process the keyword `proctype` is used. The syntax is similar to a function definition: it has a name and arguments. The two coroutines A and B from Kotlin example, lines 5 and 10, are defined in the Promela example on lines 9 and 14. Note that in the Kotlin example, the channel references are available from the lexical scope, and in Promela, these channels are passed explicitly.

In Kotlin, the `launch` builder is used to start a coroutine asynchronously (lines 5 and 10). In Promela, the `run` keyword is used to start a process (lines 5 and 6).

Listing 5.3: A Kotlin example program that leads to a channel deadlock.

```
fun main() = runBlocking {
  val ch1 = Channel<String>()
  val ch2 = Channel<String>()

  launch { // coroutine A
    ch1.send("Hello")
    ch2.send("World")
  }

  launch { // coroutine B
    val world = ch2.receive()
    val hello = ch1.receive()
    println(hello + world)
  }
}
```

Listing 5.4: A Promela example program that leads to a channel deadlock.

```
proctype main() {
  chan ch1 = [0] of {int}
  chan ch2 = [0] of {int}

  run corA(ch1, ch2)
  run corB(ch1, ch2)
}

proctype corA(chan ch1;chan ch2){
  ch1 ! 0
  ch2 ! 0
}

proctype corB(chan ch1;chan ch2){
  ch2 ? 0
  ch1 ? 0
}
```

#### Functions and Calls

Promela does not support functions. Therefore, functions need to be modelled in terms of processes and channels. Since processes accept arguments and can be called through the run keyword they are closely related to a function. There is only one caveat, a process runs asynchronously from the calling code. Therefore, we need to be able to differentiate between synchronous calls (normal functions) and asynchronous calls (spawning coroutines). This can be achieved by passing to all processes a return channel `ret`. This channel can then be used to synchronize the end of the callee and the continuation of the caller.

Listing 5.5 is an example of a synchronous call, while listing 5.6 is an example of an asynchronous call. Both examples have a main process that calls another process (line 3). In both cases, there is a return channel created (line 2) that is passed to the called process. Subsequently, the called process returns by sending over the return channel (line 9). Where the examples differ is how the caller function `main` deals with the return channel `callReturn` (line 4). The synchronous call receives over the return channel (line 4). This makes the `main` process wait on the `synCall` process to finish. The asynchronous case spawns another special `receiver` process (line 4) that receives over this channel asynchronously. Basically, it makes sure that the return statement on line 9 has a corresponding receive without blocking the caller.

Listing 5.5: Synchronous function call in Promela.

```
proctype main() {
  chan callReturn = [0] of {int}
  run syncCall(callReturn)
  callReturn ? 0
}

proctype syncCall(chan ret) {
  // do stuff
  ret ! 0
}
```

Listing 5.6: Asynchronous function call in Promela.

```
proctype main() {
  chan callReturn = [0] of {int}
  run asyncCall(callReturn)
  run receiver(callReturn)
}

proctype asyncCall(chan ret) {
  // do stuff
  ret ! 0
}

proctype receiver(chan ch) {
  ch ? 0
}
```

**Return values**

A (Promela) channel not only synchronizes two processes but also allows for sending a value. Therefore, we can use the `ret` channel to return a value to the caller. Listings 5.7 (Kotlin) and 5.8 (Promela) show an example where a function `main` calls a function `buildChan` that creates and returns a channel. The return channel `callReturn` created on line 2 (of listing 5.7), is now typed as a `chan` channel. The type of value that is transferred over a channel is defined within the curly braces. At the callee function `buildChan` the created channel is returned over the `ret` channel (line 12). This returned channel is then received in the caller function `main` on line 6. And store in the `channel` variable on line 5.

Note that the channel created in the `buildChan` function is constructed by calling a special `construct_ch` function (line 11). This is due to a limitation of the Promela language. In Promela a channel lives as long as the process that created it. If the `buildChan` process would create this channel and return it to the `main` process, then the channel would be unusable after `buildChan` finishes.

Listing 5.7: A Kotlin example of a function returning a channel.

```
suspend fun main() {
  val channel = buildChan()
}

fun buildChan(): Channel<Int> {
  val ch = Channel<Int>()
  return ch
}
```

Listing 5.8: A Promela example of a function returning a channel.

```
proctype main() {
  chan callReturn = [0] of {chan}

  run buildChan(callReturn)
  chan channel
  callReturn ? channel
}

proctype buildChan(chan ret) {
  chan ch
  construct_ch(ch)
  ret ! ch
}
```

To tackle this problem, we propose a novel solution: define global channels. This is done by initializing the channel at the top of a Promela file instead of in a process. For each channel initialization in the Kotlin program, we create a global channel in the Promela model. Subsequently, in Promela, we replace the normal channel constructor by assigning the global channel to the local reference in the process. This, however, is not the complete solution. For example, when the `buildChan` function from listing 5.7 is called twice it creates two distinct channels. Therefore, we initialize an array of multiple channels for each channel creation site in Kotlin. Each time this creation site is reached, we assign a new channel from the array.

Listing 5.9 shows an example of such a global channel constructor. On line 1 the array of channels is initialized. Note that all these channels are meant for the same creation site but for different calls to the containing function. Line 2 initializes a counter that keeps track of how many channels from the array have been assigned and functions as an index for the next possible assign event. Line 3 introduces a new keyword: `inline`. This keyword defines an inline function which is similar to Kotlin's inline function discussed in section 4.2.3. The reason that this inline function cannot be used to simulate normal functions from Kotlin is that Promela does not allow channels to be created in an inline function. In fact, channels can only be created at the top of a promela file or at the top of a process. The inline function introduced on line 3 is the function that replaces the normal channel initialization.

Line 4 creates an atomic block that ensures that the code in this block will run once at a time. Making sure that there are no race conditions involving the counter. Line 5 assigns the channel at the index counter to the passed `ch` channel. Subsequently, the counter is incremented at line 6. Lastly, line 7 asserts that the counter does not exceed the size of the channel array. Since it is not trivial to determine how often a function can get called, the choice is made to initialize the channel array with a fixed size. Once the counter exceeds this number, a specific error is thrown that can be used as feedback for the model generator to create a model with more channels initialized.

Listing 5.9: In Promela the solution to destroyed channels.

```
chan global_ch[4] = [0] of {int}
int global_ch1_counter = 0
inline construct_ch1(ch) {
  atomic {
    ch = global_ch1[global_ch_counter]
    global_ch = global_ch1 + 1
    assert (global_ch1 < 4)
  }
}
```

### Class properties and constructors

To support compositional types, Promela allows defining structs. Listing 5.10 shows a Kotlin example of two simple class definitions. Class *A* is introduced on line 1 and takes one argument `ch`. This class has one property (or attribute) *b* defined on line 2.

Class *B* is defined on line 5 and has also one argument. This argument `ch1` however is a class property as well, this is indicated by the `val` keyword. Class *B* has another property named `ch2`, this property is not passed but is initialized in the constructor.

Listing 5.11 demonstrates how the discussed classes *A* and *B* can be expressed in terms of Promela structs. Class *A* needs to be defined before class *B*, since A contains an instance of B (line 7). In more complex cases with lots of classes, this would mean that we need to capture the dependencies between classes and perform a topological sort to find out in what order they need to be defined. Note that a limitation to this approach (and a lack of nullability) is that circular class dependencies are not supported.

Apart from that, defining structs is straightforward: B has two properties `ch1` and `ch2` defined on line 2 and 3 respectively, and A has a property `b` defined on line 7.

Listing 5.10: A Kotlin example of a two classes.

```
class A(ch: Channel<Int>) {
  val b = B(ch)
}

class B(val ch1: Channel<Int>) {
  val ch2 = Channel<Int>()
}
```

Listing 5.11: A Promela example constructor B.

```
typedef B {
  chan ch1
  chan ch2
}

typedef A {
  B b
}
```

A constructor can be defined like any other function, with the addition of populating the corresponding struct and returning it. An example of the constructor for class *B* in Promela is given in Listing 5.12. This function (or process) takes `ch1` as an argument just like the Kotlin constructor of *B* in Listing 5.10. The `ch2` channel is created by calling the specific `construct_ch2` global channel constructor, as discussed before.

The struct this constructor constructs is constructed on lines 8, 9, and 10. Line 8 declares the reference, lines 9 and 10 assign the contents. As with any other function modelled in Promela we provide a return channel `ret` (line 3). This channel is used to return the object this constructor created (line 12).

Listing 5.13 gives an example of a constructor for *A*. Remember that, the constructor of *A* had only one property `b` obtained by calling the constructor of *B*. So, in Promela, we need to start the constructor process of *B* (line 8) and receive the created struct over the return channel (line 9). Note that in this case the return channel transfers messages of type *B* (see line 5).

While it is possible to send and receive complete structs in one go, it is not allowed to assign a complete struct. This limitation of Promela can be overcome by assigning each field individually. Similar to a deep copy, except the channels are references. As Listing 5.13 shows, `b` is not assigned directly, but all subcomponents are assigned individually (lines 12 and 13).

Listing 5.12: A promela struct constructor for class B.

```
proctype constructorB(
  chan ch1;
  chan ret
) {
  chan ch2
  construct_ch2(ch2)

  B obj
  obj.ch1 = ch1
  obj.ch2 = ch2

  ret ! obj
}
```

Listing 5.13: A Promela example constructor A.

```
proctype constructorA(
  chan ch;
  chan ret
) {
  chan callRet = [0] of {B}

  B b
  run constructorB(ch, callRet)
  callRet ? b

  A obj
  obj.b.ch1 = b.ch1
  obj.b.ch2 = b.ch2

  ret ! obj
}
```

### Call receiver

A call with a receiver object or method call is a call to a function that is part of a class. Listing 5.14 has a class *C* (line 1) which has a method `sendMSg` (line 3). This method accesses the class property `channel` (line 4). This access would, however, not work in Promela. As in Promela, one can not couple a function to a struct like a method is coupled to a class, and that function, therefore, has no access to the context of the class or struct. In other words, the implicit reference to `this` is missing.

To solve this problem, the class context can be passed as a parameter. Listing 5.15 shows how a Kotlin method can be transformed to be a compatible function. The `sendMsg` function is moved out of the class and has an additional parameter `self` that contains the class context of *C* (line 5). On line 6, the `channel` property can be accessed through the

`self` parameter. The call to this method has to change as well, the receiver object is now passed as an argument, resulting in the call on line 11.

Listing 5.14: A Kotlin example a call with receiver.

```
class C {
  val channel = Channel<Int>()
  suspend fun sendMsg() {
    channel.send(0)
  }
}

suspend fun main() {
  val c = C()
  c.sendMessage()
}
```

Listing 5.15: A Kotlin example of passing class context.

```
class C {
  val channel = Channel<Int>()
}

suspend fun sendMsg(self: C) {
  self.channel.send(0)
}

suspend fun main() {
  val c = C()
  sendMessage(c)
}
```

#### Nested expressions

In Kotlin it is possible and common to nest expressions such as calls (function composition). To model this in Promela we need to un-nest these cases such that the result is a linear order of statements in the order of execution. This will become clear in the Listings 5.16 and 5.17. Note that both listings semantically do the same.

Listing 5.16: A Kotlin example of nested calls.

```
fun main() {
  return doSome(otherFun(foo()), bar())
}
```

Listing 5.17: A Kotlin example of un-nested calls.

```
fun main() {
  val v4 = foo()
  val v3 = otherFun(v4)
  val v2 = bar()
  val v1 = doSome(v3, v2)
  return v1
}
```

## 5.3 Translating Kotlin to Promela

Having clarified what elements of Kotlin are supported and how they can be expressed in Promela, this section will delve into the translation process. This process has five distinct phases, which will be discussed in that order. First, a callgraph with some additional information will be initialized (section 5.3.1). Second, reference usages will be linked to origins, and structs will be generated where necessary (section 5.3.2). Third, nested expressions will be un-nested (section 5.3.3). Fourth, unnecessary nodes in the graph will be removed (section 5.3.4). Fifth, Promela code will be generated from the graph (section 5.3.5).

### 5.3.1 Graph initialization

The initialization phase sets up a call graph based on the Kotlin repository under test. It creates a node for every function by traversing the PSI tree. A `FunctionNode` represents a function and basically has three elements: a parameter list, a return type, and a list of actions. This phase only captures channel parameters, parameters of other types will be gradually added in the linking phase (section 5.3.2). The return type is initially set to Unit (or void), this return type will be added in the linking phase as well. The list of actions represents the body of a function. An action can be a statement or expression of interest for our analysis, an action is one of the following:

- **Call**: a call action represents a Kotlin call expression. It contains a callee `FunctionNode`, a return type, and an argument list. As with the function nodes, the return type will be discovered in the linking phase. An argument can be one of two kinds: a reference argument or an action argument. A reference argument is a reference passed to a function like `doSomething(arg)`. An action argument is an expression that evaluates to an argument. For example, a nested call like `doSomething(getArg())`. Just like return types, reference will be explored in the linking phase. Therefore, only action arguments are added in this phase. As discussed in section 5.2.2, a potential call receiver object will be treated as an argument and added to the argument list accordingly.

- **Async call**: an async call action represents the launch of a coroutine. A `FunctionNode` will be created for the launched coroutine, which will be the callee of the async call action.

- **Out-of-scope call**: an out-of-scope call action represents a call to library code that is not part of the analysis scope. That means that this call will not have a callee `FunctionNode`, but can potentially have action arguments with calls to code that is in scope. For instance: `outOfScopeCall(inScopeCall())`.

- **Channel init**: a channel init action represents the initialization of a channel. The buffer size of the channel is captured as well.

- **Channel send and receive**: send and receive operations on a channel.

- **Property assignment**: a property assignment is the declaration and initialization of a variable like: `val prop = getProp()`. Note that this property can be either a class property or a local variable in a function.

- **Property access**: a property access action is the retrieval of a class property value like: `instance.prop`. This action contains two values: the property name or identifier, and an argument, which can be an action argument or reference argument.

Additionally, the initialization phase creates a function node for each class. This node represents the constructor of that class. The parameters of this class are added to the parameter list of the function node and the class properties are added as property assignments to the action list.

### 5.3.2 Reference linking

The linking phase makes sure every channel operation is performed on the right channel at the right moment. In other words, this phase is responsible for modelling the correct dataflow from channel initialization to channel operation. For each channel operation (send and receive actions) we track its channel reference back to all possible origins. During the backtracking process, references are added, structs are evolved, and return types are set.

This is done by recursively exploring *operands*. In other words, any part of a statement or expression that requires a value. This can be an argument for a function call, an expression to be returned, or the value that is assigned to a variable. In the context of this analysis, these operands can have either a reference expression or an action that evaluates to a value like a function call. If an operand contains a reference expression like: `someFunCall(arg)`, we resolve the origin of the reference. This origin can be one of three scenarios: a function *parameter*, a *local variable*, or the *this instance*. An operand can also contain an action, the prototype supports a *call* or *property access*.

#### Parameter

One possible origin of a reference is a parameter. Listing 5.18 shows an argument (operand) `arg` marked in red. Its origin is a function parameter marked in blue. Now, the following few steps are taken. First, a reference argument is created and added to the call action. Second, if the parameter is not yet present in the function node, it is added as well. Subsequently, the reference argument and origin parameter will be linked.

Since a backtracking process starts at a channel operation, this very argument will eventually lead to this operation. Therefore, to find all possible initialization sites of this channel, we need to visit all callsites of function `foo` and their corresponding operands. Listing 5.19 shows two of these callsites. Since the origin parameter was the 2nd parameter of function `foo`, the 2nd arguments (operands) of the callsites are to be explored (marked in green). Note that these arguments can be either a reference argument (line 3) or an action argument (line 8).

Listing 5.18: Function parameter is the origin of an argument arg.

```
1 fun foo(x: Int, arg: ParamType) {
2   someFunCall(arg)
3 }
```

Listing 5.19: Function `foo` is called twice resulting in two arguments to be explored. One reference argument `bar` and one action argument `baz()`.

```
1 fun fooCaller1() {
2   // Other code
3   foo(0, bar)
4 }
5
6 fun fooCaller2() {
7   // Other code
8   foo(y, baz())
9 }
```

Listing 5.20: Function parameter is the origin of an argument arg.

```
fun foo() {
  val arg = baz()
  someFunCall(arg)
}
```

### Local variable

Another possible reference origin is a local variable. Listing 5.20 shows such a case. The reference operand we are currently exploring is marked in red, the origin of this reference is a local variable defined on line 2 (marked in blue). The local variable and argument will be linked.

The operand assigned to variable `arg` has to be explored next and is marked in green.

### This instance

Another possibility is a reference to *this* instance. It is the instance context of the class the current method is a member of. Listing 5.21 shows a possible example in Kotlin. On line 2 there is a *this* reference and it is passed as receiver of the call to foo(). This receiver is marked in green and would be the next operand for exploration.

However, the *this* operand cannot be linked to anything local. Now we start to see the benefit of treating call receivers as arguments instead. During the initialization phase (section 5.3.1) the code example will be desugared to Listing 5.22. Semantically, the code does the same, but now the instance context is explicitly passed. Therefore, we can link the operand to a parameter and continue exploring the argument in the call site on line 5. As if it were like any other call with an argument.

Listing 5.21: A reference to *this* instance as operand. The receiver of the call expression is the next operand to explore.

```
fun foo() {
  someFunCall(this)
}
fun fn() {
  bar.foo()
}
```

Listing 5.22: A reference to *self* instance passed as parameter. Origin is marked as blue, the next operand to explore is marked in green.

```
fun foo(self: ThisClass) {
  someFunCall(self)
}
fun fn() {
  foo(bar)
}
```

### Call

As with every operand that we explore, the goal is to find the origin of the value. When the operand is a call, the value is provided by the return statement(s) of the called function. Listing 5.23 provides an example of a function call as operand (marked in red). Since a

Listing 5.23: The function call as operand is marked in red. The next operand to explore is marked in green.

```
fun foo() {
  someFunCall(bar())
}
fun bar() {
  // Other code
  return baz
}
```

nested call is added in the initialization phase as *action argument*, we do not have to link it at this stage. A reference argument will be added and linked during the *un-nesting* phase. However, the return type of the called function `bar` and the call action `bar()` will be set. This not only provides info about the return type but also marks the function and call as important, preventing them from being pruned later.

The backtracking process is continued by exploring the operand from the return statement of the called function (marked in green).

### Property access

Listing 5.24 provides an example of a property access from within a class. At first glance, this might look like a reference operand; however, the reference `prop` on line 5 is implicitly `this.prop`. That results in Listing 5.25, where *this* is added explicitly. The operand is in both listings marked in red. However, in listing 5.25 it becomes clear that the operand is a property access where property `prop` is selected from object `this`.

As with other operands, the origin of the value needs to be explored. Meaning that the next operand to explore becomes the operand of the property assign (line 2). However, since this property is not locally accessed but is part of an instance, the origin of the instance needs to be found as well. Therefore, in the case of a property access there are two next operands. As shown in listing 5.25 there are two green highlighted operands, one for the origin of the instance and one for the origin of the property. Now since this occurrence proves that the property `prop` from class `Claz` is important, it is added to the struct of `Claz`.

Listing 5.24: Property access from within class.

```
class Claz() {
  val prop = bar()
  fun foo() {
    // Other code
    someFunCall(prop)
  }
}
```

Listing 5.25: Property access from within class. Implicitly access property from *this*.

```
class Claz() {
  val prop = bar()
  fun foo() {
    // Other code
    someFunCall(this.prop)
  }
}
```

### 5.3.3 Un-nesting

Once linking is completed, the graph will be un-nested. This has already been discussed in section 5.2. Listings 5.16 and 5.17 should give a clear example. However, there is one important difference: although all nested calls will be moved out of the parent call, only nested calls that have a return type will be assigned to a variable and passed to the parent call.

### 5.3.4 Pruning

During the initialization and linking phase, function parameters have been added and return types have been set. Since even class context is modelled as a function parameter, everything a function can have an effect on has to be passed as a parameter. Therefore, when a function has no parameters, we conclude that this function does not have any side effects. Additionally, when that same function also has no return type set, we can conclude that no channel (or struct containing a channel) is returned. Therefore, within the context of this model, all calls to this function are pointless and should be removed.

Having removed all pointless calls, we are left with only calls that pass and/or return a channel. The result of this is that separate concurrent parts of the codebase which do not interact concurrently are now not connected in the graph. In other words, the graph is split up in small subgraphs that work in isolation. This allows for each subgraph to be evaluated separately, keeping the complexity of the models as small as possible.

Lastly, all functions that do not call and are not called (after pruning) are removed.

### 5.3.5 Generating Promela

When we removed all unnecessary elements from the graph, put all nested code into a linear sequence of operations, and linked all references where needed, the graph is ready to be written to Promela. In this form each element can easily be written to a string following the methods of section 5.2.

We start by listing all function nodes that are not called. Each of such nodes is a starting point for one model. The model is built by recursively traversing the graph from that point and concatenating it into a string at each step. Each element has its specific string template that represents that element in Promela.

## 5.4 SPIN model verification

SPIN is a tool that verifies concurrent models written in Promela. [21] It does so by exploring all possible execution orders. When it has explored all possible execution orders or when a given time budget expires, the model is classified as deadlock-free.

## 5.5 Validation: Accuracy

To validate the technique, it is important to know how well it detects deadlocks and if it scales well with larger codebases. This section will go into detail about the accuracy of the developed analysis technique. The next section (5.6) will go into detail about the scalability.

Section 5.5.1 will explain the method used to test the prototype. Section 5.5.2 will briefly explain the used test data. Section 5.5.3 goes into detail about all possible errors that can occur in a single test case. And lastly, section 5.5.4 reveals the results.

### 5.5.1 Method

To quantify the accuracy of the analysis method, test data is required. Although the prototype works on some parts of a real-world repository, it is not yet powerful enough to reliably extract models from opensource repositories. Additionally, it is not trivial to find repositories of which it is known to have clear channel deadlocks, and if so, how many. The opposite is true as well, for a given repository with channels, it is impossible to tell that it is deadlock-free. In other words, there is no ground truth.

What is necessary is a set of code samples that tests the prototype within its designed constraints and has a known ground truth. Large language models can be part of the solution. A large set of test cases can easily be generated by carefully crafting specific prompts. This approach allows us to create test cases that take the design constraints into account. The LLM can be instructed to not use *mutexes* or *inheritance*. Additionally, tests can be created that specifically test features that should be supported like *classes* and *properties*. Moreover, the LLM can be instructed to include a deadlock or not.

Given the stochastic nature of LLMs, not every generated test case will adhere to the rules that have been set and may not even compile. Additionally, instructing an LLM to create code that deadlocks doesn't guarantee that it actually does. Therefore, some checks need to be done on the tests. This is done by compiling and running tests individually. If the test does not compile or generates a runtime error it is classified as unfit. If the test run takes longer than 10 seconds, it is classified as a test that deadlocks. Given that the tests do not make IO operations and don't do heavy computations and, assuming that the LLM does not create an infinite loop, 10 seconds should be enough for a program to finish. However, keep in mind that when a program does finish within 10 seconds, it does not automatically mean that there is no deadlock. As shown in section 5.1 a deadlock can occur non-deterministically. Therefore, when a program does finish within 10 seconds and the prompt instructed the LLM to not include a deadlock, we will assume it is deadlock-free. In contradicting cases manual inspection is required.

### 5.5.2 The benchmark

The transformer GPT-4o has been used to generate the test set. [25] The prompt, used to generate the tests, specifically names Kotlin features to include and not to include. For each prompt a few random integers are generated to specify the number of functions, classes, coroutines, and channels. This helps to create a variety of tests in terms of size and com-

plexity. The prompt template used to generate the tests is displayed in appendix A. The generated test set can be found in a public repository [4].

### 5.5.3 Testing errors

In the process of generating, validating, and applying tests, errors can occur. Table 5.1 shows the error count per error type for all tests. The following errors can occur during a test:

- **KotlinCompileError** indicates that the test sample does not compile and means that the LLM created an invalid test.
- **KotlinRuntimeError** indicates that during the initial deadlock check by running the sample a runtime error occurred. This suggests as well that the test is invalid.
- **GraphError** means that an error occurred during the creation and manipulation of the deadlock graph. This can arise when certain complex Kotlin constructs that were not part of the prototype design happen to be part of the test. Or it could indicate a bug in the prototype implementation.
- **WriteToPromelaError** occurs during the phase where a model is converted to actual Promela code. Note that since multiple models can be extracted from one code base, this error can occur per extracted model per test. Usually, this error indicates that a certain reference is not resolved correctly. This can happen when a property is initialized by a complex expression like try/catch or if/else (yes, if/else can be an expression in Kotlin).
- **InvalidModelError** occurs during SPIN execution and means that there is a Promela compile error. Basically, it means that there is a syntactic or type error in the provided Promela model.
- **UnknownResultError** occurs after SPIN execution and means that the provided output by SPIN is not recognised.
- **Success** means that there was no error and that the test has been classified.

### 5.5.4 Results

This section presents the accuracy results. The results are summarized in a confusion matrix (table 5.2) including True Positives (TP), False Positives (FP), True Negatives (TN), and False Negatives (FN). It should be noted that the ground truth for negative cases may deviate due to non-deterministic deadlock cases. More precisely, a code sample might have an execution order that leads to a deadlock. However, during the validation of the test case, the sample finished with another execution order.

From the results, it immediately stands out that there is a relatively large number of false positives. Upon closer examination of the false positive cases, a recurring pattern emerged:

Table 5.1: Count of errors per error type for all tests.

| Error Type | Count |
|---|---|
| KotlinCompileError | 85 |
| KotlinRuntimeError | 17 |
| GraphError | 45 |
| WriteToPromelaError | 4 |
| InvalidModelError | 12 |
| UnknownResultError | 0 |
| Success | 837 |
| Total | 1000 |

Table 5.2: Confusion matrix for a total of 837 successful tests, the following numbers are displayed: true and false positive counts (left column), true and false negative counts (right column).

| | | Predicted | |
|---|---|---|---|
| | | Yes | No |
| Act | Yes | 470 | 67 |
| | No | 116 | 184 |

Table 5.3: The fraction of correctly classified tests. Yes: $TP/(TP+FN)$, No: $TN/(TN+FP)$. These values are given for: all tests, tests with loops, and tests without loops.

| Deadlock | All | Loops | No loops |
|---|---|---|---|
| Yes | 0.875 | 0.806 | 0.920 |
| No | 0.613 | 0.476 | 0.841 |

the majority of these tests contained loops. To better illustrate the impact that loops have on the performance of our prototype, two additional confusion matrices are presented that display the results both with loops (table 5.4) and without loops (table 5.5).

When loops are present, the performance of the analysis technique degrades for both negative and positive cases. In contrast, tests without loops perform substantially better. The three confusion matrices are, however, hard to compare due to a different number in ground truth values. Therefore, we present table 5.3. This table provides a clearer overview of the influence loops have on prediction, by displaying the ratio of correctly classified cases per label (*yes* or *no*), per test set (*All*, *Loops*, and *No loops*).

The reason for this gap in performance is that the prototype does not support conditional branching like *if/else* and *loops*. It captures all statements and expressions from the body of the loop and adds them to the model, as if the body of the loop was a part of the body of the function without the loop. Effectively, taking the loop out of the loop. Therefore, the body of the loop only runs once. If this body contained a send or receive operation, we possibly model the wrong number of operations made on this channel, resulting in false positives.

Other false positives and false negatives can result from the model checker timing out, bugs in the prototype implementation, or the presence of certain constructs in the Kotlin language that were not intended to be part of the test.

Table 5.4: Confusion matrix for 398 tests that contain loops.

| | | Predicted Yes | Predicted No |
|---|---|---|---|
| Act | Yes | 170 | 41 |
| Act | No | 98 | 89 |

Table 5.5: Confusion matrix for 439 that do not contain loops.

| | | Predicted Yes | Predicted No |
|---|---|---|---|
| Act | Yes | 300 | 26 |
| Act | No | 18 | 95 |

### 5.5.5 Threats to validity

While we believe the method of measuring accuracy gives good insight into the capabilities of the proposed prototype, there are some obvious flaws to this method. First of all, there is no accurate ground truth. While we can say with some confidence that a case labelled as deadlock is in fact a deadlock, due to the small sizes of the programs they should not be able to take longer than 10 seconds to execute. However, cases that are labelled as deadlock-free might still deadlock due to non-determinism. Moreover, recognizing deadlocks with a human eye is challenging and error-prone. While this emphasizes the need of a deadlock detector, it makes it difficult to accurately validate them.

Secondly, while using an LLM to create a controlled environment allowed us to validate the prototype, it probably created a biased environment only representing a subset of real world scenarios. Nevertheless, we are confident that this method has given us valuable insight into the capabilities and shortcomings of the prototype.

## 5.6 Validation: Scalability

In this section, we will go into detail about the scalability of the prototype. In other words, how does it perform on bigger repositories.

### 5.6.1 Method

Due to the constraints of the prototype, it is challenging to run it on large repositories since we have no control over what language features it contains. Nevertheless, it is important to prove that: it can detect a deadlock in a large real world repository, and it completes within a reasonable time. Therefore, from the list of the runBlocking test repositories (section 4.4), we selected the biggest repository with channels that the prototype successfully extracts models from. The selected repository is *DuckDuckGo* [12]. This repository has 351k lines of code and uses channels and coroutines for parts of their concurrency needs.

To validate that a deadlock can be detected in this repository, a deadlock bug is seeded [1].

### 5.6.2 Results

The prototype has extracted 18 models from the *DuckDuckGo* code base of which 15 were valid. One of these models correctly predicted our seeded deadlock.

Table 5.6: Execution time per phase for an average of three executions on the *DuckDuckGo* repository.

| Phase | Time (s) |
|---|---|
| Graph initialization | 108.12 s |
| Reference linking | 1.10 s |
| Un-nesting | 0.29 s |
| Pruning | 0.09 s |
| Generating and verifying models | 13.20 s |
| Total | 122.80 s |

Table 5.6 presents the (averaged) duration of each phase discussed in section 5.3. It is clear that the initialization phase takes the longest, the reason for this is the many accesses the PSI system leading to reads from disk. Validating 15 models took around 13 seconds.

## 5.7 Discussion and Future work

While having shown that the prototype works successfully within the confines of its design, it is clear that there are many painpoints that need to be addressed before it can become a useful tool.

First and foremost, the prototype fails before classification on most open source repositories. Often due to encountering unknown constructs or missing references. Efforts have been made that allow the process to continue even when encountering these cases. However, an improvement in the implementation would be appropriate.

Secondly, supporting more language features like conditional branching and mutability is a logical next step, with priority for loops. Modelling these constructs in Promela has been addressed before by *GOMELA* [11]. This approach, however, considers if/else constructs as statements while Kotlin allows them to be expressions as well. This poses new challenges that will have to be explored further.

Third, deadlocks can happen with more than only channels. Mutexes, semaphores, joins, and coroutineScope are all primitives that synchronize coroutines and can partake in a deadlock. Therefore, supporting these constructs should improve the accuracy of the proposed tool as well. There is a similarity between channels and mutexes and the implementation of these constructs has been tackled by *GOMELA* as well.

Lastly, warning that a deadlock possibly occurs in a repository of 350,000 lines is not very helpful. For this tool to become useful we need to provide the developer with feedback about the elements involved in the deadlock and how it got to that state. Luckily, SPIN provides a *trail* that can be used to deduce the steps taken that ended up at the deadlocked state. This trail, however, references the Promela model and not the Kotlin code. Therefore, transforming this trail such that it is understandable in the context of the Kotlin codebase poses a new challenge as well.

# Chapter 6

# Conclusion

This is the first work that addresses deadlocks in Kotlin coroutines. Two static analysis tools have been proposed: one tool that tackles the nested *runBlocking* antipattern, and one tool that represents the first steps towards a general deadlock detection tool for Kotlin.

The *runBlocking* tool successfully detected problems in open source repositories, which lead to their resolution. Moreover, the *runBlocking* analysis has been integrated in JetBrains flagship IDE: IntelliJ IDEA.

The prototype for static deadlock detection has demonstrated to be effective within the confines of its design. Two novel contributions allowed the prototype to model classes, attributes, and methods in Promela. We believe this prototype to be a significant step towards general deadlock detection for Kotlin coroutines.

# Bibliography

[1] Bob Brockbernd. Deadlock bug seed in duckduckgo repo, 2024. URL https://github.com/duckduckgo/Android/commit/50195a32e8b44c6eef6383ba39e28610a4683ed9.

[2] Bob Brockbernd. [bug] runblocking calls from inside coroutine, Apr 2024. URL https://github.com/duckduckgo/Android/issues/4409.

[3] Bob Brockbernd. Fix runblocking in coroutines, Apr 2024. URL https://github.com/home-assistant/android/pull/4340.

[4] Bob Brockbernd. Deadlock test repository, 2024. URL https://github.com/bbrockbernd/DeadlockTestRepo.

[5] Bob Brockbernd. Runblocking inspection implementation, Jun 2024. URL https://github.com/JetBrains/intellij-community/commit/ea8296d53925ec87ddbee66f37412793d3fbdb14.

[6] Bob Brockbernd, Nikita Koval, Arie van Deursen, and Burcu Kulahcioglu Ozkan. Understanding concurrency bugs in real-world programs with kotlin coroutines. In *38th European Conference on Object-Oriented Programming (ECOOP 2024)*. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2024.

[7] James Brotherston, Paul Brunet, Nikos Gorogiannis, and Max Kanovich. A compositional deadlock detector for android java. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 955–966. IEEE, 2021.

[8] Sam Cooper. How I fell in Kotlin's runblocking deadlock trap, and how you can avoid it, Oct 2023. URL https://betterprogramming.pub/how-i-fell-in-kotlins-runblocking-deadlock-trap-and-how-you-can-avoid-it-db9e7c4909f1.

[9] Sjoerd Cranen, Jan Friso Groote, Jeroen JA Keiren, Frank PM Stappers, Erik P De Vink, Wieger Wesselink, and Tim AC Willemse. An overview of the mcrl2 toolset and its recent advances. In *Tools and Algorithms for the Construction and Analysis of Systems: 19th International Conference, TACAS 2013, Held as Part of the European*

*Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings 19*, pages 199–213. Springer, 2013.

[10] Nicolas Dilley and Julien Lange. Bounded verification of message-passing concurrency in go using promela and spin. *arXiv preprint arXiv:2004.01323*, 2020.

[11] Nicolas Dilley and Julien Lange. Automated verification of go programs via bounded model checking. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1016–1027. IEEE, 2021.

[12] DuckDuckGo. Duckduckgo android repository, 2024. URL `https://github.com/duckduckgo/Android`.

[13] Roman Elizarov. Deadlocks in non-hierarchical csp, Jan 2019. URL `https://elizarov.medium.com/deadlocks-in-non-hierarchical-csp-e5910d137cc`.

[14] Roman Elizarov, Mikhail Belyaev, Marat Akhin, and Ilmir Usmanov. Kotlin coroutines: design and implementation. In *Proceedings of the 2021 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, pages 68–84, 2021.

[15] Mahdi Eslamimehr and Jens Palsberg. Sherlock: scalable deadlock detection for concurrent programs. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 353–365, 2014.

[16] Google. Processes and threads overview, June 2023. URL `https://developer.android.com/guide/components/processes-and-threads.html`.

[17] Hans Hüttel, Ivan Lanese, Vasco T Vasconcelos, Luís Caires, Marco Carbone, Pierre-Malo Deniélou, Dimitris Mostrous, Luca Padovani, António Ravara, Emilio Tuosto, et al. Foundations of session types and behavioural contracts. *ACM Computing Surveys (CSUR)*, 49(1):1–36, 2016.

[18] Nikita Koval, Dmitry Tsitelov, and Roman Elizarov. Dl-check: Dynamic potential deadlock detection tool for java programs. In *Tools and Methods of Program Analysis: 4th International Conference, TMPA 2017, Moscow, Russia, March 3-4, 2017, Revised Selected Papers 4*, pages 64–76. Springer, 2018.

[19] Nikita Koval, Dan Alistarh, and Roman Elizarov. Fast and scalable channels in kotlin coroutines. In Maryam Mehri Dehnavi, Milind Kulkarni, and Sriram Krishnamoorthy, editors, *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming, PPoPP 2023, Montreal, QC, Canada, 25 February 2023 - 1 March 2023*, pages 107–118. ACM, 2023. doi: 10.1145/3572848.3577481. URL `https://doi.org/10.1145/3572848.3577481`.

[20] Nikita Koval, Alexander Fedorov, Maria Sokolova, Dmitry Tsitelov, and Dan Alistarh. Lincheck: A practical framework for testing concurrent data structures on jvm. In *International Conference on Computer Aided Verification*, pages 156–169. Springer, 2023.

[21] Bell Labs. Verifying multi-threaded software with spin. URL https://spinroot.com/spin/whatispin.html. Accessed: 2024-10.

[22] Julien Lange, Nicholas Ng, Bernardo Toninho, and Nobuko Yoshida. A static verification framework for message passing in go using behavioural types. In *Proceedings of the 40th International Conference on Software Engineering*, pages 1137–1148, 2018.

[23] Mayur Naik, Chang-Seo Park, Koushik Sen, and David Gay. Effective static deadlock detection. In *2009 IEEE 31st International Conference on Software Engineering*, pages 386–396. IEEE, 2009.

[24] Nicholas Ng and Nobuko Yoshida. Static deadlock detection for concurrent go by global session graph synthesis. In *Proceedings of the 25th International Conference on Compiler Construction*, pages 174–184, 2016.

[25] OpenAI. Hello gpt-4o, May 2024. URL https://openai.com/index/hello-gpt-4o/.

[26] R Rajesh Kumar, Vivek Shanbhag, and KV Dinesha. Automated deadlock detection for large java libraries. In *Distributed Computing and Internet Technology: 17th International Conference, ICDCIT 2021, Bhubaneswar, India, January 7–10, 2021, Proceedings 17*, pages 129–144. Springer, 2021.

[27] Malavika Samak and Murali Krishna Ramanathan. Multithreaded test synthesis for deadlock detection. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, pages 473–489, 2014.

[28] Anirudh Santhiar and Aditya Kanade. Static deadlock detection for asynchronous c# programs. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 292–305, 2017.

[29] Vivek K Shanbhag. Deadlock-detection in java-library using static-analysis. In *2008 15th Asia-Pacific Software Engineering Conference*, pages 361–368. IEEE, 2008.

[30] Martin Sústrik. Structured concurrency, Feb 2016. URL https://250bpm.com/blog:71/.

[31] Dmitry Tsitelov. Data race detector, 2017. URL https://opensource.devexperts.com/display/DRD.

# Appendix A

# Prompt

The prompt used to generate the test cases for the validation of the static deadlock detection. The ${} expressions inject a value into the prompt. This allows for variation in number of function channels etc.

```
I need to test a channel deadlock detection algorithm for Kotlin coroutines.
I want you to create a test sample that ${dlString} contain a DEADLOCK.
The deadlock (if any) should arise from channels and coroutines.
I want the example to consist of:
- ${config.nFunctions} different functions
- ${config.nChannels} different channels
- ${config.nCoroutines} different coroutines
- ${config.nClasses} different classes

You ARE ALLOWED to use basic Kotlin constructs and coroutine primitives.
A few examples are:
- functions and suspend functions
- dot qualified expressions
- class properties
- local variables
- unbuffered and buffered channels
- channel init, send and receive
- return values
- function composition (nested calls)
- runBlocking and launch builder
- coroutineScope

You ARE NOT ALLOWED to use more complex features like:
- joins
- async builder
- lists, arrays or other datastructures
- mutability
```

```
- nullability
- for (i in channel)
- flow
- lateinit
- lazyval
- inheritance
- lambdas with arguments
- mutexes
```