Understanding Concurrency Bugs in Real-World Programs with Kotlin Coroutines

Bob Brockbernd
Delft University of Technology, The Netherlands

Nikita Koval
JetBrains, Amsterdam, The Netherlands

Arie van Deursen
Delft University of Technology, The Netherlands

Burcu Kulahcioglu Ozkan
Delft University of Technology, The Netherlands

Abstract

Kotlin language has recently become prominent for developing both Android and server-side applications. These programs are typically designed to be fast and responsive, with asynchrony and concurrency at their core. To enable developers to write asynchronous and concurrent code safely and concisely, Kotlin provides built-in *coroutines* support. However, developers unfamiliar with the coroutines concept may write programs with subtle concurrency bugs and face unexpected program behaviors. Besides the traditional concurrency bug patterns, such as data races and deadlocks, these bugs may exhibit patterns related to the coroutine semantics. Understanding these coroutine-specific bug patterns in real-world Kotlin applications is essential in avoiding common mistakes and writing correct programs.

In this paper, we present the first study of real-world concurrency bugs related to Kotlin coroutines. We examined 55 concurrency bug cases selected from 7 popular open-source repositories that use Kotlin coroutines, including IntelliJ IDEA, Firefox, and Ktor, and analyzed their bug characteristics and root causes. We identified common bug patterns related to asynchrony and Kotlin's coroutine semantics, presenting them with their root causes, misconceptions that led to the bugs, and strategies for their automated detection. Overall, this study provides insight into programming with Kotlin coroutines concurrency and its pitfalls, aiming to shed light on common bug patterns and foster further research and development of concurrency analysis tools for Kotlin programs.

2012 ACM Subject Classification Computing methodologies → Concurrent programming languages; Software and its engineering → Software testing and debugging

Keywords and phrases Kotlin, coroutines, concurrency, asynchrony, software bugs

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2024.8

Acknowledgements We thank our shepherd, Elisa Gonzalez Boix, and anonymous reviewers for their suggestions for improving the paper.

1 Introduction

Kotlin is a cutting-edge programming language that has recently become a primary language for Android development. Its modern syntax, seamless interoperability with Java, and enhanced features have positioned Kotlin as the preferred language for creating mobile and server-side applications for many developers. A standout feature amongst Kotlin's diverse functionalities is the built-in *coroutines* [14] support, which significantly simplifies asynchronous programming. Coroutines offers developers a streamlined approach to handling background tasks, thus enabling more intuitive and readable code.

Kotlin coroutines enables writing asynchronous code in a sequential style, thus avoiding the complexity of multithreaded programs. One may think of coroutines as lightweight threads. Following the notion of coroutines in the literature [12], Kotlin coroutines can

suspend and resume their execution, and they do that through the *suspending* functions. Suspending functions (marked with the `suspend` keyword in Kotlin) are similar to *async* functions in other asynchronous languages and frameworks [37, 33], which encapsulate asynchronous computations but look like synchronous code, and calling them looks similar to calling regular functions. A suspending function indicates that it can be suspended in the middle of computation and resumed later. By suspending and resuming at predefined points, they yield computing resources to other coroutines and coordinate their execution. For example, when a suspending function encounters a long-running task and suspends, the underlying thread does not block but takes another coroutine to execute.

While coroutines simplify writing asynchronous programs and increase performance by asynchronously running long-running tasks, asynchronous programming comes with additional challenges. Besides the inherent concurrency non-determinism of multithreading, which causes traditional classes of concurrency bugs (such as data races, deadlocks, order violations, and atomicity violations), utilizing coroutine requires developers to reason about the asynchronous interactions between the concurrent components due to coroutine semantics, synchronous/asynchronous execution contexts, suspension and resumption of coroutines. Developers unfamiliar with the Kotlin coroutine semantics can use coroutines and suspending functions improperly, resulting in subtle concurrency bugs.

Kotlin is a relatively new language, so we know little about common misconceptions and bug patterns in programs with Kotlin coroutines. This work aims to shed light on them.

Our contribution. In this study, we explored real-world concurrency bugs in programs that use Kotlin coroutines. We collected concurrency bugs from popular open-source code repositories and identified common patterns in these bugs, their root causes, and misconceptions that might cause them. The results are available in the following GitHub repository [6].

Our analysis shows that some concurrency bugs are related to *bridging synchronous and asynchronous executions*. While asynchronous functions can only be called on a coroutine and the syntax of suspendable functions helps *function coloring* by marking asynchronous functions, composing synchronous parts of the program with asynchronous functions remains a challenge to programmers inexperienced in asynchronous programming. As a quick remedy for calling asynchronous functions from synchronous functions, they may call these functions in blocking coroutines, which in turn may affect the program's performance or even introduce deadlocks in case of *nested blocking calls*. On the other hand, when calling synchronous functions from asynchronous functions, they should be aware that the called function may run in an asynchronous context. Calling asynchronous functions from such functions can demand manual bookkeeping of coroutine scopes or omitting one of Kotlin's core concurrency principles: structured concurrency [34, 9]. In addition to *nested blocking calls*, we identified classes of bug patterns where the developers may violate structured concurrency by improper *scope passing*, *querying asynchronous objects*, *synchronization with asynchronous objects*, and improper *coroutine exception handling*.

In summary, our work makes the following contributions:

- We present the first (to the best of our knowledge) comprehensive study of real-world concurrency bugs in programs that use Kotlin coroutines, examining 55 concurrency bugs selected from 7 popular open-source repositories that use Kotlin coroutines.
- We identify common bug patterns related to Kotlin coroutines, presenting real-world examples and possible corrections for each, providing root causes and misconceptions that lead to these bugs, and discussing strategies for their automated detection.

Moreover, we communicated our findings to Kotlin developers. Our initial discussions show that our findings align with some of their observations, e.g. [29, 27]. We are in contact with them to develop inspection tools for identified bug patterns and have already contributed to IntelliJ IDEA with an inspection that detects one of the identified bug patterns [7].

Impact. We envision our findings contributing to developing reliable Kotlin programs targeting multiple audiences. They can help (i) Kotlin programmers better understand concurrency bugs and write correct programs, (ii) increase the awareness of programmers using Kotlin libraries about the potential asynchrony in the functions they use, (iii) provide insights to the Kotlin language team about possible misconceptions of programmers and (iv) researchers develop suitable concurrency analysis and testing tools for Kotlin programs.

2 Background on Kotlin Coroutines

Essentially, coroutines are lightweight threads that are relatively cheap to suspend and resume. They also support efficient cancellation, which, in sum, makes them very powerful for asynchronous programming. This section briefly introduces coroutines in Kotlin. discussing important differences compared to traditional threads and the features necessary to understand the bug patterns we present in our study.

Launching a coroutine. A coroutine is a computation unit not bound to any particular thread. Instead, coroutines run on threads and reuse them. When a coroutine gets suspended (pauses in the middle of computation), the underlying thread does not park but takes another coroutine and executes it, utilizing resources more efficiently. In this essence, coroutines can be considered a framework for managing expensive threads.

When launching a new coroutine, we can specify a coroutine dispatcher, which determines how to schedule the coroutine. The default dispatcher (`Dispatchers.Default`) is essentially a thread pool, where the number of threads is bound to the number of CPUs. It is also possible to launch coroutines on the `Main` dispatcher, ensuring that they are executed on the Main (UI) thread, or the `IO` dispatcher when the code does not compute something but blocks the running thread with an I/O operation.

Listing 1 shows an example with one coroutine launching another and printing "`Hello`", and the second suspending for one second and printing "`World!`". The `main()` function starts with a `runBlocking` call, which bridges the non-coroutine and coroutine worlds, blocking the current thread for the duration of the coroutine it runs. The `launch` call starts a new coroutine concurrently with the rest of the code on `Dispatchers.Default` coroutine dispatcher, which means this coroutine will run on a shared pool of threads. The `delay` call in the launched coroutine suspends it for one second. As result, this code prints "`Hello`" followed by "`World!`".

Listing 1 Launching a new coroutine in Kotlin.

```
fun main() = runBlocking { // launches a coroutine on this thread
    launch(Dispatchers.Default) { // launch a new coroutine
        printWorldWithDelay()
    }
    println("Hello")
}

```

```
suspend fun printWorldWithDelay() {
    delay(1000L) // non-blocking delay for 1 second
    println("World!")
}
```

Suspending functions. To utilize coroutines, Kotlin provides the *suspending function* concept. Suspending functions, marked with `suspend` modifier, can be paused and resumed later without blocking the underlying thread. In Listing 1, `printWorldWithDelay()` and `delay(..)` are suspending functions, which might pause (in this case, the underlying thread switches to executing another coroutine). A suspending function can only be called from another suspending function, providing a structured way to write asynchronous and non-blocking code.

To summarize, the `suspend` keyword in Kotlin is used to mark a function that can be asynchronously completed – it can suspend its execution at some point, being resumed where it left off later, without blocking the underlying execution thread.

Structured concurrency. 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, while cancellation of one of the coroutines in a scope instantly aborts the others within the scope.

In Listing 1, `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. This is why the `runBlocking` call does not complete until the second coroutine that prints "`World!`" finishes.

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, while canceling the scope results in the cancellation of all the coroutines within it. Listing 2 contains the `printHelloWorld()` suspending function that, similarly to the code in Listing 1, 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. At the same time, in case the launched coroutine gets canceled, the whole scope gets canceled, resulting in `printHelloWorld()` cancellation.

Listing 2 Creating a custom `CoroutineScope` in Kotlin.

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

Cancellation. Kotlin coroutines provide a built-in cancellation mechanism, which is especially useful for long-running applications. For example, a user might have closed the page that launched a coroutine, so its result is no longer needed, and the coroutine can be canceled. If a coroutine gets canceled while suspending, the respective `suspend` function throws `CancellationException` – the user code must not catch and always propagate it.

Listing 3 Coroutine communication via channel.

```
fun main() = runBlocking {
    val channel = Channel<Int>(capacity = 1)
    launch {
        for (x in 1..5) channel.send(x * x) // sends to channel
    }
    repeat(5) {
        println(channel.receive()) // receives from channel
    }
}
```

With structured concurrency, when a coroutine is canceled, all the coroutines operating within the scope get canceled, too, thus ensuring that all the related computations are safely canceled and do not leak.

Channels. When programming with coroutines, developers typically use channels for implicit synchronization and communication instead of manipulating shared memory. A channel is a blocking queue of bounded capacity with `receive` operation suspending if the channel is empty and `send` suspending when the channel is full. Listing 3 illustrates an inter-coroutine communication via channel. One coroutine sends square numbers to the channel, and the main coroutine reads these numbers from the same channel and prints them.

Coroutines and threads. Coroutines do not introduce a new concurrency model but enable cooperative concurrency and efficient and safe thread management, with the structured concurrency feature and explicit communication primitives in particular. However, one may still program with coroutines in a way similar to programming with threads, sharing a mutable state (e.g., a concurrent cache) and using the same synchronization primitives (e.g., mutex).

Discussion. Programming with coroutines varies significantly from traditional thread-based programming. These differences might give rise to unique bugs distinct from those typically encountered when manipulating threads and shared memory. This work sheds light on popular concurrency bug patterns discovered in real-world applications with Kotlin coroutines.

3 Bug Study Methodology

Our bug study targets seven open-source repositories listed in Table 1 together with the numbers of Kotlin code lines, commits, and GitHub stars. The repositories are selected based on three criteria: (i) the repository mainly contains Kotlin code, (ii) the project depends on the Kotlin coroutines library, and (iii) the repository has a high number of commits, indicating its active development and high number of stars indicating its popularity and the interest of the community.

As Kotlin is the primary language for Android development, we started with identifying top-starred Android projects on GitHub and eliminated the projects that have less than `1,000` commits and lack descriptive commit messages. As a result, we obtained two repositories: the `Shadowsocks` proxy client [31] and the `Tachiyomi` comic reader [35]. Next, we determined the Android repositories with the highest commit count. After elimination, the

Table 1 The selected GitHub repositories for the bug analysis.

Repository	Commits	Stars	Kotlin LOC	Total LOC
JetBrains/intellij-community	427.4 K	16.1 K	1 603.4 K	9 805.0 K
wordpress-mobile/WordPress-Android	83.2 K	2.9 K	243.5 K	4 561.7 K
woocommerce/woocommerce-android	46.8 K	259	250.5 K	367.6 K
mozilla-mobile/firefox-android	30.2 K	1.3 K	431.1 K	717.0 K
jshaw29/tachiyomi-backup [1]	6.2 K	25.8 K	66.1 K	114.1 K
ktorio/ktor	5.1 K	11.8 K	152.5 K	152.6 K
shadowsocks/shadowsocks-android	3.6 K	34.3 K	7.8 K	12.2 K

following three were selected: the `WordPress` website builder [3], the `WooCommerce` webshop manager [2], and the `Firefox` web browser [25]. Additionally, we selected the `Ktor` [18] framework for building asynchronous server-side and client-side applications and `IntelliJ` IDEA Community Edition [17], both developed by JetBrains – the main maintainers of Kotlin coroutines.

To collect concurrency bugs related to Kotlin coroutines, we analyzed all commits in the selected repositories. Specifically, (1) we filtered the commits based on whether the commit messages contained specific concurrency-related keywords, and (2) manually reviewed the sifted commits. Finally, (3) we categorized the identified bugs by the root causes of the errors.

Filtering the commits based on the commit messages. Following prior research on concurrency bug studies [38, 40, 23], we selected the commits that include at least one of the following keywords: `race`, `deadlock`, `synchronization`, `concurrency`, `lock`, `mutex`, `atomic`, `compete`, or `semaphore`. With our work focusing on Kotlin coroutines, we expanded the filter with the coroutine-related keywords: `runBlocking`, `Dispatcher`, `CoroutineScope`, `cancel`, and `CancellationException`. For the feasibility of the manual analysis, we limited the number of selected commits associated with each keyword to the most recent 30 commits.

Manual analysis of the selected commits. After filtering the commits in the repositories, we had `1353` commits to analyze. We manually reviewed them, examining their commit messages and code changes. We selected the commits that fixed a concurrency bug involving Kotlin coroutine primitives, with the change being comprehensible without in-depth knowledge of the codebase. Thus, we also filtered out the classic concurrent bugs unrelated to coroutines. We ended up with `55` bugs that involve Kotlin coroutine constructs.

Manual analysis and categorization of the bugs. Finally, we categorized the filtered `55` bugs by their root causes, analyzing the programming patterns that led to the errors. The source code links to the studied bugs are available in our GitHub repository [6].

Classic concurrency bugs. As the goal of this work is to analyze concurrency bugs that are related to Kotlin coroutines, the study does not cover traditional multithreaded concurrency bug patterns [23] such as data races, order violations, or atomicity violations. Rather, we focus on the bug patterns related to and introduced by using Kotlin coroutines constructs.

[1] The Tachiyomi repository has been taken down since January 2024: https://tachiyomi.org/news/2024-01-13-goodbye, so we reference a backup repository instead.

Table 2 Collected bugs and their categorization into bug patterns of nested `runBlocking`, scope passing, querying asynchronous objects, synchronizing with cancellation, and `CancellationException` bugs. All observed bugs that did not fall into one of these categories are listed under "Uncategorized".

Repository	Nested `runBlocking`	Scope Passing	Querying Async	Sync Cancel	Cancellation Exception	Uncategorized
`Intellij`	6	0	1	1	10	5
`Firefox`	2	2	2	2	0	4
`Tachiyomi`	2	0	0	0	3	1
`Ktor`	0	0	1	0	0	3
`Shadowsocks`	0	0	1	0	1	1
`Wordpress`	1	0	0	0	0	0
`Woocommerce`	0	2	0	1	0	3
Total	11	4	5	4	14	17

4 Categorization of Bugs

In this section, we analyze the bug patterns and root causes of the concurrency bugs and categorize them based on their characteristics. We discuss possible developer misunderstandings that lead to these mistakes, offer insight into possible remedial steps to rectify these errors and suggest methods for their automatic detection.

Table 2 lists the number of bugs in the repositories that fall into each of the bug categories. Based on our analysis of the Kotlin-specific concurrency primitives that are commonly involved in bugs and their root causes, we categorized the concurrency bugs in Kotlin programs into the classes of bugs due to (1) nested `runBlocking` calls (Section 4.1), (2) coroutine scope passing (Section 4.2), (3) querying asynchronous objects (Section 4.3), and (4) synchronizing with cancellation (Section 4.4). A special class of bugs occurs when a `CancellationException` is accidentally caught or incorrectly rethrown (Section 4.5). While it is not strictly a concurrency bug, it is caused by the Kotlin coroutines machinery, so we included it in our analysis.

Lastly, not all bugs found in the collection phase can be categorized into one of the categories. These bugs are displayed in Table 2 in the "Uncategorized" column. We omit these bugs in our analysis.

4.1 Calling `runBlocking` in a Coroutine

A common concurrency bug in Kotlin manifests when `runBlocking` coroutine builder is called from a coroutine and blocks the underlying coroutine dispatcher thread. Such a pattern can lead to a deadlock. We observed `11` bugs caused by this.

Root cause. The root cause of this bug is an improper use of the `runBlocking` coroutine builder designed to bridge non-coroutine and coroutine worlds and not expected to be called from another coroutine. Such an improper use can block the underlying scheduler thread, which might lead to a deadlock. Figure 1 and Listing 4 provide program examples with this bug pattern.

Figure 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 calls

`runBlocking` (line 8) which launches a coroutine *B* scheduled on `Dispatcher.Main` (line 9). The `runBlocking` builder blocks the UI thread and, due to structured concurrency, waits for coroutine *B* to finish. 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* completion, which results in a deadlock.

The same deadlock might also occur in a multi-thread scenario. Listing 4 provides such an example, using the `Default` multi-threaded dispatcher to schedule coroutines. Similarly to the code in Figure 1, the program gets into a deadlock when all threads are executing the coroutines launched in `main()`, schedule new coroutines in `nonSuspendingFunction()` calls. However, these new coroutines cannot be executed – all the scheduler threads are occupied with the coroutines launched in `main()` and wait for the completion of these coroutines launched in `nonSuspendingFunction()`.

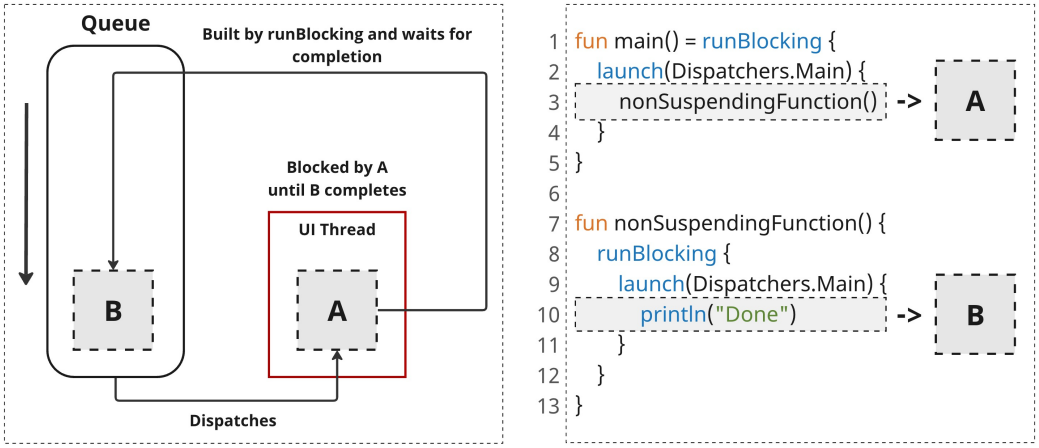

Figure 1 A program with a deadlock due to a `runBlocking` call from a coroutine on a single-threaded dispatcher.

Misconceptions. 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. The right course of action, however, is not always clear and requires careful consideration by the developer. 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 5. 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` also to be suspending. In other cases, one might prefer to acquire a coroutine scope created elsewhere. This scope, however, comes with its own set of challenges, which we explain in Section 4.2.

Listing 4 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 {
    launch (Dispatchers.Default) {
      println("Done")
    }
  }
}
```

Listing 5 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")
    }
  }
}
```

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.

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 [10].

Possible automated detection. A static analysis can inspect the program's call graph and detect situations when `runBlocking` is called from a `suspend` function. This analysis could also be implemented as an IDE inspection, and we have successfully added one into IntelliJ IDEA [7].

4.2 Scope Passing

Scope passing is a coding pattern in which a function launches a coroutine in a coroutine scope created outside the function. This situation can lead to an unexpected execution order of program statements, which may violate their intended order. While passing the coroutine scope is not incorrect and sometimes might be necessary, it makes it difficult to reason about the execution order.

Root cause. The root cause is the nondeterminism in the completion time of the coroutines launched on an external scope. Listing 6 provides a program example for the bug pattern. Consider the function `loadTopPerformers` (line 4), which is responsible for loading some data and storing it in the `lastUpdateTopPerformers` variable (line 2). The developer expects the data to be available in the `lastUpdateTopPerformers` variable once the `loadTopPerformers` function returns. However, this is not necessarily the case: the coroutine scope used to launch the coroutine on line 5 is not bound by the function `loadTopPerformers`; it is inherited from the `DashBoardViewModel` class.

As illustrated in Figure 2, the `this` object in the `loadTopPerformers` function refers to class `DashboardViewModel`, which extends `CoroutineScope`. This results in the coroutine unexpectedly outliving the function it was created in.

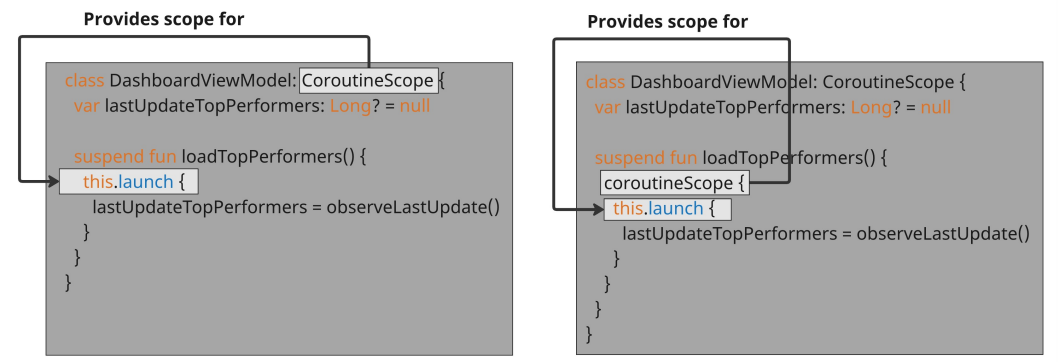

Figure 2 Visual clarification of coroutine scope origin and usage for Listing 6. Left shows the situation where the spawned coroutine can outlive the function it was created in. Right shows how this can be solved by creating a scope in the same function.

The problem can be fixed by ensuring that `loadTopPerformersStats` returns after the `lastUpdateTopPerformers` variable is set. As given in Listing 7, the example bug can be fixed by starting a `coroutineScope` call wrapping the function body (line 5). Then, the scope

Listing 6 An example scope passing bug, which is a simplified version of the bug in [32].

```
1 class DashboardViewModel: CoroutineScope {
2   var lastUpdateTopPerformers: Long? = null
3
4   suspend fun loadTopPerformers() {
5     launch {
6       lastUpdateTopPerformers = observeLastUpdate()
7     }
8   }
9 }
```

Listing 7 A potential solution to example scope passing bug in Listing 6.

```
class DashboardViewModel: CoroutineScope {
  var lastUpdateTopPerformers: Long? = null

  suspend fun loadTopPerformers() {
    coroutineScope {
      launch {
        lastUpdateTopPerformers = observeLastUpdate()
      }
    }
  }
}
```

Listing 8 An example for a scope passing bug, which is a simplified version of the bug in [28].

```
class ProductShippingClassViewModel(): CoroutineScope {
  private var loadJob: Job? = null

  fun load() {
    waitForCurrentLoadJob()
    loadJob = launch { /* loading logic */ }
  }

  fun waitForCurrentLoadJob() {
    launch { // launch since join cannot be called from normal fun
      loadJob?.join() // join suspends until load job is done
    }
  }
}
```

suspends the function `loadTopPerformersStats` until all its children are completed. Note that this solution would not have been an option if `loadTopPerformers` was a non-suspending function since the `coroutineScope` can only be called from a suspending function.

Another example of a coroutine unexpectedly outliving its calling function due to launching on an external scope is provided in Listing 8. The function `load` starts an expensive load operation by spawning a coroutine (line 6). To ensure this operation only runs once at a time, the developer keeps a reference to its `Job` (line 2). Next, he creates a function `waitForCurrentLoadJob` that performs a `join` operation. Then, at line 5, this waiting function is called before the expensive load operation is started. However, since `waitForCurrentLoadJob` is a normal function, it cannot call the suspending `join` method. In order to access this `join` operation, a coroutine is launched (line 10). However, this coroutine is launched on a scope that is defined outside the `waitForCurrentJob` function. The wait function will, therefore, return immediately, allowing a new load job to be started before the old one is completed.

Misconceptions. A developer might expect a function that launches a coroutine to wait for the coroutine to finish since that is often the case due to structured concurrency. This, however, only holds when the coroutine scope is created in that same function. In the example of Listing 6, the `loadTopPerformersStats` sits in between scope creation and

coroutine launch. Listing 8 shows that a normal function `waitForCurrentLoadJob` calls a `join` operation by spawning a coroutine. As discussed in Section 4.1, a problem arises when a suspend function needs to be called from a non-suspending function that runs in a coroutine. The developer needs to choose between calling `runBlocking`, passing scope, or refactoring all depending code to suspend functions.

Automated detection. Detecting this bug pattern is challenging since there can be valid reasons for passing the scope. However, the problem is that the developer might be unaware that, in some cases, a launched coroutine outlives the function that launched it. A simple analysis can be made that checks whether the scope used to launch a coroutine is created in the same function or not. A simple and non-intrusive visual indication might aid the developer in understanding the lifetime and scope of the launched coroutine.

4.3 Querying Asynchronous Objects

A race condition that commonly occurs with many languages is the result of querying the state of an object in shared memory and, based on that, deciding how to act on it. Listing 9 provides such an example where the developer checks if the channel is closed and sends a message if it is not closed.

Root cause. The root cause for this bug is simple and similar to race conditions in classical multithreaded programs. When an object is shared among threads, its state might change between checking its state and acting on it depending on the accesses to the object. In the example of Figure 9, the channel can be closed between the check `isClosedForSend` (line 1) and sending the message (line 2). A visual representation of this particular example is provided in Figure 3.

Misconceptions. The misconception is that the state of such an object will not change between the query and the action, disregarding the possibility of interleavings from other threads. The fix for the bug is to avoid sending if the channel is closed. As locking is undesired in Kotlin coroutines, the bug fix does not introduce explicit synchronization but wraps the `send` call in a `try/catch` block. A call on a closed channel should throw a `ClosedSendChannelException`, which allows the developer to handle it gracefully.

Automated detection. As the bug occurs in the case of a race condition on the channels, detecting this pattern can benefit from existing data race detectors.

4.4 Synchronizing with Cancellation

Synchronization with Cancellation is an attempt to ensure that a certain coroutine only runs once at a time by canceling the previous coroutine before the next is launched.

Root cause. The cancel method of a coroutine returns before the coroutine actually cancels or stops. In other words, cancel cannot be used to synchronize executions. In the example of Listing 11 there is a coroutine launched on line 7. The desired behavior is that this coroutine runs only once at a time. Otherwise, there exists a possible data race between the `read` (line 8) and `write` (line 11) actions. The `refresh` function (line 3) might be called again before the coroutine on line 7 is finished. To prevent a second coroutine from running in parallel, the developer keeps a reference to the `Job` of that coroutine and cancels it before a new

Listing 9 Race condition on channel status, taken from the bug fix in [24].

```
if (!channel.isClosedForSend) {
  channel.send(message)
}
```

Listing 10 Potential solution to Listing 9.

```
try {
  channel.send(message)
} catch (e: ClosedSendChannelException) {
  // handle closed channel if needed
}
```

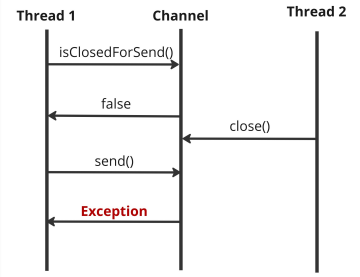

Figure 3 Visual representation of the execution order that could lead to a send operation over an unexpected closed channel.

Listing 11 Missed synchronization with `Job.cancel`, a simplified version of the bug in [5].

```
var pendingJob: Job? = null

suspend fun refresh() {
  pendingJob?.cancel() // does not wait for the coroutine to stop
  coroutineScope {

    pendingJob = launch(Dispatchers.IO) {
      val result = read(someVar)

      launch(Dispatchers.Main) {
        write(someVar, result)
      }
    }
  }
}
```

coroutine is started. This should ensure that the coroutine only runs once at a time. However, cancellation does not guarantee that execution will stop immediately, and the `cancel()` call does return immediately. Therefore, this coroutine can exist in parallel. Primitives that allow for synchronization are mutexes, joins, and channels. In this case, a potential solution is provided in Listing 12, a mutex that wraps the scope of the launched coroutine will make sure this coroutine can only be launched once the previous one is finished.

Misconceptions. A developer might be unaware that canceling of coroutines is cooperative, meaning that they can only cancel and stop when they reach a suspension point or manually check their cancellation status. Therefore, canceling the coroutine never guarantees that it actually stops. Additionally, the cancel method does not wait for the coroutine to be stopped.

4.5 Swallowing `CancellationException`

Incorrect handling of `CancellationExceptions` can introduce bugs manifesting in the executions with exceptions. While these bugs are not strictly concurrency bugs, they are specific to Kotlin coroutines. Therefore, we cover them in this section.

Listing 12 A potential solution to the synchronizing with cancel example in Listing 11.

```
val mutex = Mutex()
var pendingJob: Job? = null

suspend fun refresh() {
  pendingJob?.cancel() // optional
  mutex.withLock { // waits for coroutine to stop
    coroutineScope {

      pendingJob = launch(Dispatchers.IO) {
        val result = read(someVar)

        launch(Dispatchers.Main) {
          write(someVar, result)
        }
      }
    }
  }
}
```

Root cause. A `CancellationException` is thrown to signal that a coroutine is canceled. When this exception is caught, it interferes with the canceling mechanism of the coroutines. In Listing 13, a call to `suspendingFunction` is wrapped in a try/catch block to log any occurred errors. However, when the coroutine gets canceled while executing `suspendingFunction` a `CancellationException` is thrown which is then caught and logged. While logging a cancellation might be unfortunate, a bigger problem is that the cancellation is swallowed, since for it to work the exception needs to be propagated. A solution is given in Listing 14. The `CancellationException` is specifically caught and rethrown. We observed 14 bugs caused by accidentally swallowing `CancellationException`. This is also one of the most discussed issues in the Kotlin Coroutines issue tracker [29].

A bug that we did not observe but can occur involving `CancellationException` is when older Java frameworks throw these exceptions that could potentially run in a coroutine, making it stop silently when it shouldn't. In the example of Listing 15, a coroutine is launched on line 2. This coroutine calls a function `libraryCall` (line 3), which in this example is part of the same file but, in practice, can be any java library that throws a `CancellationException`. When this code example is executed, it will never reach the `println` statement on line 4. However, the program does finish gracefully (exit code 0). This is discussed in greater detail in the article "The Silent Killer That's Crashing Your Coroutines" [11].

Listing 13 Swallowed `CancellationException`.

```
suspend fun foo() {
  try {
    suspendingFunction()
  } catch (e: Exception) {
    Log.error(e)
  }
}
```

Listing 14 A potential solution to swallowed `CancellationException` in Listing 13.

```
suspend fun foo() {
  try {
    suspendingFunction()
  } catch (e: CancellationException) {
    throw e
  } catch (e: Exception) {
    Log.error(e)
  }
}
```

Listing 15 A library call throws a CancellationExcpetion and incorrectly cancels the coroutine.

```
fun main() = runBlocking {
  launch {
    libraryCall()
    println("Unreachable")
  }
}

fun libraryCall() { // Anything that throws CancellationException
  throw CancellationException() // Exception unrelated to coroutines
}
```

Misconceptions. The developer might forget or not be aware of the canceling mechanism of coroutines. Accidentally catching a `CancellationException` is the result of that.

Possible automated detection. One may implement a static analysis that inspects the call graph and searches for `try-catch` blocks that can call a `suspend` in the `try` block and catch `CancellationExcetion` (or a more generic one, e.g., `Exception` or `Throwable`) without propagating it by rethrowing outside the `catch` block. However, one should be careful when other constructs from the standard Java library that may throw `CancellationException` are used in the `try` block.

5 Threats to Validity

Potential threats to the validity include the representativeness of the studied concurrency bugs and our study methodology. Similar to other bug studies, our work analyzes a limited set of project repositories and a limited set of their commits.

We studied the Kotlin repositories based on our selection criteria for well-maintenance and popularity, which are based on the lines of code, number of commits, and stars. However, these criteria can potentially miss some Kotlin repositories with concurrency bugs. Similarly, we study a subset of commits in the selected repositories filtered by some keywords. We do not include some Kotlin framework keywords in our repository search, such as “channel” and “suspend”, which are frequently used during development with coroutines, appear in many of the commits, and introduce noise in the search results. Hence, the search results may potentially miss some bugs. Moreover, some bugs may not be explicitly discussed in the repositories’ commit messages or may not even have been diagnosed or fixed; therefore, they

may be missed by a repository search. Finally, our methodology involves a manual analysis of the commit source codes. While we aimed the analysis to be comprehensive, it may have missed some concurrency bugs studied or fixed in the commits.

While the study has limitations, some of which are inherent to real-world bug studies, we believe the studied bugs provide a useful sample of real-world concurrency bugs to shed light on misunderstandings and bug patterns in Kotlin programs with coroutines.

6 Key Takeaways and Discussion

While Kotlin coroutines provide a robust and straightforward mechanism for writing asynchronous programs, our study shows that developers can introduce specific concurrency bugs if they need to correctly use the asynchrony features.

Key takeaways. Our main observation is that developers may find it hard to identify *function coloring*, i.e., distinguishing which parts of their code run in asynchronous contexts, *bridging asynchronous and synchronous parts of their code*, and they may be unaware of or disregard the *semantics and mechanisms of some coroutine features* (e.g., the coroutine cancellation mechanism).

- While the Kotlin Coroutines framework helps developers identify asynchrony in their code by marking suspendable functions, programmers should be aware of regular functions that are called by asynchronous functions. Such functions, in turn, can run in an asynchronous context, and it can be hard to follow if they run in synchronous or asynchronous context, especially in large programs with deep execution call stacks.
- Developers should be careful when bridging the synchronous and asynchronous parts of their programs. Our analysis shows that when they need to call a suspend function from a synchronous context, they may tend to use *runBlocking* calls as a quick solution. This mistake is understandable since a suspend function calling a synchronous one is unaware that it might reach a *runBlocking*, while the synchronous function is unaware it is called from a coroutine. However, this unawareness can lead to dangerous *runBlocking* calls, which can result in serious concurrency errors such as deadlocks.
- When developers are aware of the asynchronous execution context but still need to call a suspend function from a normal one, they might choose to pass a coroutine scope. This solution, however, can introduce unexpected executions: the suspend function can outlive the synchronous function that called it. Incorrect reasoning about the function scopes and making incorrect assumptions about the completion of functions can result in unexpected execution orders of the program's statements.
- Similarly, the developers should be aware of the asynchronous objects and use the correct library structures to access or run operations on them. Incorrect assumptions (e.g., on the synchronization with channels) and ignorance of possible interference from other threads result in concurrency errors.
- Finally, developers should be aware of the semantics and guarantees of the programming abstractions and features they use. For example, we observed that there is common confusion about the canceling behavior of coroutines. Canceling a coroutine is cooperative and, therefore, does not guarantee it will be canceled. Similarly, the developers unaware of the cancellation exception handling mechanism of coroutines can introduce serious problems as incorrectly catching for these exceptions potentially silences critical exceptions in their programs.

Discussion. Besides increasing developers' awareness of common misconceptions, understanding common bug patterns can lead to the development of suitable program analysis tools for Kotlin programs. We communicated our findings to the Kotlin and IntelliJ teams at JetBrains. Our findings have led to the development of an inspection tool for detecting problematic `runBlocking` calls, which is currently part of the IntelliJ source code [7].

7 Related Work

7.1 Studies of Real-world Concurrency Bugs

Similar studies have been conducted that collect and categorize concurrency bugs in different programming languages and frameworks. Earlier work analyses of C/C++ concurrency bugs from server and client applications [23, 15], and report that most non-deadlock concurrency bugs are caused by atomicity and order violations. Focusing on misuse of asynchronous constructs in C# programs, the work in [26] identifies problems due to misuse or unnecessary use of asynchronous methods, invocation of long-running tasks in asynchronous methods and some anti-patterns specific to C#'s async and await model. Another study on real-world concurrency bugs [40] targets asynchronous and event-driven Node.js programs. The work identifies the concurrency bug patterns in Node.js programs as atomicity violations, order violations, and starvation in the execution of event handlers.

For Golang, the studies in [8] and [38] collect and analyze real-world concurrency bugs. The bug study in [8] focuses on data races in Go programs, and [38] focuses on the inter-thread communication mechanisms, i.e., whether message passing or shared memory concurrency is less error-prone. The findings of these works successfully led to the research and development of multiple concurrency bug analysis techniques for Go [39, 21, 13, 20, 36].

Targeting the actor model of concurrency, bug studies in [16, 22, 4] focus on actor programs, where a program consists of a set of actors that concurrently operate on their local states and communicate by exchanging asynchronous messages.The work in [16] categorizes the bugs in actor programs into communication bugs (problems in handling messages or due to delivery orderings of messages) and coordination bugs (e.g., ungraceful shutdown or recovery of actors). Following the categorization of classical shared-memory concurrency bugs, the work in [22] categorizes the actor program bugs into lack of progress and message protocol violations and defines specific subclasses of each category for actor programs. The study of actor concurrency bugs in [4] focuses on real-world Akka actor programs and analyzes their symptoms, root causes, and API usage. The bug characteristics in actor programs differ from classical shared memory programs and coroutine programs we study in this work since actors provide a high-level concurrency model without a shared state, and the concurrency nondeterminism is in the order of asynchronous events.

Different from these works, which identify the classical bug patterns of atomicity and order violations in the shared memory accesses [23, 15, 1], atomicity and order violations in the handling of events [40], message protocol violations in actor programs [16, 22, 4], or misuses and bugs in asynchronous programming in C# [26] or Go [8, 38], in this work, we focus on concurrency bugs in Kotlin programs and identify new bug patterns specifically related to Kotlin coroutines.

7.2 Analysis of Kotlin Programs

Kotlin is a relatively new programming language and only some recent research addresses the analysis of Kotlin programs. A related work targeting channel-based concurrent programs [30], which tests channel-based systems through fuzzing, has found and led to the resolution of a

bug in the Kotlin coroutine implementation. Another related work, Lincheck [19], provides a testing framework for concurrent algorithms that run on the JVM, which has been adopted in Java and Kotlin communities.

The bug patterns we discovered in this work can be useful for designing and developing program concurrency analysis tools for Kotlin programs.

8 Conclusion

This paper introduces the first real-world concurrency bug study for Kotlin coroutines, shedding light on the typical patterns of concurrency bugs. Having examined 55 concurrency bugs selected from 7 popular open-source repositories that use Kotlin coroutines, we identified common bug patterns related to Kotlin coroutine semantics. We distilled suggestions for Kotlin developers to avoid these programming errors and discussed possible techniques to detect such issues automatically. We reported our findings to the Kotlin and IntelliJ teams at JetBrains, and we believe our findings will help future research and development of concurrency analysis tools for Kotlin.

References

1 Sara Abbaspour Asadollah, Daniel Sundmark, Sigrid Eldh, Hans Hansson, and Wasif Afzal. 10 years of research on debugging concurrent and multicore software: a systematic mapping study. *Softw. Qual. J.*, 25(1):49–82, 2017. doi:10.1007/S11219-015-9301-7.

2 Automattic. Woocommerce Android app, 2023. URL: https://github.com/shadowsocks/shadowsocks-android.

3 Automattic. Wordpress for android, 2023. URL: https://github.com/wordpress-mobile/WordPress-Android.

4 Mehdi Bagherzadeh, Nicholas Fireman, Anas Shawesh, and Raffi Khatchadourian. Actor concurrency bugs: a comprehensive study on symptoms, root causes, API usages, and differences. *Proc. ACM Program. Lang.*, 4(OOPSLA):214:1–214:32, 2020. doi:10.1145/3428282.

5 Jeff Boek. Bugfix commit, firefox for android, November 2018. URL: https://github.com/mozilla-mobile/firefox-android/commit/d18bfe9f1bb5f0ed0f85e5fa36cddfaee84b5d47.

6 Bob Brockbernd. Found bugs per bug type, April 2024. URL: https://github.com/bbrockbernd/kotlin-coroutine-bugs.

7 Bob Brockbernd. Runblocking inspection implementation, June 2024. URL: https://github.com/JetBrains/intellij-community/commit/ea8296d53925ec87ddbee66f37412793d3fbdb14.

8 Milind Chabbi and Murali Krishna Ramanathan. A study of real-world data races in Golang. In Ranjit Jhala and Isil Dillig, editors, *PLDI '22: 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation, San Diego, CA, USA, June 13 - 17, 2022*, pages 474–489. ACM, 2022. doi:10.1145/3519939.3523720.

9 Yi-An Chen and Yi-Ping You. Structured concurrency: A review. In *Workshop Proceedings of the 51st International Conference on Parallel Processing, ICPP Workshops 2022, Bordeaux, France, 29 August 2022 - 1 September 2022*, pages 16:1–16:8. ACM, 2022. doi:10.1145/3547276.3548519.

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

11 Sam Cooper. The silent killer that's crashing your coroutines, February 2023. URL: https://medium.com/better-programming/the-silent-killer-thats-crashing-your-coroutines-9171d1e8f79b.

12 Ana Lúcia de Moura and Roberto Ierusalimschy. Revisiting coroutines. *ACM Trans. Program. Lang. Syst.*, 31(2):6:1–6:31, 2009. doi:10.1145/1462166.1462167.

13 Nicolas Dilley and Julien Lange. Automated verification of go programs via bounded model checking. In *36th IEEE/ACM International Conference on Automated Software Engineering, ASE 2021, Melbourne, Australia, November 15-19, 2021*, pages 1016–1027. IEEE, 2021. doi:10.1109/ASE51524.2021.9678571.

14 Roman Elizarov, Mikhail A. Belyaev, Marat Akhin, and Ilmir Usmanov. Kotlin coroutines: design and implementation. In Wolfgang De Meuter and Elisa L. A. Baniassad, editors, *Onward! 2021: Proceedings of the 2021 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, Virtual Event / Chicago, IL, USA, October 20-22, 2021*, pages 68–84. ACM, 2021. doi:10.1145/3486607.3486751.

15 Rui Gu, Guoliang Jin, Linhai Song, Linjie Zhu, and Shan Lu. What change history tells us about thread synchronization. In Elisabetta Di Nitto, Mark Harman, and Patrick Heymans, editors, *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015*, pages 426–438. ACM, 2015. doi:10.1145/2786805.2786815.

16 Brandon Hedden and Xinghui Zhao. A comprehensive study on bugs in actor systems. In *Proceedings of the 47th International Conference on Parallel Processing, ICPP 2018, Eugene, OR, USA, August 13-16, 2018*, pages 56:1–56:9. ACM, 2018. doi:10.1145/3225058.3225139.

17 JetBrains. Intellij idea community edition, 2023. URL: https://github.com/JetBrains/intellij-community.

18 JetBrains. Ktor, 2023. URL: https://github.com/ktorio/ktor.

19 Nikita Koval, Alexander Fedorov, Maria Sokolova, Dmitry Tsitelov, and Dan Alistarh. Lincheck: A practical framework for testing concurrent data structures on JVM. In Constantin Enea and Akash Lal, editors, *Computer Aided Verification - 35th International Conference, CAV 2023, Paris, France, July 17-22, 2023, Proceedings, Part I*, volume 13964 of *Lecture Notes in Computer Science*, pages 156–169. Springer, 2023. doi:10.1007/978-3-031-37706-8_8.

20 Ziheng Liu, Shihao Xia, Yu Liang, Linhai Song, and Hong Hu. Who goes first? detecting go concurrency bugs via message reordering. In Babak Falsafi, Michael Ferdman, Shan Lu, and Thomas F. Wenisch, editors, *ASPLOS '22: 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, 28 February 2022 - 4 March 2022*, pages 888–902. ACM, 2022. doi:10.1145/3503222.3507753.

21 Ziheng Liu, Shuofei Zhu, Boqin Qin, Hao Chen, and Linhai Song. Automatically detecting and fixing concurrency bugs in go software systems. In Tim Sherwood, Emery D. Berger, and Christos Kozyrakis, editors, *ASPLOS '21: 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Virtual Event, USA, April 19-23, 2021*, pages 616–629. ACM, 2021. doi:10.1145/3445814.3446756.

22 Carmen Torres Lopez, Stefan Marr, Elisa Gonzalez Boix, and Hanspeter Mössenböck. A study of concurrency bugs and advanced development support for actor-based programs. In Alessandro Ricci and Philipp Haller, editors, *Programming with Actors - State-of-the-Art and Research Perspectives*, volume 10789 of *Lecture Notes in Computer Science*, pages 155–185. Springer, 2018. doi:10.1007/978-3-030-00302-9_6.

23 Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In Susan J. Eggers and James R. Larus, editors, *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2008, Seattle, WA, USA, March 1-5, 2008*, pages 329–339. ACM, 2008. doi:10.1145/1346281.1346323.

24 Sergey Mashkov. Bugfix commit, ktor, October 2020. URL: https://github.com/ktorio/ktor/commit/7dfa6d9f1650430738e76cba165eb3529687be3c.

25 Mozilla. Firefox for android, 2023. URL: https://github.com/mozilla-mobile/firefox-android.

26 Semih Okur, David L. Hartveld, Danny Dig, and Arie van Deursen. A study and toolkit for asynchronous programming in c#. In Pankaj Jalote, Lionel C. Briand, and André van der Hoek, editors, *36th International Conference on Software Engineering, ICSE '14, Hyderabad, India - May 31 - June 07, 2014*, pages 1117–1127. ACM, 2014. doi:10.1145/2568225.2568309.

27 Daniil Ovchinnikov. Runblocking should let go of CPU token before parking the thread, December 2023. URL: https://github.com/Kotlin/kotlinx.coroutines/issues/3983.

28 Ondrej Ruttkay. Bugfix commit, woocommerce android app, January 2021. URL: https://github.com/woocommerce/woocommerce-android/commit/6156420791b1e78067cd5dacbe5a15d0cc24979d.

29 Anton Spaans. Provide a runcatching that does not handle a cancellationexception but rethrows it instead, February 2020. URL: https://github.com/Kotlin/kotlinx.coroutines/issues/1814.

30 Quentin Stiévenart and Magnus Madsen. Fuzzing channel-based concurrency runtimes using types and effects. *Proc. ACM Program. Lang.*, 4(OOPSLA):186:1–186:27, 2020. doi:10.1145/3428254.

31 Mygod Studio. Shadowsocks for android, 2023. URL: https://github.com/shadowsocks/shadowsocks-android.

32 Petr Surkov. Bugfix commit, woocommerce android app, March 2024. URL: https://github.com/woocommerce/woocommerce-android/commit/5dbb3b1834f67f097be7db96cab04c3ca99d7294.

33 Don Syme, Tomas Petricek, and Dmitry Lomov. The f# asynchronous programming model. In Ricardo Rocha and John Launchbury, editors, *Practical Aspects of Declarative Languages - 13th International Symposium, PADL 2011, Austin, TX, USA, January 24-25, 2011. Proceedings*, volume 6539 of *Lecture Notes in Computer Science*, pages 175–189. Springer, 2011. doi:10.1007/978-3-642-18378-2_15.

34 Martin Sústrik. Structured concurrency, February 2016. URL: https://250bpm.com/blog:71/.

35 Tachiyomiorg. Tachiyomi, 2023. URL: https://github.com/tachiyomiorg/tachiyomi.

36 Saeed Taheri and Ganesh Gopalakrishnan. Automated dynamic concurrency analysis for go, 2021. arXiv:2105.11064.

37 Andrew Troelsen and Andy Olsen. *Pro C# 5.0 and the. NET 4.5 Framework*, volume 6. Springer, 2012.

38 Tengfei Tu, Xiaoyu Liu, Linhai Song, and Yiying Zhang. Understanding real-world concurrency bugs in go. In Iris Bahar, Maurice Herlihy, Emmett Witchel, and Alvin R. Lebeck, editors, *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2019, Providence, RI, USA, April 13-17, 2019*, pages 865–878. ACM, 2019. doi:10.1145/3297858.3304069.

39 Oskar Haarklou Veileborg, Georgian-Vlad Saioc, and Anders Møller. Detecting blocking errors in Go programs using localized abstract interpretation. In *37th IEEE/ACM International Conference on Automated Software Engineering, ASE 2022, Rochester, MI, USA, October 10-14, 2022*, pages 32:1–32:12. ACM, 2022. doi:10.1145/3551349.3561154.

40 Jie Wang, Wensheng Dou, Yu Gao, Chushu Gao, Feng Qin, Kang Yin, and Jun Wei. A comprehensive study on real world concurrency bugs in node.js. In Grigore Rosu, Massimiliano Di Penta, and Tien N. Nguyen, editors, *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, Urbana, IL, USA, October 30 - November 03, 2017*, pages 520–531. IEEE Computer Society, 2017. doi:10.1109/ASE.2017.8115663.