



Runtime Verification of Kotlin Coroutines

Denis Furian, Shaun Azzopardi, Yliès Falcone, Gerardo Schneider

► To cite this version:

Denis Furian, Shaun Azzopardi, Yliès Falcone, Gerardo Schneider. Runtime Verification of Kotlin Coroutines. RV 2022 - 22nd International Conference on Runtime Verification, Sep 2022, Tbilisi, Georgia. pp.1-19. hal-03911794

HAL Id: hal-03911794

<https://inria.hal.science/hal-03911794>

Submitted on 23 Dec 2022

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Runtime Verification of Kotlin Coroutines

Denis Furian¹, Shaun Azzopardi^{2*}, Yliès Falcone^{3**}, and Gerardo Schneider²

¹ Opera Software AB, Gothenburg, Sweden

² University of Gothenburg, Sweden

³ Univ. Grenoble Alpes, CNRS, Grenoble INP, Inria, LIG, 38000 Grenoble, France

Abstract. Kotlin was introduced to Android as the recommended language for development. One of the unique functionalities of Kotlin is that of coroutines, which are lightweight tasks that can run concurrently inside threads. Programming using coroutines is difficult, among other things, because they can move between threads and behave unexpectedly. We introduce runtime verification in Kotlin. We provide a language to write properties and produce runtime monitors tailored to verify Kotlin coroutines. We identify, formalise and runtime verify seven properties about common runtime errors that are not easily identifiable by static analysis. To demonstrate the acceptability of the technique in real applications, we apply our framework to an in-house Android app and micro-benchmarks and measure the execution time and memory overheads.

1 Introduction

Coroutines were introduced at the beginning of the 1960s by Joel Erdwinn and Melvin E. Conway to achieve separability in the context of compiler optimisation (for COBOL) [6]. Coroutines have been the object of discussion and analysis after their introduction (see, for instance, Clarke’s paper about the correctness of coroutines [4] and the more recent survey [20]⁴), but they only became “fashionable” again with their introduction in the Kotlin programming language.

Though coroutines have been defined in slightly different ways, in a nutshell, they are programming language control structures with the following characterising features [19]: i) the values of the coroutine local data persist between successive calls; ii) the execution of a coroutine is suspended as control leaves it, only to carry on where it left off when control re-enters the coroutine at some later stage. Besides, de Moura and Ierusalimschy [20] write: “We can identify three main issues that distinguish different kinds of coroutine facilities: i) the

* Supported by ERC Consolidator grant D-SynMA (No. 772459).

** Supported by the Région Auvergne-Rhône-Alpes within the “Pack Ambition Recherche” programme, the H2020- ECSEL-2018-IA call – Grant Agreement number 826276 (CPS4EU), the French ANR project ANR-20-CE39-0009 (SEVERITAS), and LabEx PERSYVAL-Lab (ANR-11-LABX-0025-01).

⁴ See also references therein for different uses of coroutines (e.g., for simulation, in artificial intelligence, text processing, etc.).

control-transfer mechanism, which can provide *symmetric* or *asymmetric* coroutines; ii) whether coroutines are provided in the language as *first-class* objects, which can be freely manipulated by the programmer, or as constrained constructs; iii) whether a coroutine is implemented as a *stackful* construct, that is, whether it can suspend its execution from within nested calls.”

We consider the use of coroutines in the Kotlin programming language. In theory, the use of coroutines seems straightforward and appealing. In practice, though, its use is far from being unproblematic: as with many other concurrency constructs, it is difficult to determine how they would behave at execution time. Why is it difficult to program with coroutines? Many things can go wrong; we just mention a couple of issues to make our point. An example of undesirable behaviour is a coroutine holding references to an object that has since been destroyed: this often happens in Android, where most components have their own lifecycle. Such behaviour results in memory leaks during execution. Another problem is when a coroutine executes using an undesirable dispatcher and an I/O operation is carried out inside a thread dedicated to the UI: since operations like this can cause the UI to be slowed down or become unresponsive, the Android OS attempts to crash the whole application by throwing a `NetworkOnMainThreadException` at runtime.

These examples illustrate the difficulty in ensuring, statically, that coroutines will behave as expected. Consequently, the best we can do is to identify specific harmful situations during execution and try to prevent the error from happening, or at least to identify and report on the error. Runtime verification (RV) can help programmers, before deployment to debug their software (as for testing) or after deployment to identify (and prevent) errors and eventually correct them.

In this paper, we are concerned with the runtime verification of coroutines in general and, particularly, in Kotlin. We design properties concerning what could go wrong when programming using coroutines and use runtime verification techniques to monitor them. Our ultimate goal is to develop a dedicated tool that allows users to write properties in a declarative language (from which a monitor could automatically be extracted) tailored to monitor programs using coroutines during execution. As a first step, we start by targeting Kotlin developers who might want to use runtime verification during testing as a debugging tool. For that, we implemented in Kotlin monitors for properties that capture many cases not easily, if at all, detectable by manual code inspection or static analysis.

More concretely, our contributions are as follows:

1. We identify seven properties concerning coroutines which, if not satisfied, may cause undesirable behaviour (Section 3);
2. We propose a declarative property language for coroutines (Section 4);
3. We present an RV algorithm for the above language (Section 5);
4. We implement the properties discussed in Section 3, as coroutines and discuss their effectiveness (Section 6).

Related work is described in Section 7 and concluding remarks are given in Section 8. In the next section, we summarise the main differences between Kotlin and Java and explain some coroutines’ features.

2 Background: Coroutines in Kotlin

Kotlin is a programming language whose main features are its interoperability with Java and the native support of asynchronous programming via coroutines⁵. While it was developed in 2011 by JetBrains, it has been officially supported for Android development alongside Java since October 2017 until it became the preferred language by Google in 2019. Kotlin can be compiled to JVM bytecode, JavaScript or native code via LLVM. Compiling to JVM bytecode makes Kotlin easy to interoperate with Java and *vice versa*, despite the few but noticeable differences between the two languages. These include the handling of exceptions (always unhandled in Kotlin), and support for non-nullable types and coroutines.

In Kotlin, coroutines employ structured concurrency, which means that entry and exit points must be made clear and all tasks are either completed or cancelled before the end of the execution [22].

A Kotlin coroutine runs inside a thread, and a thread can run several coroutines: they follow a pattern of *suspend/resume* where they can be suspended at any time, their state is saved and then restored whenever they resume, as mentioned in the previous section. A coroutine can also suspend on one thread and resume on another after transferring its state. This can happen, for example, when a coroutine runs on a multithreaded dispatcher [16] such as `Dispatchers.Default`, which uses a number of threads between two and the number of CPU cores.

In order to handle mutual exclusion, Kotlin provides a coroutine-specific class called `Mutex`. This contains a suspend function `lock()` that allows the caller to gain exclusive access to a portion of code. The complementary function `unlock()` releases the lock and must be called before any other coroutine can gain access to the critical section. In other words, a coroutine that invokes `Mutex.lock()` and then crashes without invoking `Mutex.unlock()` will consistently starve any other jobs waiting on that lock. Since a coroutine may terminate at any given time, the `Mutex` class provides a functional block `withLock` that automatically requests the lock. It then releases it no matter what before termination.

The Kotlin standard library also provides a more conventional tool for mutual exclusion in the form of the `Lock` class. This class is, however, intended for use with threads and attempting to gain a lock inside a coroutine will make the whole thread dormant and disabled for scheduling.

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

- A standard job that executes a block of code without returning any value is created with the method `launch`.
- A job that is expected to return a value is created with the method `async`: this returns an instance of the `Deferred` class. Using the `await` method on the deferred object will suspend the current coroutine until a value is

⁵ Kotlin's documentation can be found in [15]. Here we give a brief background of Kotlin features pertinent to our work.

returned. If the coroutine is cancelled before it can return a value, however, awaiting the deferred object will throw an exception.

Creation of a coroutine The methods discussed, `launch` and `async`, are called “coroutine builders” since they prompt the creation of a new coroutine to run the asynchronous task. Both must be invoked inside a *coroutine scope*, which delimits the coroutine lifetime following the principle of structured concurrency.

Coroutines are suspendable. The `await` method (from the `Deferred` class) is one example of a suspending function. Depending on the coroutine scope, different behaviour is exhibited during suspension. The coroutine may be blocking (through the `runBlocking` coroutine builder), in which case the coroutine calling a suspending function will block the whole thread until all its tasks are completed. It can also be just suspending, were calling a suspending function releases the thread to do other work.

The scope contains the *coroutine context*: a composite object containing the job to be executed as well as its dispatcher, which determines what thread (or threads) the coroutine uses for its execution.

A *dispatcher*, either custom or provided by a library, can be explicitly assigned to a coroutine when used as argument for the builder function: for example, `launch { foo() }` will use the same context as the parent task while `launch(Dispatchers.IO) { foo() }` will use the base I/O dispatcher. The library `kotlinx.coroutines` provides three base dispatchers as well as methods for generating thread pools:

- `Dispatchers.Default` uses a common pool of shared background threads and is used normally by all builders if no other dispatcher is specified;
- `Dispatchers.IO` is designed for blocking operations that are I/O-intensive, like file up- or downloads;
- `Dispatchers.Unconfined` starts coroutine execution in the current thread until the first suspension and then allows it to resume in whatever thread the corresponding suspending function uses: using this dispatcher takes control away from the programmer and leads to potentially unwanted results, so it is discouraged by JetBrains.

The *job* inside the coroutine context is used for tracking the coroutine’s parent and children. This is needed when enforcing structured concurrency in order to ensure that the children do not outlive the parents. It is also possible to spawn an independent coroutine by using a new `Job` instance as an argument to either builder function.

The job and dispatcher can be combined inside a builder arguments list, while, it is possible to retrieve their values as entries of the `coroutineContext` instance, as follows:

```
launch(Dispatchers.IO + Job()) {
    println( "Running job ${coroutineContext[Job]} on " +
            "dispatcher ${coroutineContext[CoroutineDispatcher]}" ) }
```

Termination of a Coroutine The execution of a coroutine can be cancelled at any time by invoking the `CoroutineScope`’s `cancel` method: this causes a

`CancellationException` to be sent to that coroutine. This is not treated as a “real” exception as much as a prompt to terminate. Any coroutine receiving this exception will quickly execute the following steps: i) execute any code it might find inside a `finally` block; ii) recursively cancel all of its children (by forwarding the same `CancellationException` to them; and then iii) terminate.

If the coroutine was created from `async`, executing the `await` call will throw the `CancellationException`; otherwise the coroutine will simply terminate.

Any other kind of exception will result in the coroutine’s termination. Since coroutines follow the paradigm of structured concurrency, cancellation, in this case, is propagated both downstream and upstream. This means that both the children *and the parent* of the failed coroutine will terminate. There is one way to prevent the cancellation from propagating upstream: the failing coroutine must be spawned by a `SupervisorJob`. In this case, the parent will not be affected by the failure and will simply receive the exception that caused the failure.

Note that catching an exception in a `try-catch` block will not prevent termination. It may, however, allow for a quick handling such as, for example, logging. Any code inside a `finally` block will be executed.

3 Informal Description of Properties

Because of the various features and perks of both the Kotlin language and Android development, we have identified seven properties that should be monitored, which we index and describe in this section.

Of these seven, properties 1 and 7 are about coroutines outliving their caller and the subsequent risk of leaking memory mentioned earlier; properties 5 and 6 address the possibility that a coroutine may run inside an undesirable dispatcher, with serious risks of fatal crashes as a consequence; properties 2, 3 and 4 concern how Kotlin handles successful and failed tasks, as well as how exception traces may be lost when a crash takes place in an asynchronous computation. When this work was performed properties 2 and 3 were concerning, but Kotlin now handles these by design (from v1.4). We consider them for completeness.

Other undesirable scenarios, like the one presented in the first property, can now be avoided by making use of first-party libraries which provide, for example, coroutine scopes that are lifecycle-aware.

We present a summary of all relevant properties at the end of this section.

3.1 Property 1: `DestroyedWithOwner`

Coroutines execute a given block of code which may or may not contain references to an Android lifecycle component. We do not go into detail of what these are, but these components are destroyed and recreated arbitrarily and we do not want them to persist inside an asynchronous task as that would leak memory.

RV is required since the destruction of a lifecycle component does not happen regularly and can be triggered by events external to the app (like the device battery running low) or to the device itself (like the user rotating the screen).

Static analysis could be used before the execution to ensure, for example, that a lifecycle-aware coroutine scope is being used: a scope like this can be tailor-made or imported from the official `ktx` library, but there is no guarantee that a programmer will be doing either. This makes RV more desirable as it does not necessarily impose a restriction on the programmer’s choice of libraries.

3.2 Property 2: NormalAsync

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

- **launched** tasks will yield `Unit`, the Kotlin equivalent of Java’s `void`;
- **async** tasks will yield a `Deferred<T>` value, i.e. a “future” result that eventually evaluates to a value of type `T`.

Either of these scenarios is the “optimal” behaviour for its kind of task. This property is only broken when a coroutine throws an exception as a “successful” scenario, e.g., a **launched** task that executes on an infinite loop, throwing a `RuntimeException` to force termination, as for the following code snippet:

```
suspend fun foo() { if (goodScenario()) doThings()
                    else throw RuntimeException() }
```

To help ensure JetBrains’ recommendation that exceptions should not be used as return values, a monitor can be used to identify and notify when exceptions are yielded by coroutines (which may not be as easily determinable by static analysis as in the previous example).

3.3 Property 3: ExceptionalAsync

An exception thrown inside an **async** coroutine will flag the current context for termination; the only outliers are cancellation exceptions, which are seen as “normal” termination directives rather than crashes.

Thrown exceptions should ideally be stored in the current context or in another *throwable* saved in the current context, to avoid losing information about the crash. In the newer versions of Kotlin (from v1.4), exceptions thrown after a crash are stored inside the field `Exception.suppressed`. The exception is then thrown upon the invocation of `await`, ensuring that all crash data is available for the programmer to handle, as exemplified in the following code:

```
fun main() = runBlocking {
    val deferred = GlobalScope.async { throwOneAndSuppress(10000) }
    try { deferred.await() }
    catch (e: Exception) {
        println("Suppressed ${e.suppressed.size} exceptions") }}

suspend fun throwOneAndSuppress(amount: Int) = coroutineScope {
    repeat(amount) { launch {
        try { delay(Long.MAX_VALUE) }
```

```

        finally { throw ArithmeticException() }}}
launch { delay(100L) // This will be thrown first.
        throw IOException() }
delay(Long.MAX_VALUE) }

```

For versions of Kotlin that do not support this, we considered monitoring as a way to collect these exceptions.

3.4 Property 4: NeedHandler

Any exception thrown inside `launch` coroutines should be rethrown between parent tasks all the way until the `CoroutineExceptionHandler` at the top level handles the failure. As mentioned for the previous property, exceptions should be stored to preserve information about the crash. In the case of `launch` jobs, this means that the context needs an exception handler that carries exception data rather than simply crashing. The exception thrown references other failures that occurred after it inside the field `Exception.suppressed` and all the information can then be accessed from the handler. The following snippet represents a compliant scenario where the function `throwOneAndSuppress` from the previous example is started with a `launch` rather than `async` coroutine builder:

```

fun main() = runBlocking {
    val handler = CoroutineExceptionHandler { _, exception ->
        println("Suppressed ${exception.suppressed.size} exceptions") }
    val job = GlobalScope.launch(handler) {
        throwOneAndSuppress(10000) }
    job.join() }

```

Here, instrumentation can be used to enforce automatically the propagation of exceptions happening inside a `launch` coroutine upwards.

3.5 Property 5: NoBlockUI

Android apps should leave the UI thread as lightweight as possible and avoid blocking it with heavy and/or slow computations. Some scenarios are downright forbidden, like when an I/O operation is executed on the UI thread. In these situations the Android runtime will launch a `NetworkOnMainThreadException`.

As an example, let us consider an app that reads a JSON stream from an endpoint and uses it to update some components on the screen. The reading and the UI update are carried out inside suspend functions that execute in whatever context they are launched in. The view model invokes them in the background whenever the activity is resumed, and then the activity itself invokes them every time the user presses a refresh button. The expectation is that the app loads and displays the data when the activity is started and then repeats the operation every time the refresh button is pressed or every time the app returns to the foreground. The actual scenario is that the app will crash as soon as the activity

is started, as the view model invokes the read coroutine (on the UI thread), triggering a `NetworkOnMainThreadException`.

Kotlin does not issue any warnings about this possibility. Static analysis could be developed to ensure the correct dispatcher is used for each task, assuming a precise-enough analysis. RV instead provides us with a lightweight method, and allows for possible enforcement.

3.6 Property 6: UpdateUI

In a similar situation as the one above, when a background thread tries to access UI elements, the Android runtime throws a `CalledFromWrongThreadException`. This can be seen in the code example below, where the methods `okState` and `errorState` update the screen with the outcome of a network operation, and thus use within a view model (a UI element) will cause the system to crash.

```
private suspend fun getJson(uri: String) = coroutineScope {
    try { URL(uri).readText().let { data: String ->
        val parsedData = parseJsonResponse(data)
        if (parsedData is Failure) { errorState("Error: $uri") }
        else { okState(parsedData.getOrNull()) }}}
    catch (e: Exception) { errorState(e.toString()) }}
```

3.7 Property 7: ResumeIfNeeded

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

In cases where computation takes a large amount of time to complete, however, there might not be a chance for the dispatcher to check for cancellation. It is good practice to check the flag `isActive`, which returns `false` whenever the current coroutine is not supposed to execute anymore; there is also an `ensureActive` method that throws a `CancellationException` unless the `isActive` flag is `true`. These checks are executed at runtime so it would be appropriate to ensure at runtime that a task is only completed if necessary.

Let us consider an asynchronous task that is set to download a large file and store it in the user's smartphone. This task is launched inside a coroutine that starts the download process. The Android activity is suddenly terminated: we *ideally* want the download to be interrupted. The scope used to launch the coroutine will determine whether the task is allowed to continue or not: if the coroutine context is tied to the activity, it will notify the download task (along with any other tasks) of the activity having terminated using a `CancellationException`.

3.8 Properties summarised

- *DestroyedWithOwner*: any coroutine launched from a component with a life-cycle should be destroyed together with it to avoid memory leaks;
- *NormalAsync*: coroutines should not throw exceptions as special return values but, rather, reserve their use for failures;
- *ExceptionalAsync*: multiple exceptions occurring inside an `async` coroutine should all be able to be tracked and retrieved, rather than only one of them;
- *NeedHandler*: any `launch` coroutine should have a handler that keeps track of exceptions;
- *NoBlockUI*: coroutines that are launched from the UI thread should keep the thread lightweight;
- *UpdateUI*: coroutines that are not launched from the UI thread should never interact with UI elements;
- *ResumeIfNeeded*: coroutines carrying out slow computations should check periodically that they are still needed.

4 A Coroutine-Aware Specification Language

We define a tailor-made version of LTL for coroutines in the context of Kotlin. Our requirements were the ability to specify the behaviour of coroutines and their relation to other objects (e.g., scopes and threads). We add a first-class notion of coroutines, allowing quantification over them. We consider a set of events that allow relating coroutines with other objects in the language and relating coroutines with each other. To allow for this, our alphabet is two-layered: general program events and events tied to a coroutine.

Definition 1. A corLTL specification π is defined by the following grammar:

$$\begin{aligned}
\Sigma_x &\stackrel{\text{def}}{=} \text{LaunchType } x \mid \text{AsyncType } x \mid \text{Running } x \mid \text{Active } x \mid \text{Blocked } x \mid \\
&\quad \text{OnMainThread } x \mid \text{TransferToMainThread } x \mid \text{HasActiveScope } x \mid \dots \\
\psi_x &\stackrel{\text{def}}{=} \Sigma \mid \Sigma_x \mid \pi \mid \psi_x \wedge \psi_x \mid \psi_x \mid X\psi_x \mid \psi_x U \psi_x \\
\pi &\stackrel{\text{def}}{=} \pi \wedge \pi \mid \neg \pi \mid \forall x : \text{coroutines} . \psi_x
\end{aligned}$$

We define disjunction (\vee), globally (G), eventually (F), and existential quantification (\exists) as usual. We limit ourselves to well-formed formulas where every variable appearing in the formula is bound. We assume predicates and relations are typed, and use only well-typed formulas. We assume a static number n of coroutines that all exist at the start of a program, with unique identifiers from $ID = \{0, \dots, n\}$. We then can define system traces as sequences over sets of $\Sigma \cup \{p_n \mid p_x \in \Sigma_x \wedge i \in ID\}$. We number events to relate to a coroutine (e.g., p_1 and p'_1 are about the coroutine with identifier 1).

The semantics of corLTL extends that of LTL to reason about coroutines.

Definition 2 (Semantics). We say the infinite trace w is a model of π , denoted by $w \models \pi$, according to the rules in Table 1.

$$\begin{aligned}
w \models \pi \wedge \pi' &\stackrel{\text{def}}{=} w \models \pi \wedge w \models \pi' \\
w \models \neg \pi &\stackrel{\text{def}}{=} \neg(w \models \pi) \\
w \models \forall x : \text{coroutines} \cdot \psi &\stackrel{\text{def}}{=} \forall i \in ID \cdot w \models \psi[x/i] \\
w \models p &\stackrel{\text{def}}{=} p \in w[0] \\
w \models p_n &\stackrel{\text{def}}{=} p_n \in w[0] \\
w \models X\psi &\stackrel{\text{def}}{=} w_1 \models \psi \\
w \models \psi \wedge \psi' &\stackrel{\text{def}}{=} w \models \psi \wedge w \models \psi' \\
w \models \neg \psi &\stackrel{\text{def}}{=} \neg(w \models \psi) \\
w \models \psi U \psi' &\stackrel{\text{def}}{=} \exists j \cdot w_j \models \psi' \wedge \forall k \cdot 0 \leq k < j \implies w_k \models \psi
\end{aligned}$$

Fig. 1. corLTL semantics where w_x denotes the x -th strict suffix of w .

With appropriate events, we can capture the properties from the previous section and more.

Example 1. A coroutine can only be active if its scope is still active: $\forall x : \text{coroutines} \cdot G(\text{active}(x) \implies \text{hasActiveScope}(s))$.

Example 2. When a coroutine is doing an I/O operation, then it currently is on a background thread: $\forall x : \text{coroutines} \cdot G(\text{doingIO}(x) \implies \neg \text{onMainThread}(x))$.

Example 3. When a coroutine is updating the UI then it is on the main thread: $\forall x : \text{coroutines} \cdot G(\text{updatingUI}(x) \implies \text{onMainThread}(x))$.

We can also express application-specific properties.

Example 4. A job is handled only once: $\forall x : \text{coroutines} \cdot G(\text{handledJob}(x) \implies XG(\neg \text{handledJob}(x)))$.

5 Monitoring Kotlin Coroutines

In this section, we discuss monitoring for corLTL specifications. Since corLTL has the full power of LTL, it is not fully monitorable, instead in this language we focus on the safety subset of corLTL, consisting of the negation normal form (NNF) and until/release-free subset. Thus we restrict ψ in Definition 2 with negation only on the atomic events (and π), without until but with *weak until* (W). We consider two options for monitoring for this sub-language.

Standard LTL monitoring One could try to re-use standard LTL monitoring for our tailor-made logic, by transforming a corLTL formula into a standard LTL formula by eliminating the quantifiers recursively as follows: $\forall x : \text{coroutines} \cdot \psi \iff \bigwedge_{i \in ID} \psi[x/i]$.

Each coroutine could be instrumented to output appropriately labelled events to a channel that is only listened to by a monitor for this quantifier-free formula. However, this transformation is exponential in the number of nested quantifiers. It bears asking then whether nested quantifiers are useful. The basic properties about coroutines we have detailed do not require these, but program-specific properties may require the power of nested quantifiers, for example:

Example 5. If a coroutine holds a resource then no other coroutine also holds it:
 $\forall x : \text{coroutines} \cdot \text{HoldsResource } x \implies \nexists y : \text{coroutines} \cdot \text{HoldsResource } y.$

Instead of paying the cost for the exponential transformation, we next explore the option to distribute monitoring over the coroutines through automata communicating through channels.

Communicating automata Given a specification, assign every quantification with an identifier from \mathbb{N} , abstract each away by replacing it with an event e^j (j corresponding to the identifier of the quantified sub-formula). Then the specification becomes an LTL formula over $\Sigma \cup \{e^j | j \in \mathbb{N}\}$. For Example. 5 we can assign the event e^0 to $\nexists y : \text{coroutines} \cdot \text{HoldsResource } y$, and e^1 to $\forall x : \text{coroutines} \cdot \text{HoldsResource } x \implies e^0$, and the top-level formula is just e^1 .

A monitor can be extracted for the top-level formula, and other monitors for the formula corresponding to each e^j . An issue is that the value of e^j at a time-step t may only be knowable in a future time-step, e.g. if $e^j = \forall x : \text{coroutines} \cdot Xe$. The monitor would then have to branch on both values, and discard one of the branches when the value of e^j is eventually set.

Here, for simplicity and as a first step, we further restrict the language by only allowing quantification at the high level of a formula (π), and disallowing quantification at the LTL level. Then we can re-use standard LTL synthesis for the LTL parts of the specifications and use communicating automata to monitor for the high-level logic. We illustrate our proposed approach with a variation on symbolic automaton monitors (e.g., [5, 3]).

Definition 3 (DEAC). A *Dynamic Event Automaton with Channels (DEAC)* is a tuple $D_x = \langle C_r, C_s, \Sigma, \Sigma_x, \Sigma_c, Q, V, q_0, \theta_0, A, B, \rightarrow \rangle$, where $C_r, C_s \in \mathbb{C}$ are finite sets of channels (s.t. $C_r \cap C_s = \{\}$), Σ is a finite alphabet, Σ_x is a finite alphabet over a free variable x , Σ_c is a finite set of channel events, Q is a finite set of states, V is a finite set of variables, $q_0 \in Q$ is the initial state, $\theta_0 : V \rightarrow \mathbb{VAL}$ is the initial valuation of the variables V , $A \subseteq Q$ is the set of accepting states, $B \subseteq Q$ is the set of bad states, and $\rightarrow : Q \times (2^{\Sigma \cup \Sigma_x \cup (\Sigma_c \times C_r)} \times V \rightarrow \{\text{true}, \text{false}\}) \rightarrow (2^{\Sigma_c \times C_s} \times (V \rightarrow \mathbb{VAL}) \times Q)$ is the transition function guarded by sets of program, coroutine-specific, and channel events received from C_r , and the current variable valuation, and that can send events on C_s .

We write $q \xrightarrow{g \rightarrow (\text{out}, a)} q'$ for $(q, g, \text{out}, a, q') \in \rightarrow$. We write $\neg D$ for the DEAC D with the accepting and bad states swapped. We write $t \in L(D)$ when t is a trace over events and channel events that reaches an accepting state in D .

$$\frac{q \xrightarrow{g \mapsto (ecs', a)} q' \quad g(E, ecs, \theta) \quad q \notin A \cup B}{(q, \theta, ecs'') \xrightarrow{E} (q', a(E, ecs, \theta), (ecs'' \setminus 2^{\Sigma_C \times C_r}) \cup ecs')} \quad \frac{\text{otherwise}}{(q, (\theta, ecs)) \xrightarrow{E} (q, (\theta, ecs))}$$

Fig. 2. DEAC semantics

$$\begin{array}{c} \frac{q_1 \xrightarrow{g_1 \mapsto (ecs_1, a_1)} q'_1 \quad q_2 \xrightarrow{g_2 \mapsto (ecs_2, a_2)} q'_2}{(q_1, q_2) \xrightarrow{g_1 \wedge g_2 \mapsto (ecs_1 \cup ecs_2, a_1 \circ a_2)} (q'_1, q'_2)} \\ \\ \frac{q_1 \xrightarrow{ecs|g_1 \mapsto (ecs_1, a_1)} q'_1 \quad q_2 \xrightarrow{ecs|g_2 \mapsto (ecs_2, a_2)} q'_2}{(q_1, q_2) \xrightarrow{ecs|g_1 \wedge \neg g_2 \mapsto (ecs_1, a_1)} (q'_1, q_2)} \\ \\ \frac{q_1 \xrightarrow{ecs|g_1 \mapsto (ecs_1, a_1)} q'_1 \quad \nexists q_2 \xrightarrow{ecs|g_2 \mapsto (ecs_2, a_2)} q'_2}{(q_1, q_2) \xrightarrow{ecs|g_1 \mapsto (ecs_1, a_1)} (q'_1, q_2)} \end{array}$$

Fig. 3. Network of DEACs

We define our variation of DATEs, DEACs, here to include a semantics where a buffer of channel events is kept, with a DEAC consuming events sent on the channels it is listening to (C_r).

Definition 4. *The operational semantics of DEACs, presented in Fig. 2, is given over configurations of triples of states, valuations $(Q, V \rightarrow \mathbb{V}_{\text{ALL}})$, and sets of channel events, with transitions labelled by channel events.*

We characterise a network of DEACs that communicate with each other through these channel events as their composition.

Definition 5. *A network of two DEACs D_x^1 and D_y^2 , with non-intersecting receive channels ($C_r^1 \cap C_r^2 = \{\}$), denoted by $D_x^1 \| D_y^2$, is a DEAC $\langle C_r^1 \cup C_r^2, C_s^1 \cup C_s^2, \Sigma, \Sigma_x \cup \Sigma_y, \Sigma_C, Q^1 \times Q^2, V^1 \cup V^2, (q_0^1, q_0^2), (\theta_0^1, \theta_0^2), A^1 \times A^2, (B^1 \times Q^2) \cup (Q^1 \times B^2), \rightarrow \rangle$, with the transition function \rightarrow being the composition of both DEACs' transition functions as in Fig. 3 (the last two rules apply symmetrically for D_y^2).*

The safety subset of *LTL* is monitorable, and a corresponding deterministic finite-state automaton (doubly exponential in the size of the formula) exists [17]. We thus assume such construction from safety LTL formulas to DEACs, and we denote the DEAC corresponding to an LTL formula ψ by $aut(\psi)$. For every LTL formula ψ , in a (restricted) corLTL specification, we assume a corresponding event, denoted by e_ψ^v , where $v \in \{\top, \perp\}$ denotes the verdict on ψ . We assume the construction of $aut(\psi)$ is such that e_ψ^\top is outputted to a channel c_x on accepting transitions, and e_ψ^\perp is outputted to c_x on bad transitions.

$$\begin{aligned}
mon(\forall x : coroutines \cdot \psi) &\stackrel{\text{def}}{=} D_\psi \| (\|_{n \in ID} aut(\psi)[x/n]) \\
mon(\pi \wedge \pi') &\stackrel{\text{def}}{=} mon(\pi) \| mon(\pi') \\
mon(\neg \pi) &\stackrel{\text{def}}{=} \neg mon(\pi)
\end{aligned}$$

Fig. 4. Monitor construction for corLTL.

Given an LTL formula ψ we define D_ψ as the DEAC with $C = \{c_i \mid i \in ID\}$, $\Sigma_C = \{e_\psi^\top, e_\psi^\perp\}$, $Q = \{q_0, q_A, q_B\}$, $V = \{v_i \mid i \in ID\}$, θ_0 sets every v_i to \perp , q_A being the only accepting state, q_B being the only bad state, and with

$$\rightarrow = \{q_0 \xrightarrow{\bigvee_{c \in C_r} (e_\psi^\perp, c) \in ecs \mapsto (\{\}, \bigwedge_{i \in ID} (e_\psi^\top, c_i) \in ecs \implies v'_n = \top)} q_0\} \cup \{q_0 \xrightarrow{\bigvee_{c \in C_r} (e_\psi^\perp, c) \in ecs} s_B\} \cup \{q_0 \xrightarrow{\bigwedge_{i \in ID} v_i} s_A\}.$$

Note how this automaton remains in q_0 , setting v_i to be true when ψ is identified as true on c_i , while if all variables become true then there is a transition to the accepting state. If instead ψ is false on some c_i then there is a transition to a bad state.

A monitor (for a top-level safety formula) can be given as shown in Fig. 4. We can show correspondence between the infinite traces models of a property and the infinite traces accepted by a corresponding monitor:

Theorem 1. *For a safety corLTL formula π : $w \models \pi \iff w \in L(mon(\pi))$.*

However, note that the monitor may take an extra step to determine satisfaction given finite traces since D_ψ separates the step of marking satisfying events and that of determining whether all the coroutines have satisfied the property. It should be clear that a monitor that does this at the same time can be constructed but would have more complex guards and actions.

6 Implementation and Evaluation

In order to monitor the properties identified earlier, we developed an API⁶ that would be as transparent as possible to a developer. This was achieved by creating a new interface called `MonitoredComponent`, implemented by subclasses of `Activity` and `ViewModel` that would provide a familiar set of utilities and coroutine builders while carrying out the monitoring under the surface.

The interface holds records of any tasks started, as well as their dispatchers and exception handlers:

- `recommendedDispatchers`, a `HashMap` storing the best coroutine dispatcher to use with each task;
- `defaultHandler`, a `CoroutineExceptionHandler` that should be inserted into unhandled coroutine contexts according to `NeedHandler`;
- `monitoredApplication`, an accessor providing communication between the component and the `MonitoredApplication` instance.

⁶ The code can be found at <https://gitlab.com/denf86/kotlin-rv>.

The two implementations of `MonitoredComponent` provided by the API are `MonitoredActivity` and `MonitoredViewModel`. They expose an overloaded version of the coroutine builder methods that uses a lifecycle-aware scope by default (identified as the `lifecycleScope` and `viewModelScope`, respectively, both provided by `kotlinx`) to uphold the *DestroyedWithOwner* property.

Since tasks started via the `launch` method will only employ the handler at top level, the subclasses of `MonitoredComponent` try and extract a coroutine handler from inside the context. If no handler is found, the `defaultHandler` will be added so as to ensure that an instance of `CoroutineExceptionHandler` is present at all times. For example, any invocation of `viewModelScope.launch { foo() }` in a handler-less context is translated to `viewModelScope.launch(defaultHandler) { foo() }`. The `MonitoredComponent` has, however, no way of knowing whether the task currently being started lies in the top level or not.

In the case of the `async` coroutine builder, the `MonitoredComponent` focuses instead on re-throwing any exceptions according to *ExceptionalAsync*.

Any given instance of `MonitoredComponent` upholds the properties *Slow-DownUI* and *UpdateUI* by means of runtime enforcement. While it has no way of knowing in advance what a block of code will do once executed, it remembers what a given block of code did during its last execution. In order to do so, it detects cases of `CalledFromWrongThreadException` and `NetworkOnMainThreadException` being thrown inside a coroutine and uses them to infer what thread may be a better choice should the same task be executed again. The component then adds an entry to the `recommendedDispatchers` consisting of the inferred ideal dispatcher and an identifier that was arbitrarily composed of class and name, as well as line number, of the method invoked inside the failed coroutine, read from the exception stack trace.

To provide an example, if the same I/O method `readFromFile` is launched inside a coroutine on the UI thread twice in a row:

- the first time will result in failure with a `NetworkOnMainThreadException` and the `MonitoredComponent` will update the `recommendedDispatchers` with a new entry `Dispatchers.IO` for this method;
- the second time, the `MonitoredComponent` will look up the entry created and overwrite the given dispatcher with `Dispatchers.IO`, allowing the method to execute correctly.

Since a task can fail for any given exception type outside of the above two, the `MonitoredComponent` internally replaces the first occurrence of either `CalledFromWrongThreadException` or `NetworkOnMainThreadException` with a newly-defined `WrongDispatcherException`, with the original as its cause, and rethrows it to the upper layer. At the top level, the `CoroutineExceptionHandler` can detect whether the crash was originally triggered by a `WrongDispatcherException` and only then will it save the new `recommendedDispatchers` entry.

The entries saved are stored inside a `MonitoredApplication`, which extends the standard Android application. Despite its name, the `MonitoredApplication`

is not instrumented but only contains a map of the `recommendedDispatchers` collections for each `MonitoredComponent` in the app. The map is loaded by monitored component instances during their initialisation and updated by them before their deletion.

Right before the app is terminated, a service prints the content of each saved `recommendedDispatchers` map. This string is visible on the device log, which was thought to be a good compromise between storing everything in memory and creating an output file. The printout looks like this:

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

The output above means that the coroutine launched on l.51 of the `loadFrom` function (source file “View.kt”) should use a dispatcher for background threads while the invocation on l.47 of the same function should use the UI thread.

Application-specific properties We tested the API on a simple Android app developed ad-hoc: this application would look for images on an online repository and display them on screen after downloading their bytes in an asynchronous task in a coroutine. We identified some more properties to monitor, specific to the application at hand. We defined two new properties:

- *AlwaysOneJob* checks that only one task can look up images at a time: multiple lookup operations would overwrite the displayed list of images, resulting possibly in a waste of mobile data when the search button is double tapped;
- *SuccessWithJSON* checks that the app can normally execute in the case of the expected image data, which should always be a JSON object, being malformed or otherwise unreadable.

Evaluation of the MonitoredComponent API. After it had been tested and benchmarked on an ad-hoc Android app, the API was found to work on a general case but still needs some improvements. Overall, the recognition of thread-based crashes was found to not be foolproof: it could not handle a use case where the recommended dispatcher was not the best option, and it needed to experience a small number of crashes before the `recommendedDispatchers` map had enough entries to be reliable. Its main weakness was, in fact, the reliance on a task failing several times with the *right* exceptions.

The benchmarks, carried out using the Android Jetpack tools, showed that the overheads added by the API could grow significantly:

- tasks created with `launch` could be between 0.46 – 1.60% slower;
- tasks created with `async` could be between 0.45 – 0.86% slower;
- running a thousand `async` tasks in parallel could take between 56.43 – 144.49% more time while using the API.

Table 1. Sony device benchmarks.

Test	Execution time (ms)		Overhead	
	No monitors	Monitored	(ms)	(%)
Tasks created with <code>launch</code>	1022	1039	17	1.60
Tasks created with <code>async</code>	1026	1034	8	0.86
1000 * parallel <code>async</code>	1198	1873	675	56.43

Table 2. Huawei device benchmarks.

Test	Execution time (ms)		Overhead	
	No monitors	Monitored	(ms)	(%)
Tasks created with <code>launch</code>	1008	1013	5	0.46
Tasks created with <code>async</code>	1008	1012	4	0.45
1000 * parallel <code>async</code>	1446	3521	2075	144.49

Performance was measured with executing the same methods ten times on two smartphone models, a Sony Xperia XZ2 Compact H8324 and a Huawei Y6 ATU-L21. See the results in Table 1 (Table 2) for the Sony (Huawei) device.

The instrumented app was additionally tested for its memory footprint using Android Studio’s built-in profiler. This exposed another weakness of the `recommendedDispatchers` map in that storing the class names as keys meant that the longer a name of a class, the more space it would take.

7 Related Work

Runtime verification of concurrency has mostly focused on checking generic properties such as deadlock freedom, the absence of data races, atomicity violations, etc. For this, specific approaches and tools exist, such as [13, 1, 14]. See [18] for a recent description of existing properties and approaches as well as [12] for some dedicated tools. In addition, approaches to monitoring user-provided properties in mono-threaded programs have been lifted to to multithreaded ones as are. However, the soundness of the produced verdicts depends on the specifications and the program locations producing the events of interest [9].

The approach described in this paper is novel in that it introduces support for runtime verification in the Kotlin programming language. We note that, even though Kotlin compiles to (Java) bytecode, existing approaches to the monitoring of Java programs cannot be applied for our purposes. One originality of our approach is in the runtime verification of a specific concurrency construct, though we verify generic properties as well as program-specific properties. It is the specific form of structured concurrency that allows the design of a tailored specification language to express coroutine-specific properties. We note that we have not described desirable properties of coroutines that could be checked statically even though they are easily expressible in our framework. However, they are implemented and available in our tool since, to the best of our knowledge, there is no static analyser to check these properties. Henceforth, while corou-

tines facilitate concurrent programming, they are prone to errors, and our tool approach provides programmers with the means to debug their programs.

The official library `kotlinx.coroutines` provides a basic set of debugging tools, consisting of a *debug mode* and a *stacktrace recovery* feature. The same Kotlinx library also provides an experimental module dedicated to debugging: this keeps records of all coroutines alive and introspecting and dumping them to enhance stacktraces with additional information like where a coroutine was created. The module can be used as a standalone JVM agent. This enables debug probes on the application startup and allows the monitoring of the whole application. However, the overheads caused by the recording and dumping of each coroutine are very noticeable and not recommended in production.

Finally, we mention a couple of Kotlin static analysis tools. The Detekt [8] tool allows checking six predefined rules on coroutines. The tool allows programmers to expand the range of checks by defining custom rules. More rules for Kotlin coroutines are provided by Sonar [21], a company leader in code analysis. These rules are taken from the official Kotlin guidelines and include some of the cases already mentioned for Detekt. Compared to these, our tool benefits from the same advantages and limitations as runtime verification over static analysis but could benefit from a combination with the latter (e.g., as in [2]).

8 Concluding Remarks

We have introduced a language for writing properties about coroutines, and we have provided an implementation to verify several properties concerning the execution of coroutines in Kotlin.

Despite our approach being (in theory) usable both pre-deployment (as a debugging tool) and post-deployment (as a monitor during the real execution of the system), we have, in this paper, only focused on the former. Our benchmarks show that the introduced overhead would be hardly noticeable by a developer but may have scalability issues in extreme situations. As a first proof-of-concept, we have thus implemented our monitors as (hardcoded) coroutines for all the identified properties instead of as a general monitoring tool that extracts the monitors for the properties (written in our language). The implementation of a dedicated tool to write properties using our language is left for future work.

We have applied our implementation to many programs and identified, in some cases, that the properties were violated. Though some of the programs we used in our evaluation were written with the explicit intention of producing the error, the value of the exercise relies on that: i) the errors are not easy to detect (they would be very difficult or impossible to be identified by the programmer); ii) the errors were detected by our implemented monitors.

Note that all properties capture actual potential problems of coroutines in Kotlin. The only exceptions to this in newer versions of Kotlin are *NormalAsync* and *ExceptionalAsync*, which are solved with exception suppression.

Finally, as Kotlin became the favourite language for developing Android applications, our approach allows revisiting the existing monitoring frameworks for Android [10, 11, 7, 23].

References

1. Agarwal, R., Bensalem, S., Farchi, E., Havelund, K., Nir-Buchbinder, Y., Stoller, S.D., Ur, S., Wang, L.: Detection of deadlock potentials in multithreaded programs. *IBM J. Res. Dev.* 54(5), 3 (2010), <https://doi.org/10.1147/JRD.2010.2060276>
2. Azzopardi, S., Colombo, C., Pace, G.J.: CLARVA: model-based residual verification of java programs. In: *MODELSWARD2020*. pp. 352–359. SCITEPRESS (2020), <https://doi.org/10.5220/0008966603520359>
3. Azzopardi, S., Ellul, J., Pace, G.J.: Monitoring smart contracts: Contractlarva and open challenges beyond. In: *RV18. LNCS*, vol. 11237, pp. 113–137. Springer (2018), https://doi.org/10.1007/978-3-030-03769-7_8
4. Clarke, E.M.: Proving the correctness of coroutines without history variables. In: *16th Annual Southeast Regional Conference*. p. 160–167. ACM-SE 16, ACM (1978), <https://doi.org/10.1145/503643.503680>
5. Colombo, C., Pace, G.J., Schneider, G.: Dynamic event-based runtime monitoring of real-time and contextual properties. In: *FMICS’08. LNCS*, vol. 5596, pp. 135–149. Springer (2009)
6. Conway, M.E.: Design of a separable transition-diagram compiler. *Commun. ACM* 6(7), 396–408 (1963)
7. Daian, P., Falcone, Y., Meredith, P.O., Serbanuta, T., Shiriashi, S., Iwai, A., Rosu, G.: Rv-android: Efficient parametric android runtime verification, a brief tutorial. In: Bartocci, E., Majumdar, R. (eds.) *Runtime Verification - 6th International Conference, RV 2015 Vienna, Austria, September 22–25, 2015. Proceedings. Lecture Notes in Computer Science*, vol. 9333, pp. 342–357. Springer (2015)
8. Detekt team: Detekt. <http://detekt.dev>, accessed: 2022-05-18
9. El-Hokayem, A., Falcone, Y.: Can we monitor all multithreaded programs? In: *RV’18. LNCS*, vol. 11237, pp. 64–89. Springer (2018)
10. Falcone, Y., Currea, S.: Weave droid: aspect-oriented programming on android devices: fully embedded or in the cloud. In: *IEEE/ACM International Conference on Automated Software Engineering, ASE’12, Essen, Germany, September 3–7, 2012*. pp. 350–353. ACM (2012)
11. Falcone, Y., Currea, S., Jaber, M.: Runtime verification and enforcement for android applications with rv-droid. In: *Runtime Verification, Third International Conference, RV 2012, Istanbul, Turkey, September 25–28, 2012, Revised Selected Papers. Lecture Notes in Computer Science*, vol. 7687, pp. 88–95. Springer (2012)
12. Falcone, Y., Krstic, S., Reger, G., Traytel, D.: A taxonomy for classifying runtime verification tools. *Int. J. Softw. Tools Technol. Transf.* 23(2), 255–284 (2021)
13. Havelund, K., Rosu, G.: An overview of the runtime verification tool java pathexplorer. *Formal Methods Syst. Des.* 24(2), 189–215 (2004), <https://doi.org/10.1023/B:FORM.0000017721.39909.4b>
14. Huang, J., Meredith, P.O., Rosu, G.: Maximal sound predictive race detection with control flow abstraction. In: *PLDI’14*. pp. 337–348. ACM (2014)
15. Kotlin: Language Documentation, <https://kotlinlang.org/docs/home.html>
16. Kotlin documentation: Dispatchers and threads, <https://kotlinlang.org/docs/coroutine-context-and-dispatchers.html#dispatchers-and-threads>, accessed: 2022-05-18
17. Kupferman, O., Vardi, M.Y.: Model checking of safety properties. In: *CAV’99. LNCS*, vol. 1633, pp. 172–183. Springer (1999), https://doi.org/10.1007/3-540-48683-6_17

18. Lourenço, J.M., Fiedor, J., Krena, B., Vojnar, T.: Discovering concurrency errors. In: Lectures on Runtime Verification - Introductory and Advanced Topics, LNCS, vol. 10457, pp. 34–60. Springer (2018)
19. Marlin, C.D.: Coroutines: A Programming Methodology, a Language Design and an Implementation, LNCS, vol. 95. Springer (1980)
20. de Moura, A.L., Ierusalimsky, R.: Revisiting coroutines. *ACM Trans. Program. Lang. Syst.* 31(2), 6:1–6:31 (2009)
21. SonarSource S.A.: Kotlin rules for coroutines, <https://rules.sonarsource.com/kotlin/tag/coroutines>, accessed: 2022-05-18
22. Sústrik, M.: Blog post detailing structured concurrency, <http://250bpm.com/blog/71>, accessed: 2022-05-18
23. Vella, M., Colombo, C.: Spotcheck: On-device anomaly detection for android. In: Örs, S.B., Elçi, A. (eds.) *SIN 2020: 13th International Conference on Security of Information and Networks*, Virtual Event / Istanbul, Turkey, November 4-6, 2020. pp. 20:1–20:6. ACM (2020)