Federal University of Pernambuco
Center of Informatics
BSc Computer Science

CEPlin: A Complex Event Processing Framework for Kotlin

Jonas de Araújo Lins

Undergraduate Thesis

Supervisor: Kiev Santos da Gama

Recife
November, 2018

Federal University of Pernambuco
Center of Informatics
BSc Computer Science

Jonas de Araújo Lins

CEPlin: A Complex Event Processing Framework for Kotlin

Thesis submitted to the Center of Informatics of the Federal University of Pernambuco as partial requirement for the degree of Bachelor of Science in Computer Science.

Supervisor: Kiev Santos da Gama

Recife
November, 2018

Acknowledgements

A very special thank you to my parents, my sister, my girlfriend and those who supported me during this long journey. A thank you to my friends who made my college routines funnier. I would also like to thank Kiev for the patience and comprehension during this work.

Abstract

Reactive applications has been gaining traction in the industry due to the ever increasing demand for interactive systems of information that reacts to the user in a timely manner such as mobile apps, web systems, sensor networks, monitoring system, and others. Therefore, many approaches have been adopted by different communities in order to develop such applications. Amongst many of them, we can highlight Reactive Languages and Complex Event Processing due their similarities that were identified by previous researches. However, there are few solutions available that integrates the best of both solutions. In this work, the goal is to implement and publish a new framework written in Kotlin and using reactive programming in order to make the development of CEP applications easier.

Keywords: Reactive Programming, Complex Event Processing, Event-Driven Programing.

Resumo

Aplicações reativas têm se tornado cada vez mais presentes devido a crescente demanda de sistemas interativos e em tempo real como aplicativos móveis, sistemas web, redes de sensores, sistemas de monitoramento, entre outros. Portanto, várias abordagens têm sido adotadas para desenvolver tais aplicações, dentre elas, podemos destacar as Linguagens Reativas (ou RL, do inglês *Reactive Language)* e o Processamento de Eventos Complexos (ou CEP, do inglês *Complex Event Processing)*. Devido a semelhanças identificadas em pesquisas entre CEP e RLs, é possível implementar conceitos de CEP utilizando programação reativa, porém ainda não há uma vasta cobertura de soluções que utilizem as duas abordagens de forma complementar. Neste trabalho, o objetivo será desenvolver e disponibilizar um framework escrito em Kotlin, utilizando programação reativa, para facilitar o desenvolvimento de aplicações que utilizem CEP.

Palavras-chave: Programação Reativa, Processamento de Eventos Complexos, Programação Orientada a Eventos.

List of Figures

Figure 1 - CEP Application Flow on the Healthcare Example 11

Figure 2 - Figure 2 – ReactiveX Operations over Stream 12

Figure 3 - A Comparison of how CEP and RLs Implement the Five Phases of a Reactive Behavior [1] 13

Figure 4 - Framework high-level architecture (Adapted from Guedes, 2017, p. 25) [2] 17

Figure 5 - CEPlin UML class diagram 19

Figure 6 - TouchEvent class that models the user screen touches 21

Figure 7 - EventManager instance for TouchEvent 21

Figure 8 - Horizontal gesture rule using sequence operator 22

Figure 9 - Horizontal gesture rule using ReactiveX operators 22

Figure 10 - ProximityEvent class that models the phone proximity to an object 23

Figure 11 - AccelerationEvent class that models the phone proximity to an object 23

Figure 12 - EventManager instances for ProximityEvent and AccelerationEvent 24

Figure 13 - Phone in pocket action rule using CEPlin operators 24

Figure 14 - Phone in pocket action rule using ReactiveX operators 25

List of Tables

Table 1 - Available Operators. Summarized from Cugola and Margara, 2012 [3] ……….. 14

Table 2 - Implemented Operators ……………………………………………………….. 20

Table 3 - Metrics for the horizontal gesture case ……………………………………….. 23

Table 4 - Metrics for the phone in pocket action case ……………………………………...25

List of Acronyms

CEP	Complex Event Processing
RL	Reactive Language
IFP	Information Flow Processing
LOC	Lines of Code

Contents

1. Introduction **10**

2. Background **11**

2.1. Complex Event Processing 11

2.2. Reactive Language 12

2.3. CEP and Reactive Language integration 13

2.4. CEP Operators 14

2.5. Kotlin 16

3. Implementation **17**

3.1. Proposal 17

3.2. Core entities 18

3.3. Libraries and tools 19

3.4. Implemented operators 19

4. Use cases **21**

4.1. Gesture sequence detection 21

4.2. Phone in pocket action detection 23

4.3. Discussion 25

5. Conclusion **27**

6. References **28**

1. Introduction

A reactive application can be defined as software capable of detecting changes and reacting to events of interest. These applications vary from simple detection of changes on user interface elements to financial fraud detection, IoT data insights, sensor network monitoring systems. The ever increasing demand for this type of application can be easily identified as the society and machines are more connected to multiple sources of information, which in turn should be properly handled by a software system in order to react to them in a timely manner.

Several approaches have been proposed and deployed by different communities to address the development of such applications. Particularly, two of them have been researched [1] and compared to the extent of their differences, similarities and even integration, they are: Reactive Languages (RL) and Complex Event Processing (CEP), an event-based programming approach.

Due to similarities between CEP and Reactive Languages, thoroughly researched by [1] and further elaborated by [2][4], it has been possible to implement CEP concepts by using reactive programming.

For this reason, this work aims to develop a new CEP framework using reactive programming, provided by Reactive Extensions, and Kotlin as the functional language, which inspired the name of framework: CEPlin.

This document is organized as follows: Chapter 2 briefly introduces CEP and RLs concepts as well as related works that motivate the development of this framework. Chapter 3 discusses the framework proposal. Chapter 4 details the implementation. Chapter 5 demonstrates the work by implementing use cases. Finally, Chapter 6 discusses conclusions and future work.

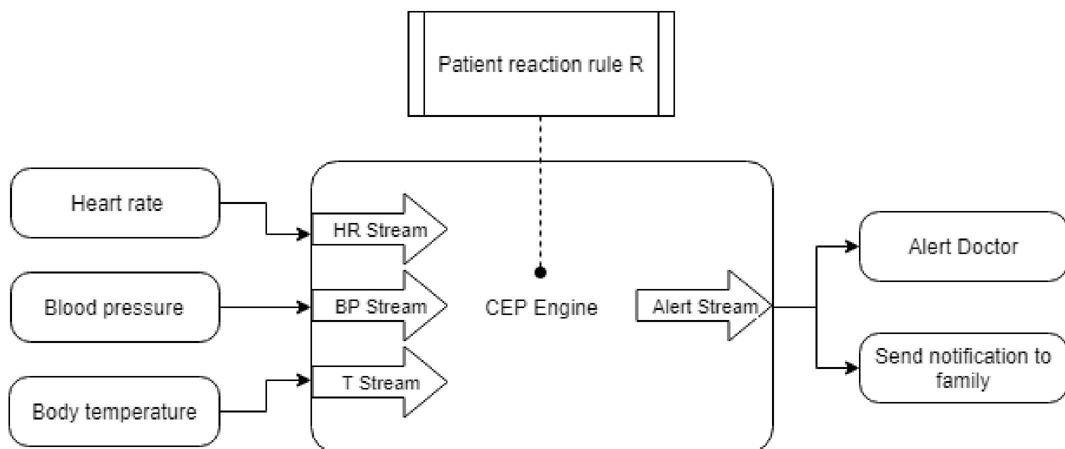
2. Background

This chapter introduces the important concepts and tools related to this work. First, CEP and RLs are introduced, then their comparison and integration are discussed. After that, CEP operators are presented. Finally, the Kotlin language used in this work is briefly introduced.

2.1. Complex Event Processing

The role of Complex Event Processing can be summarized as the interest of combining multiple streams of information to detect complex pattern situations and notify them as they happen. For example (Figure 1), in healthcare a doctor may want to check the patient heart rate, temperature and blood pressure to monitor if any reaction starts after taking a certain medication. In this case, we have three streams of information to be considered: heart rate, temperature and blood pressure. A bad reaction R could be identified and notified to the doctor as soon as both blood pressure and heart rate starts increasing within a time frame of 30 seconds while the temperature stays stable within a given range. The notification can provide sufficient feedback to the doctor to react accordingly given that they now know a reaction R is starting to develop on the patient.

Figure 1 - CEP Application Flow on the Healthcare Example



The CEP concept itself belongs to a wider domain class of systems called as Information Flow Processing (IFP) systems. The IFP domain includes all solutions that are designed to process flows of information on-the-fly, to timely compute and update results [1].

Data Stream Management Systems (DSMS), for instance, also belong to this domain and the increasing demand for timely high data processing throughput led the database community to develop it as a response to the traditional Database Management System (DBMS) whose data availability is slow in a reactive scenario due its dense persistence requirements.

Although DSMSs solve the data streaming issue, it cannot extract high-level situations, this task is generally left to their clients. This is the requirement that Complex Event Processing covers, CEP engines not only allow the filtering, combination and aggregation over data streams, but also the detection and notification of high-level event occurrences.

2.2. Reactive Language

A Reactive Language abstracts the handle of streams so that a programmer can focus on managing them by defining the business logic directly instead of coding the stream handling itself. Still on the healthcare case, a RL fits as the tool to detect and propagate changes of each interested data source over time, e.g. every time the patient temperature changes the reactive application is notified so it can react to that information.

Although RLs were first introduced as dedicated functional languages, known as Functional Reactive Programming (FRP) [9], other approaches have been developed as libraries, design pattern extensions and other abstractions. ReactiveX (Rx) [7], for instance, provides reactive capabilities for multiple languages and platforms and it has been widely used in industry. Essentially, ReactiveX extends the Observer pattern to deal with asynchronous data streams and it also takes advantage of functional programming to provide seamlessly composition of operators over streams.

Figure 2 – ReactiveX Operations over Stream

```
heartRateObservable
    .filter{ it > 100 }
    .map{ HeartRateEvent(it) }
    .subscribe { it: HeartRateEvent!
        doSomething()
    }
```

Figure 2 shows how you can observe only the relevant values by applying an operator function called *filter,* which evaluates if the input stream emits a value over 100 beats per second, then transforming the filtered stream into a HeartRateEvent instance in the system by using the *map* operator and finally subscribing to the resulting stream that notifies when the desired event is emitted.

It is important to notice that reactive programming, its languages and tools generally aim to achieve the same concept of dealing with changes of values over time, thus we naturally see common subjects being often mentioned among the IFP domain.

2.3. CEP and Reactive Language integration

Margara and Salvaneschi were able to identify relevant similarities between RL and CEP which then triggered the development of frameworks to implement CEP operators using reactive programming [1]. In order to compare the two domains, they presented five main phases that represent the behavior of a reactive application as shown below:

- **Observation:** this phase stands for the observation of changes of an interested source;
- **Notification:** after a change happens, the notification is triggered and sent to the system;
- **Processing:** the system process the notification through a set of rules or expressions;
- **Propagation:** the results of the processing phase is propagated to all interested components;
- **Reaction:** finally, all the interested components receive the results and can react accordingly.

Figure 3 - A Comparison of how CEP and RLs Implement the Five Phases of a Reactive Behavior [1]

Phase	CEP	RLs
Observation	Generic Events	Value Changes
Notification	Explicit – Push	Implicit – Push
Processing	Rules (from primitive to composite events)	Expressions (from signals to signals)
Propagation	Explicit – Multicast – Push	Implicit – Multicast – Push or Pull
Reaction	Generic Procedures – User-Defined	Value Changes

As shown in the comparison between CEP and RLs in Figure 3, the CEP observes the occurrences of events rather than the value changes of a source as RLs do. The notification of a CEP system is explicit, i.e., the user must push an event to the CEP engine, whereas in the RL the notification is implicitly triggered when there is a change in value of an interested source. The CEP processing phase is performed by a set of rules while in RLs the user defines how the output will be defined based on the input values. The propagation phase of CEP and RLs only defer on how they are delivered. Finally, the CEP reaction phase generally acts as

a message informing that the interested event happened, whereas the RL reaction phase returns the value that changed.

To summarize, the similarities revealed in [1] show they can be complementary solutions to support reactive applications. For example, on the observation phase, a RL can collect the value changes of a source A, transform A into an event, then filter the event stream based on some predicate function over A, then apply a CEP operator extended from the RL function manipulations and finally the RL can be responsible for propagating the changes accordingly to the passed reaction function. In fact, the REScala and CEPSwift projects were able to implement the described scenario by integrating CEP and RLs features.

2.4. CEP Operators

Complex event processing operators provide ways of defining rules and also allow selecting, transforming and managing flows of information of a CEP engine. Cugola and Margara listed all the available operators collected during their analysis (see Table 1). Some of these operators are limited to a particular CEP framework model, while others are common to a set of them. Moreover, there is usually the possibility of combining multiple operators in order to achieve the desired behavior or rule.

Table 1 - Available Operators. Summarized from Cugola and Margara, 2012 [3]

Operator type	Operator class	Description
Single-Item Operators	Selection operator	A selection operator can filter items that satisfies a given constraint
	Elaboration operator	An elaboration operator can transform items by selecting or changing their information
Logical Operators	Conjunction	A conjunction of items $I_1, I_2...I_n$ is satisfied when all the items $I_1, I_2...I_n$ have been detected
	Disjunction	A disjunction of items $I_1, I_2...I_n$ is satisfied when at least one of the items $I_1, I_2...I_n$ have been detected
	Repetition	A repetition of an information item I of degree $<m, n>$ is satisfied when I is detected at least m times and not more than n times

	Negation	A negation of an item I is satisfied when I is not detected.
Sequences	Sequence operator	A sequence defines an ordered set of information items $I_1, I_2 \ldots I_n$, which is satisfied when all the elements $I_1, I_2 \ldots I_n$ have been detected in the specified order
Windows	Fixed windows	A fixed window does not move and process items within a given time frame
	Landmark windows	A landmark window has a fixed lower bound, while the upper bound advances every time a new item arrives
	Sliding windows	A sliding window has only a fixed size, i.e., both upper and lower bounds advance when new items arrive
	Pane and tube windows	A pane and tumble window is a variant of sliding windows, in which both the lower and the upper bounds move by k elements, as k elements enter the system. The difference between pane and tumble windows is that the former have a size greater than k, while the latter have a size smaller than (or equal to) k
Flow Management Operators	Join operator	Join operators are used to merge two flows of information
	Union	Union merges two or more input flows of the same type, creating a new flow that includes all the items coming from them
	Except	Except takes two input flows of the same type and outputs all those items that belong to the first one but not to the second one. It is a blocking operator
	Intersect	Intersect takes two or more input flows and outputs only the items included in all of them. It is a blocking operator.
	Remove-duplicate	Remove-duplicate removes all duplicates from an input flow
	Duplicate	Duplicate operators allow a single flow to be duplicated in order to use it as an input for different processing chains

	Group by	Group by operators are used to split information flows into partitions in order to apply the same operator (usually an aggregate) to the different partitions
	Order by	Order-by operators are used to impose an ordering to the items of an input flow
	Flow creation	Flow creation operators generally provide a way of creating a new information flow from a set of items
Aggregates	Detection aggregates	Detection aggregates are those used during the evaluation of the condition part of a rule
	Production aggregates	Production aggregates are those used to compute the values of information items in the output flow

2.5. Kotlin

Kotlin is a modern open source programming language that offers interoperability with the well-known Java and its JVM technologies. It also has support for Javascript, Android and Native platforms. The Kotlin project was started by JetBrains in 2010, but the first official stable release was only published in February 2016 [8]. Since then, Kotlin has been gaining rapid adoption and the remarkable point of it can be considered when Android developers decided to support it as the first-class language of their platform in 2017 [10] since many developers were already using and recommending it to new projects.

Besides the multiplatform support, Kotlin especially offers an appealing bundle for modern JVM programmers because it has both functional and object-oriented enhanced features, which makes the learning curve smoother for Java users. Furthermore, Kotlin has other features such as Lambda Functions as first-class citizens, Null Safety, Extension Methods and Data Classes that helps producing less boilerplate, more expressive and concise code in comparison to Java.

In the context of reactive applications, functional capabilities can be helpful in a reactive development environment because usually a stream is composed by the input of another stream besides the original source, thus we often see chains of operations over inputs and outputs of streams to achieve the desired result.

3. Implementation

This chapter briefly presents the proposal, tools, libraries and the language used in the project implementation, then it introduces the main CEPlin classes and their features. The code is available on GitHub[1] and it is open source under MIT license. Both CEPlin and CEPSwift are under the RxCEP Organization Profile.

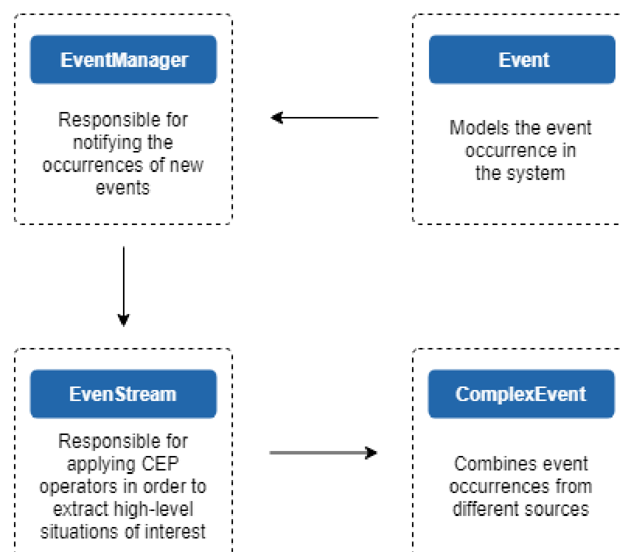
3.1. Proposal

This work introduces a new framework called CEPlin. The main goal of this work follows its precursor, called CEPSwift [2], which is making event and stream handling easier and also providing CEP features, with the exception of the CEPlin framework proposal is for Kotlin users. Additionally, CEPlin brings few implementation enhancements in comparison to CEPSwift, mostly due to Kotlin features while others due to simple code refactorings. CEPlin also offers a slightly different set of implemented operators listed on Table 2.

At the time of this work, there is no solution that contemplates CEP features for Kotlin. Furthermore, Kotlin has been gaining traction around Java, one of the most popular programming languages, and also Android developers due its full interoperability with them and seamlessly functional paradigm and extensibility integration, which makes the combination very suitable and powerful for programmers.

The framework architecture follows the same architecture proposed by Guedes (Figure 4), each entity will be further detailed in the next section.

Figure 4 - Framework high-level architecture (Adapted from Guedes, 2017, p. 25) [2]



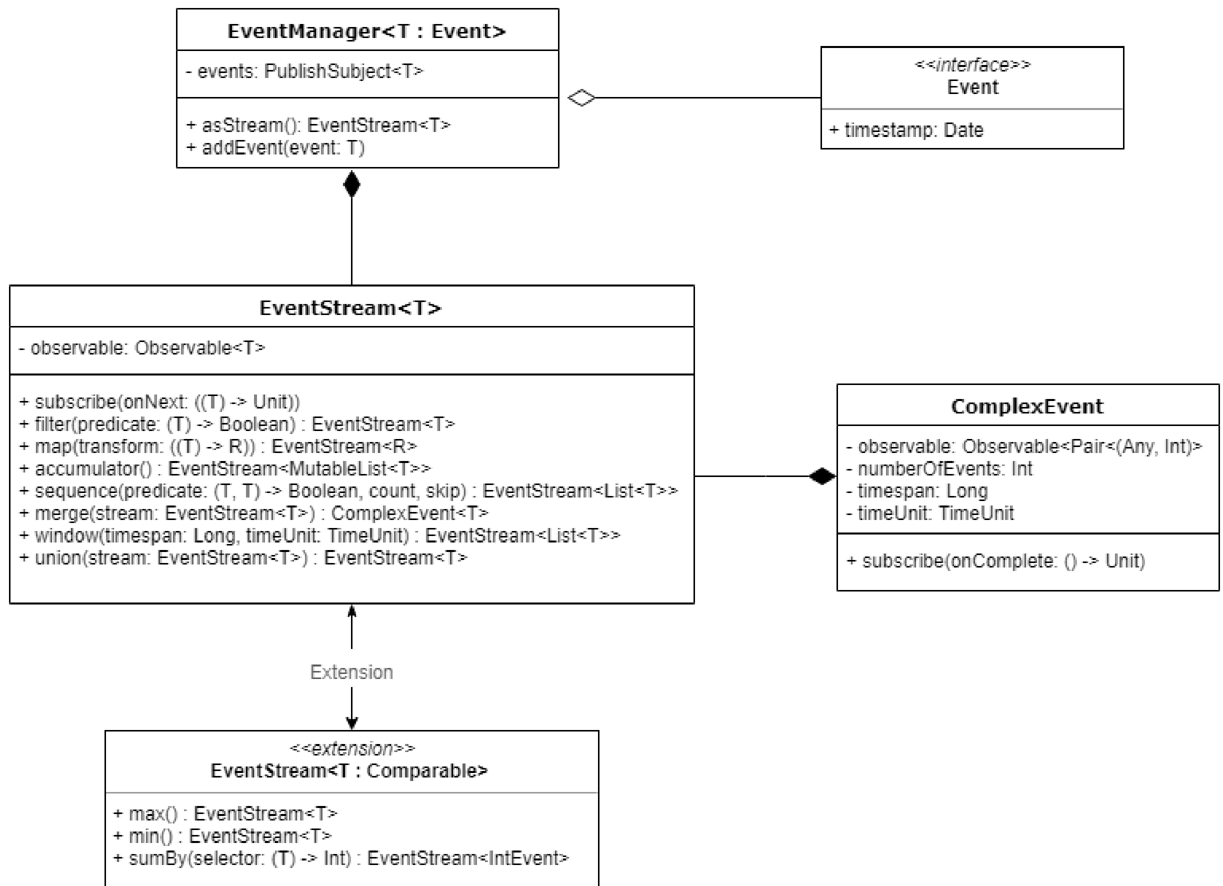
[1]Avaiable at <https://github.com/RxCEP/CEPlin>

3.2. Core entities

CEPlin has four main entities: Event, EventManager, EventStream and Complex Event. These entities are the core of the framework, thus it is important to know the role that each of them represents. They are summarized from Guedes [2] as follows:

- **Event:** the Event entity is an interface that every class that models a well-formatted event should conform to. Every Event subclass must have a timestamp attribute in order to keep the temporal aspect, e.g. a GPS signal can be mapped into a GPSEvent that has a latitude, longitude and a timestamp;
- **EventManager:** the EventManager<E> is a generic class that manages the occurrences of events, where E must be an Event class. This way, every time an Event E occurs, it should be added to the EventManager<E> by using the *addEvent* function (Figure 5). An user interested in those events must get an EventStream instance from the EventManager by calling the *asStream* function and subscribing to it;
- **EventStream:** the EvenStream is a class that represents the stream of all occurrences of an event. This class has all the CEP operators functions and each function may either return a new EventStream or a ComplexEvent (Figure 3). This is where the functional paradigm can be really helpful when managing the event stream because the user can easily compose a rule by filtering, merging and mapping each resulting EventStream and then finally subscribe by passing a function that will react to that information.
- **ComplexEvent:** the ComplexEvent class is required when the user is interested in merging different EventStreams that have different event types. A ComplexEvent instance is created in order to detect when the merged EventStreams emitted events happen within a time frame, the user must also pass a function that will react to that information

Figure 5 - CEPlin UML class diagram



3.3. Libraries and tools

CEPlin is entirely written in Kotlin. However, in order to add reactive capabilities to this work it was necessary to use RxJava and RxKotlin [5][6] libraries from the ReactiveX (Rx) project. ReactiveX is a library that extends the Observer pattern to support asynchronous data streams [7]. RxJava is the core reactive library for JVM and provides all the Rx features required for it, while the RxKotlin adds Kotlin-only features, such as extension functions in addition to RxJava. At the time of this work, the CEPlin library is available as a module inside of an Android project where the case studies are implemented using the framework.

3.4. Implemented operators

The implemented operators are a subset of possible operators defined by Cugola (see section 2.4). They were chosen by relevance, complexity and available development time. Some operators are a straightforward wrapped implementation from the ReactiveX library,

illustrated by the *Wrap* column, while others range from minor to complex compositions as described by the *Implementation* column (*Impl.,* for short) in the Table 2 below:

Table 2 - Implemented Operators

Operator type	Operator	Description	Wrap	Impl.
Single-Item Operators	Filter	The filter operator emits only events from an EventStream that satisfies a predicate function.	●	
	Map	This operator transforms an EventStream by creating a new EventStream through a projection function.	●	
Sequences	Sequence	The sequence operator emits only events that follows a specified order within a set of events. The operator takes a predicate function as the sequence condition and the length of the sequence to be considered.		●
Windows	Window	This operator only emits events that happened within a given time frame.	●	
Flow Management Operators	Merge	This operator merges two EventStreams and notifies the subscriber through a ComplexEvent object when both EventStreams happen within a given time frame.		●
	Union	This operator merge two EventStreams into one EventStream that emits events from both streams as they arrive.	●	
Aggregates	Min and Max	In order to use the aggregates functions the event must implement the Comparable interface. The operators emit the maximum and minimum, respectively.		●
	SumBy	This operator sums the values emitted by an EventStream. The sum operator takes a selector function that should return the data to be added.		●

4. Use cases

This section shows simple cases using CEPlin framework. The first case aims to detect an horizontal gesture sequence on the phone screen and the other will detect the action of putting the phone in the pocket. For each case a table with metrics highlights the differences of implementing by just using ReactiveX versus CEPlin library.

4.1. Gesture sequence detection

The goal for this use case is to be able to set rules for a specific sequence of gestures in order to react to them as soon as they happen. The gestures are captured from the screen touch event function called *onTouchEvent*, which returns a MotionEvent with the x and y axis values.

The case is very simple: the application wants to know if the user is moving their finger along the x axis with a certain tolerance of a vertical displacement. Each captured movement is modeled to a class called TouchEvent.

Figure 6 - TouchEvent class that models the user screen touches

```
class TouchEvent(val x: Float, val y: Float) : Event {
    override val timeStamp = Date()
}
```

Now, the application must create an EventManager instance in order to add TouchEvents to the stream as the touch movement changes:

Figure 7 - EventManager instance for TouchEvent

```
private val gestureManager = EventManager<TouchEvent>()

override fun onTouchEvent(event: MotionEvent): Boolean {
    val action = event.actionMasked
    when (action) {
        MotionEvent.ACTION_MOVE -> {
            gestureManager.addEvent(TouchEvent(event.x, event.y))
        }
    }
    return true
}
```

Finally, the application can define the rule that will detect this particular gesture movement. For this case we'll consider it valid by checking if the movements within a window of K events follows a sequence where each consecutive x-axis distance difference

must be greater than a threshold *X*, while the y-axis movements must be lower than a threshold *Y*. The thresholds *X* and *Y* here are just distance in pixels, for simplicity.

Figure 8 - Horizontal gesture rule using the CEPlin sequence operator

```
private fun setHorizontalRule(){
    gestureManager.asStream()
        .sequence({ a, b ->
            abs( x: a.x - b.x) > X &&
            abs( x: a.y - b.y) < Y }, count: 5)
        .subscribe { it: List<TouchEvent>
            log( s: "Horizontal gesture")
        }
}
```

For a matter of comparison the following the Figure 9 shows how to achieve the same behaviour by only using the Rx library. Then the Table 3 presents the metric reports for each way of defining the same rule. As we can see the CEPlin reduces the lines of code (LOC) for the sequence rule by a factor of almost 3 in comparison to the Rx only code. Furthermore, it also reduced the numbers of operator calls which produces a more readable code, this is naturally due to the abstraction that the CEPlin proposes.

Figure 9 - Horizontal gesture rule using ReactiveX operators

```
private fun setHorizontalRuleRxOnly(count: Int = 5, skip: Int = count) {
    val sequenceEquals = this.publishSubject
        .buffer(count, skip)
        .filter { it: (Mutable)List<TouchEvent!>
            var filter = true
            if (it.isNotEmpty() && count > 1) {
                for (i in 1..(it.size - 1)) {
                    if (!(abs( x: it[i - 1].x - it[i].x) > X
                            && abs( x: it[i - 1].y - it[i].y) < Y)) {
                        filter = false
                        break
                    }
                }
            }
            ^filter filter
        }
    sequenceEquals.subscribe { it: (Mutable)List<TouchEvent!>!
        log( s: "Horizontal gesture")
    }
}
```

Table 3 - Metrics for the horizontal gesture case

Library	LOC	Op. Calls
RxJava/Kotlin	18	2
CEPlin	7	1

4.2. Phone in pocket action detection

In this case we want to detect if the user just put the phone in the pocket. The goal here is to cover one of the possible ways of detecting the action of putting the phone in the pocket. The covered scenario is when the user is holding the phone and then they point it downwards in the direction of their pant pocket. Evidently, we cannot rely only on the orientation of the phone, otherwise one could just turn the phone downwards, thus we must track other sensors in order to detect the described behaviour. The solution was to track the values of the downwards acceleration and proximity sensors.

First, we must define the event classes that represents each sensor data (Figure 10 and Figure 11).

Figure 10 - ProximityEvent class that models the phone proximity to an object

```
class ProximityEvent(val dist: Float) : Event {
    override val timeStamp = Date()
}
```

Figure 11 - AccelerationEvent class that models the phone proximity to an object

```
class AccelerationEvent(val x: Float, val y: Float, val z: Float) : Event {
    override val timeStamp = Date()
}
```

Then two EventManager instances must be created for each interested type of event. All sensor data are retrieved from the native Android Sensor Library that calls the *onSensorChanged* function with the state of the sensor over time (Figure 12).

Figure 12 - EventManager instances for ProximityEvent and AccelerationEvent

```
private val accManager = EventManager<AccelerationEvent>()
private val proxManager = EventManager<ProximityEvent>()

override fun onSensorChanged(sensorEvent: SensorEvent) {
    when (sensorEvent.sensor.type) {
        Sensor.TYPE_ACCELEROMETER -> {
            val x = sensorEvent.values[0]
            val y = sensorEvent.values[1]
            val z = sensorEvent.values[2]
            accManager.addEvent(AccelerationEvent(x, y, z))
        }
        Sensor.TYPE_PROXIMITY -> {
            val prox = sensorEvent.values[0]
            proxManager.addEvent(ProximityEvent(prox))
        }
    }
}
```

Finally, the application declares the two rules to be considered that will validate the in pocket action (Figure 13). First, the acceleration rule is responsible for detecting when the user is moving the phone downwards by using the sequence operator of length 2 and passing the predicate function that checks if the y-axis acceleration is going downwards. Then the proximity rule checks if the distance to an object is decreasing by also using the sequence operator and passing a function that evaluates two consecutives values. At last, we merge the two rules by applying the merge function, i.e., whenever the acceleration rule and the proximity rule are triggered within a time frame the complex event will notify its subscribers.

Figure 13 - Phone in pocket action rule using CEPlin operators

```
private fun setInPocketRule() {
    val accelerationRule = accManager.asStream().sequence({ a, b -> a.y > b.y && b.y < -5 }, count: 2, skip: 1)

    val proximityRule = proxManager.asStream().sequence({ a, b -> a.dist > b.dist && b.dist < 4 }, count: 2, skip: 1)

    accelerationRule.merge(proximityRule).subscribe { log( s: "IN POCKET!") }
}
```

Figure 14 shows how to accomplish the same application by only using RxJava/Kotlin and the Table 4 highlights the LOC difference between the two solutions. The results are very similar to the previous case, besides the RxJava/Kotlin LOC is now 10 times greater than the CEPlin code due to the more complex scenario.

Figure 14 - Phone in pocket action rule using ReactiveX operators

```
private fun setInPocketRuleRxOnly() {
    val accelerationRule = getSequenceObservable(accelerationEvents, { a, b -> a.y > b.y && b.y < -5 },  count: 2,  skip: 1)
    val proximityRule = getSequenceObservable(proximityEvents, { a, b -> a.dist > b.dist && b.dist < 4 },  count: 2,  skip: 1)

    val mergedRules = Observable.merge(accelerationRule, proximityRule)
    mergedRules.buffer( timespan: 5, TimeUnit.SECONDS,  count: 2).subscribe { bundle ->

            val events = bundle.listIterator()
            val values = mutableSetOf<Int>()

            for (item in events) {
                when(item.getOrNull( index: 0)){
                    is AccelerationEvent ->{
                        values.add(1)
                    }
                    is ProximityEvent ->{
                        values.add(2)
                    }
                }
            }

            if (values.count() == 2) {
                log( s: "IN POCKET!")
            }
        }
}

private fun <T> getSequenceObservable(subject: PublishSubject<T>, predicate: (T, T) -> Boolean,
                                      count: Int = 5, skip: Int = count): Observable<MutableList<T>>? {
    return subject.buffer(count, skip)
        .filter {  it: (Mutable)List<T!>
            var filter = true
            if (it.isNotEmpty() && count > 1) {
                for (i in 1..(it.size - 1)) {
                    if (!predicate(it[i - 1], it[i])) {
                        filter = false
                        break
                    }
                }
            }
            ^filter filter
        }
}
```

Table 4 - Metrics for the phone in pocket action case

Library	LOC	Op. Calls
RxJava/Kotlin	33	4
CEPlin	3	3

4.3. Discussion

This chapter introduced two use cases using the CEPlin library. Both used a small set of operators proposed in this work that aims to improve the process of defining complex rules. As this is the first version of the library, the range of use case possibilities might be limited

by the operators implemented, thus the examples were focused on the usage of operators not directly available by the Rx libraries.

Furthermore, as previously shown in Table 3 and 4, CEPlin not only significantly reduced the LOC amount to build the same solution by using pure Rx, but the API abstraction also improved the readability and the ability to define the rules more intuitively, this can be verified by the amount of operators called (Tables 3 and 4, *Op. calls* column) needed to achieve the same results. Therefore the end user can focus on implementing the business logic and the complex rules to detect the desired events rather than wasting time setting up their own solution.

5. Conclusion

In this work, a new CEP engine written in Kotlin called CEPlin was presented, the library extends reactive programming operators to help building CEP applications thanks to researches that triggered the integration between RLs and CEP. First, the CEP and RLs concepts were introduced as well as their integration scenario highlighting their similarities. After the implementation details, two use cases showed that CEPlin can improve the development of a CEP application by using reactive programming.

It is important to notice that this work was inspired by the CEPSwift library and one of the goals was also to make a version for Kotlin. Swift and Kotlin are adopted by different platforms and communities so it makes sense to cover the availability for Kotlin and potentially JVM users. Although they are similar approaches, they naturally have their own syntax and performances particularities, not to mention that their library and community ecosystem vary accordingly.

This work is the first step to support CEP using reactive programming in Kotlin, there are many other CEP operators to be developed in the future to increase the coverage of CEP requirements. There is also the need for performance and usability tests to improve the framework to match the real-time and the ease of use requirements, respectively. Needless to say, in the early stages of the library it is also necessary to use other design metrics to identify possible flaws to avoid technical debts and improve the long term maintainability of the code.

6. References

[1] MARGARA, Alessandro; SALVANESCHI, Guido. Ways to react: Comparing reactive languages and complex event processing. REM, 2013.

[2] GUEDES, George. CEPSwift: Complex Event Processing Framework for Swift (Undergraduate Thesis, Center of Informatics, UFPE). 2017

[3] CUGOLA, Gianpaolo; MARGARA, Alessandro. Processing flows of information: From data stream to complex event processing. ACM Computing Surveys (CSUR), Volume 44, Issue 3, June 2012, Pages 1-62

[4] SALVANESCHI, Guido; HINTZ, Gerold; MEZINI, Mira. REScala: Bridging Between Object-oriented and Functional Style in Reactive Applications. MODULARITY 2014-Proceedings of the 13th International Conference on Modularity (Formerly AOSD), Pages 25-36.

[5] RxJava: Reactive Extensions for the JVM. Retrieved from https://github.com/ReactiveX/RxJava.

[6] RxKotlin, Kotlin Extensions for RxJava. Retrieved from https://github.com/ReactiveX/RxKotlin

[7] ReactiveX - An API for asynchronous programming with observable streams. Retrieved from http://reactivex.io/

[8] Reference Documentation - Kotlin Programming Language. Retrieved from http://kotlinlang.org/docs/reference/

[9] P. Hudak. Functional reactive programming. In Programming Languages and Systems, pages 1–1. Springer, 1999.

[10] Android Developers Blog: Android Announces Support for Kotlin. Retrieved from https://android-developers.googleblog.com/2017/05/android-announces-support-for-kotlin.html