Universidade Federal de Pernambuco
Centro de Informática

Graduação em Ciência da Computação

Extensão e Análise de Performance da Biblioteca CEPlin

Dicksson Rammon Oliveira de Almeida

Trabalho de Graduação

Recife
13 de dezembro de 2018

Universidade Federal de Pernambuco
Centro de Informática

Dicksson Rammon Oliveira de Almeida

Extensão e Análise de Performance da Biblioteca CEPlin

Trabalho apresentado ao Programa de Graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco como requisito parcial para obtenção do grau de Bacharel em Ciência da Computação.

Orientador: *Kiev Santos da Gama*

Recife
13 de dezembro de 2018

To my mother, the greatest gift I will ever have.

Acknowledgements

It goes without saying that I am extremely grateful to all the people that have contributed in some way to my journey. To all my family, friends, and colleagues, I am grateful for your influence as well as your support.

I would also like to thank the professors and instructors of the Federal University of Pernambuco, both from the Center of Informatics as well as outside it, for their years of knowledge, experience, and dedication.

Finally, yet importantly, I would like to thank Prof. Dr. Kiev Gama for his guidance and support, without which this work could not have been completed. I look forward to the growth of the RxCEP project, and hope that more people can get involved in it.

I am what time, circumstance, history, have made of me, certainly,
but I am also, much more than that.
So are we all.

—JAMES BALDWIN (preface from *Notes of a Native Son*, 1984)

Resumo

Processamento de eventos complexos (CEP) é uma área da ciência da computação que estuda métodos de rastreamento e análise de fluxos de informação que representam eventos, a fim de derivar informações novas a partir de padrões complexos dentre eles. Em paralelo, linguagens reativas (RLs), oriundas do paradigma de programação reativo, fornecem meios para a detectar mudanças em valores, e assim aplicar operações como seleção, transformação e agregação sobre fluxos de dados. Este trabalho dá continuação à biblioteca CEPlin, parte do projeto RxCEP, que visa oferecer uma ferramenta de processamento de eventos complexos sobre uma camada reativa. Além de acrescentar novos operadores de cunho matemático, este trabalho também propõe uma análise de performance que compara a eficiência da biblioteca com uma implementação puramente reativa. Os resultados demonstram que a perda de performance ao usar o CEPlin é bastante pequena em contraste com uma implementação sem a biblioteca, estabelecendo a biblioteca como uma escolha viável para uso em aplicações reais.

Palavras-chave: Processamento de Eventos Complexos, Linguagens Reativas, ReactiveX, Kotlin, Android, Performance.

Abstract

Complex Event Processing (CEP) is a computer science field that studies methods for tracking and analysis of information flows that represent events, in order to derive new information from complex patters within these events. Parallel to this is another concept named reactive languages (RLs), which has been originated from the reactive programming paradigm; such languages provide means to detect value changes, and thus apply operations such as selecting, filtering and aggregation on data streams. This work intends to continue improving the CEPlin library, which is part of RxCEP, a project that aims to provide a complex event processing framework implemented on a reactive layer. Besides including new mathematical operators, this work also proposes a performance analysis framework that compares the library against a purely reactive implementation. The results demonstrate that the performance cost of using CEPlin is very similar to an implementation without using it, establishing the library as a viable choice in real-world applications.

Keywords: Complex Event Processing, Reactive Languages, ReactiveX, Kotlin, Android, Performance.

Contents

1 Introduction **1**

2 Concepts **3**
2.1 Complex Event Processing 3
2.2 Reactive Programming 4
2.3 RxCEP 5
2.4 Kotlin 6
2.5 Performance Evaluation 7

3 Proposal and Implementation **9**
3.1 Operators 9
3.1.1 Average 9
3.1.2 Probability 10
3.1.3 Expected value 12
3.1.4 Variance 13

4 Experiment **15**
4.1 Purpose of experiment 15
4.2 Application context 15
4.3 Performance metric 16
4.4 Workload characterization 17
4.4.1 Dataset 17
4.4.2 Load generator 18
4.4.3 CEP Rules 19
4.4.3.1 Up Trend Rule 20
4.4.3.2 Down Trend Rule 20
4.4.3.3 Engulfing Bullish Rule 21
4.4.3.4 Engulfing Bearish Rule 22
4.4.3.5 Down Trend Engulfing Bullish Event 22
4.4.3.6 Up Trend Engulfing Bearish Event 23
4.5 Monitor 23
4.6 Factors and Levels 24

5 Results **25**
5.1 Simulations 25
5.1.1 First configuration 25

5.1.2 Second configuration 26
5.1.3 Third configuration 26
5.1.4 Fourth configuration 27
5.1.5 Fifth configuration 27
5.2 Discussion 28
5.2.1 Comparison with CEPSwift 28

6 Conclusion 29
6.1 Limitations 29
6.2 Future work 29

List of Figures

2.1 CQL rule for joining two streams of information [3] 3
2.2 "Marble" diagram, visually representing observable streams [16] 5
2.3 High-level architecture of the RxCEP libraries [2] 7

4.1 Two patterns in candlestick chart visualization. (a) Bullish engulfing pattern. (b) Bearish engulfing pattern. 16

List of Tables

2.1 Comparison between CEP and RLs over the five phases of reactive systems [9] 6

4.1 Experiment factors and levels 24

5.1 Experiment results for the first simulation 25
5.2 Experiment results for the second simulation 26
5.3 Experiment results for the third simulation 26
5.4 Experiment results for the fourth simulation 27
5.5 Experiment results for the final simulation 27

CHAPTER 1
Introduction

Event Processing, part of a broader field known as **Information Flow Processing** [3], is a software systems approach based on events; these can be defined as any occurrence that has taken place within a system or domain at a point in time. This definition, albeit simple, is powerful in its flexibility and expressiveness, and there are a number of modern applications that have adopted an event-driven approach as their programming model. More specifically, event processing is mainly centered around the detection, transformation, and reporting of events in a system.

The ever-increasing amount of event-driven applications has made Event Processing both a widely adopted enterprise solution and a broadly discussed research domain. Of particular interest is the **Complex Event Processing (CEP)** model, a more advanced type of event processing that further abstracts streams of information to derive more meaningful events. A number of CEP systems have been created over the past two decades, from simple publish-subscribe systems to enterprise-grade streaming platforms, such as Esper and Apache Flink. However, many CEP systems are designed around domain-specific languages that are similar to database query languages, which may not be intuitive at first, adding an implementation overhead.

Parallel to CEP, another concept known as **reactive programming** has attracted similar interest; like CEP, it is concerned with data streams, but unlike CEP-oriented languages, it achieves its goal through a more declarative approach. **Reactive Languages (RLs)**, as they are most commonly called, are often built on top of existing languages (e.g. as libraries), having all the advantages of general-purpose languages with minimum overhead.

While CEP and RL systems share some similarities [9], there is little academic work overlapping the two. Recent research focusing on this relationship demonstrates the implementation of a basic CEP system with a reactive programming approach, released as two libraries - one written in Swift, the other in Kotlin. Known as CEPSwift [2] and CEPlin, respectively, both have implemented a small subset of CEP techniques, being built on top of libraries from the ReactiveX project [10]; therefore, there is still room for future work. This work focuses on expanding the CEPlin library, introducing new mathematical operators (*average*, *probability*, *expected value* and *variance*) and proposing a framework for performing a performance analysis of its execution.

This paper is organized as follows:

1. Section 2 explores the concepts from CEP, RLs, and the relationship between them;

2. Section 3 discusses the implementation of the mathematical operators;

3. Section 4 describes the framework for the performance analysis of the CEPlin library, compared with a purely reactive approach, by running a series of simulations, running under different parameters;
4. Section 5 discusses the results;
5. Section 6 concludes the paper, describing limitations and future work.

CHAPTER 2

Concepts

This chapter will introduce some of the concepts and topics that surround this work. A general overview will be presented, since previous work in the CEP libraries provide more detail.

2.1 Complex Event Processing

Complex Event Processing (CEP) is an information flow processing model that interprets information items as notifications of events, and relies on a number of techniques to abstract these events to ultimately derive information in a higher level. Such information, in turn, is represented as a complex event, which can either be reprocessed by the CEP engine to produce more information or be sent as a notification to interested parties.

CEP systems can be seen as an evolution to the *publish-subscribe* pattern, where senders of messages, or *publishers*, associate these messages to specific topics, where the receivers, or *subscribers*, only receive messages that are associated with the topic that they are interested in [3]. A CEP system extends this functionality further, by considering complex patterns within the information that enters the system.

Traditionally, CEP systems are described in terms of its *rule language*, a domain-specific language that expresses the rules of detecting and triggering events. CEP systems provide a set of *operators* for filtering, aggregation, and transformation of events. For example, the rule displayed in Figure 2.1 in the **Sase+** language [4] applies operators to detect food contamination in a supply chain - given a *contamination* alert, it iterates over *Shipment* information items to report on every possible series of contaminated shipments within a three-hour window:

```
PATTERN SEQ(Alert a, Shipment+ b[ ])
WHERE skip_till_any_match(a, b[ ]) {
a.type = 'contaminated' and
b[1].from = a.site and
b[i].from = b[i-1].to }
WITHIN 3 hours
```

Figure 2.1: CQL rule for joining two streams of information [3]

Cugola and Margara [3] provide a comprehensive list of operators, some of which are categorized as follows:

1. *Single-item operators*: these perform processing on information items one by one (e.g., operators that select items based on a given criteria, or apply a given transformation on each item);

2. *Logic operators*: used to combine the detection of information items, such as to consider items from more than one source;

3. *Sequence operators*: while similar to logic operators, sequence operators take into consideration the order of arrival of information items;

4. *Flow management operators*: these operators allow manipulation of flows of information, such as merging two flows, thus combining items from them;

5. *Aggregate operators*: these produce new information over an aggregation of information items, such as performing some kind of calculation from these items.

2.2 Reactive Programming

Reactive programming has quickly gained prominence as a paradigm that is well-suited for developing event-driven and interactive applications. It provides abstractions to manage event handling logic, and it achieves this by being built around the notion of continuous time-varying values and the propagation of change. In short, changes in a given state of the application are automatically communicated to computations that depend on this state, eliminating the need for programmers to coordinate interactions within the system.

Almost all *reactive languages* (RLs) have been created as an extension to general-purpose languages, and thus can easily integrated to existing applications. Of particular note in this article is a set of libraries collectively named **Reactive Extensions**, or **ReactiveX** [10]. ReactiveX libraries are available in numerous languages, and is widely used in reactive applications.

ReactiveX implements reactive programming with the notion of asynchronous *data streams*, which are sequences of ongoing events ordered in time - such as event buses or click events in a mobile application. It allows the creation, combination, filtering and transformation of streams; subscription to these streams provides the reaction mechanism to its changes. This subscription approach draws heavily from the *Observable* pattern, in which actors (the *observers*) are notified of any state changes by an object these actors subscribe to. Indeed, ReactiveX leverages the Observable pattern to act on its data streams; however, the main feature of its APIs is that it allows composability of data streams, making them more flexible and reusable.

Figure 2.2 is a graphical representation of a data stream, known as a *marble diagram*; these diagrams are widely used to illustrate streams of data. A marble diagram consists of a time sequence (represented as an horizontal line, read from left to right), while the marbles represent values that enter the stream. In this figure, the marble diagram illustrates a click event stream, where operators are applied to select only those clicks that have been performed in rapid succession. Subscribers to this stream can now react accordingly to multiple clicks.

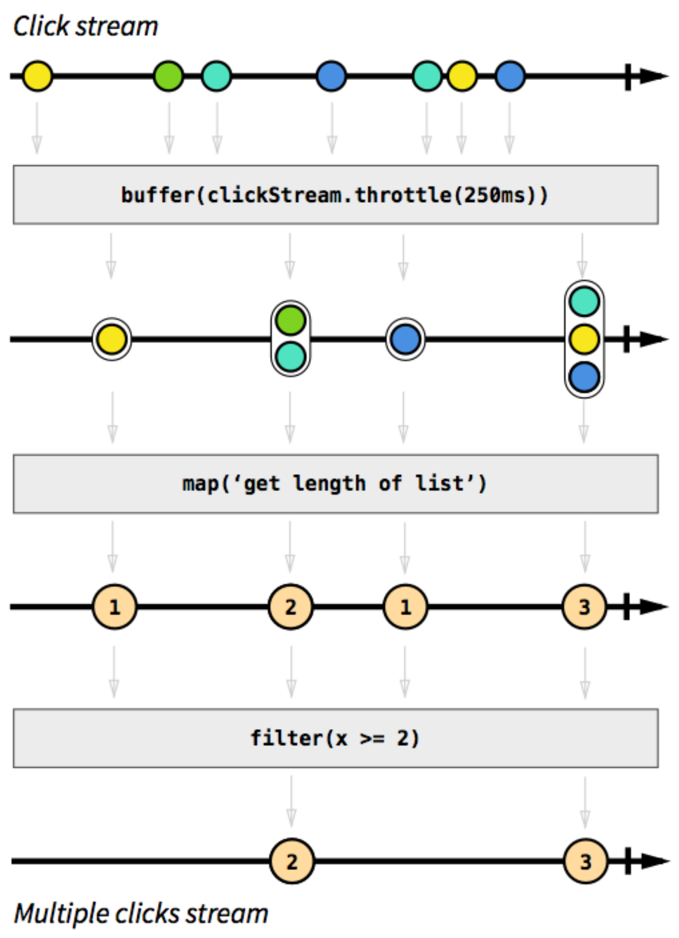
Click stream
buffer(clickStream.throttle(250ms))
map('get length of list')
1
2
1
3
filter(x >= 2)
2
3
Multiple clicks stream

Figure 2.2: "Marble" diagram, visually representing observable streams [16]

2.3 RxCEP

From a high level perspective, both concepts have similar aspects, most notably in the manner the reactive behavior is implemented (as seen in Table 2.1). As such, they are believed to be vastly complementary, although academic discussion between these two communities have seldom occurred, and an integration of these two fields has been lacking, until very recently. In 2017, the complementary nature of these fields has inspired work towards the creation of complex event programming libraries that leverage the reactive programming capabilities of the ReactiveX framework. These libraries are collectively named **RxCEP** [14], beginning with a focus on Web-based and mobile applications.

Indeed, an implementation of a CEP library using reactive languages presents many advantages for a number of reasons. First, the functional nature of reactive languages can be seen as a complement to the object-oriented nature of events - it is possible to represent an occurrence in time as an event object, while leveraging the functional aspect of reactive behavior to establish composition and detection of patterns that describe these events.

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 Cahnges

Table 2.1: Comparison between CEP and RLs over the five phases of reactive systems [9]

Additionally, the functional programming style present in reactive languages allows the programmer to express reactive behavior in a direct and declarative way [15]. This makes the program easier to read and understand, while the abstractions provided by the ReactiveX libraries handle changes in an application. This approach to event processing, although intuitive, has been underexplored; few libraries, such as REScala or Apache Fink have demonstrated this integration in a meaningful way.

As of this writing, three libraries are in development: **CEP.js** [11], **CEPSwift** [13] and **CEPlin** [12], written in JavaScript, Swift and Kotlin, respectively. Initially, the goal of RxCEP is to provide the complex event processing functionality to these languages; indeed, CEPSwift and CEPlin are the first known implementations of a CEP system model in the Swift and Kotlin languages. All three libraries use ReactiveX, as it provides both the reactive behavior and facilitates the implementation of CEP operators, some of which are very similar to the ReactiveX's own operators.

Figure 2.3 displays the basic architecture of the RxCEP libraries. The `EventStream` interface supplies an abstraction over ReactiveX's observable streams; additionally, the CEP operators can be applied to them, with the underlying implementation leveraging ReactiveX's own operators to achieve the desired results. In this work, the `EventStream` is the interface that is modified to accommodate the new operators, by way of extension functions implemented in it.

The `EventManager` interface allows users to define events and add them to a stream as they occur. Finally, a `ComplexEvent` is an entity that is created when event streams are combined.

2.4 Kotlin

Kotlin is a statically typed programming language created by JetBrains, first appearing in 2011 - but with a stable release ocurring only years later, in 2016. Since then, there has been a rapid adoption of the language; in the following year, Google announced first-class support for Kotlin on Android, allowing applications to be written entirely in Kotlin.

One of the main features is the language interoperability with Java, allowing it to run on a Java Virtual Machine. Furthermore, its syntax is oriented towards a functional programming style; consequently, it is considerably less verbose than Java, reducing the amount of boilerplate code. Other syntax features, such as extension methods (allowing users to add methods to

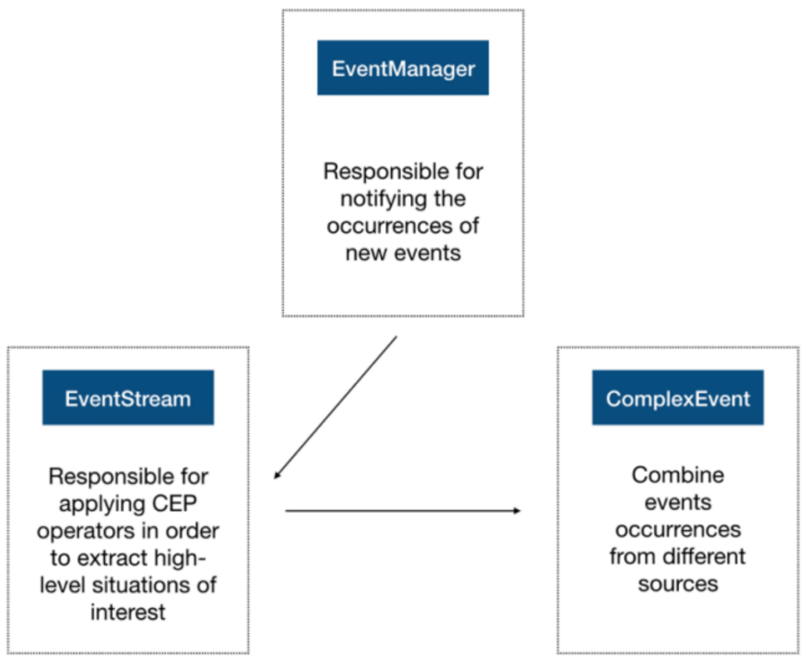
EventManager
Responsible for notifying the occurrences of new events
EventStream
Responsible for applying CEP operators in order to extract high-level situations of interest
ComplexEvent
Combine events occurrences from different sources

Figure 2.3: High-level architecture of the RxCEP libraries [2]

any class without the formalities of creating a derived class with new methods) and compiler features such as null safety, make Kotlin a more concise and safer language to use.

Kotlin is an adequate choice for the development of an RxCEP library, due to its functional style, improved readability, and smooth learning curve. In addition, and to the best of the author's knowledge, there is no known CEP library available in the Kotlin language.

2.5 Performance Evaluation

One of the goals of this work is to provide a performance evaluation of the CEPlin library, applying a small subset of the techniques introduced in Li's work on benchmarking CEP systems. In particular, we are interested in measuring the time that events are processed, from the time it enters the system to the time a new event is produced after a given operation. A comparison between a purely reactive approach and the proposed implementation will be performed; it is believed that the overhead introduced by the latter is small enough not to present a disadvantage over the former.

To this end, a framework will be implemented, consisting of three core components:

1. **Load generator**: This component is responsible for producing a workload to the target system, that is, a series of inputs that will be prepared for the system's processing layer, which will in turn will be measured. In this work, a load generator will implement a mechanism to trigger multiple events, spread evenly across multiple "sources", or threads.

2. **Simulator**: The simulator is responsible for the experiment setup and execution. It constructs the load generator, and contains logic that reproduces a given application use case. In this work, the simulator consists of an application that detects patterns in historical stock market information, using real-world data sets from multiple companies.

3. **Monitor**: The monitor is responsible for collecting information pertaining to a desired set of observations on a system. It is present in parts of the system to measure specific computations; therefore, in this work, it reports information from within both the load generator and the simulator components.

CHAPTER 3

Proposal and Implementation

RxCEP is an ongoing effort from academics at the Federal University of Pernambuco (UFPE), and there have been a number of theses focused on improving its libraries. In the same manner, one of the goals of this work is to further improve the CEPlin library, by expanding the amount of operators.

To that end, four operators have been implemented: *average*, *probability*, *expected* and *variance*. Each operator aggregates and emits an event containing the result of its respective mathematical function, by using some of ReactiveX's available operators. Furthermore, the *averageBuffer*, *probabilityBuffer*, *expectedBuffer* and *varianceBuffer* are versions of these operators that are applied on a list of events, in order to allow chaining with the *buffer* operator. The following section will provide more detail into how they were implemented.

The project was implemented in Android Studio 3, and executed on a Google Nexus 5X emulator, running API version 24 of the Android operating system (Android 7.0 Nougat).

3.1 Operators

To increase the scope of the CEPlin library, and consequently broaden the number of applications it can be used, *mathematical operators* have been added to the set of existing ones. These operators can be classified as *aggregates*, since they accumulate the events of a stream, and perform a computation over their values.

All operators were implemented as extension functions to the `EventStream` class. The following subsections describe their implementation.

3.1.1 Average

Given a stream of events, with each event containing a numeric value, the *average* operator accumulates these values in a sum, dividing that sum by the number of events, as represented by the mathematical expression of the arithmetic mean:

$$\frac{1}{n}\sum_{i=1}^{n} a_i = \frac{a_1 + a_2 + \cdots + a_n}{n} \tag{3.1}$$

The implementation is as follows. As a new event enters the stream, the method uses the *scan* operator to iterate through every event that has entered in the stream, accumulating the numeric value in the event in a sum as well as counting the number of events. The filter operator is used because, since the starting value is zero, a division by zero would cause the program to raise an exception. Finally, the map operator is used to compute the division of the sum-count value pair.

```
fun <T : Number>EventStream<out NumericEvent<T>>.average() :
  ↪ EventStream<NumericEvent<Double>> {
  val avg = this.observable
      .scan(Pair(0.0, 0),
            { acc, v ->
              // first: sum, second: count
              Pair(acc.first + v.value.toDouble(), acc.second
                ↪ + 1)
            })
      // prevent division by 0
      .filter { pair ->
        pair.second > 0
      }
      .map { pair ->
        NumericEvent(pair.first/pair.second)
      }

  return EventStream(avg)
}
```

The *averageBuffer* is a version of the operator that allows chaining with the *buffer* operator, taking a list of events and computing their average value.

```
fun <T : Number>EventStream<out
  ↪ List<NumericEvent<T>>>.averageBuffer() :
  ↪ EventStream<NumericEvent<Double>> {
  val avg = this.observable.map {
      val sum = it.sumByDouble {
        it.value.toDouble()
      }

      NumericEvent(sum/it.size.toDouble())
  }

  return EventStream(avg)
}
```

3.1.2 Probability

Given a stream of events, with each event containing a numeric value, the *probability* operator accumulates these events in a list, and calculates the probability of the latest event's value -

assuming a discrete probability distribution, that is, that the value of the event belongs to a discrete variable X, and

$$\sum_u \mathrm{P}(X = u) = 1 \tag{3.2}$$

as u runs through the set of all possible values of X.

The implementation is as follows. As a new event E enters the stream, the method uses the *scan* operator to iterate through every event that has entered in the stream, accumulating them in a list data structure. The filter operator is used to remove the starting value, which is an empty list. Finally, the map operator is used to compute the probability of E's numeric value, given all the preceding events in the sequence, using a helper function.

```
fun <T : Number>EventStream<out NumericEvent<T>>.probability()
  ↪  : EventStream<NumericEvent<Double>> {
  val prob = this.observable
      .scan(mutableListOf<NumericEvent<T>>(),
            { acc, ev ->
              acc.add(ev)
              acc
            })
      .filter { list -> list.size > 0 }
      .map { NumericEvent(prob(it, it.last().value)) }

    return EventStream(prob)
}

fun <T : Number> prob(list: List<NumericEvent<T>>, outcome: T)
  ↪  : Double {
    val occ = list.count { it.value == outcome }
    return occ/list.size.toDouble()
}
```

The *probabilityBuffer* is a version of the operator that allows chaining with the *buffer* operator, taking a list of events and computing their probability value.

```
fun <T : Number>EventStream<out
  ↪  List<NumericEvent<T>>>.probabilityBuffer() :
  ↪  EventStream<NumericEvent<Double>> {
  val prob = this.observable.map {
    NumericEvent(prob(it, it.last().value))
  }

  return EventStream(prob)
}
```

3.1.3 Expected value

Given a stream of events, with each event containing a numeric value, the *expected* operator accumulates these values in a list, and calculates the expected value of the random variable X, which represents the set of numeric values in the event stream, as described by the mathematical expression

$$\mathrm{E}[X] = \sum_{i=1}^{k} x_i p_i = x_1 p_1 + x_2 p_2 + \cdots + x_k p_k \tag{3.3}$$

where $x_1, x_2, \cdots, x_k$ are the possible outcomes of X, occuring with probabilities $p_1, p_2, \cdots, p_k$, respectively.

The implementation is as follows. As a new event enters the stream, the method uses the *scan* operator to iterate through every event that has entered in the stream, accumulating them in a list data structure. The filter operator is used to remove the starting value, which is an empty list. Finally, the map operator is used to compute the expected value based on the values of all the events in the stream, using a helper function.

```
fun <T : Number>EventStream<out NumericEvent<T>>.expected() :
  ↪ EventStream<NumericEvent<Double>> {
  val exp = this.observable
      .scan(mutableListOf<NumericEvent<T>>(),
            { acc, ev ->
              acc.add(ev)
              acc
            })
      .filter { list -> list.size > 0}
      .map { list -> NumericEvent(computeExpectedValue(list))
        ↪ }

    return EventStream(exp)
}

fun <T : Number> computeExpectedValue(list:
  ↪ List<NumericEvent<T>>) : Double {
  val probabilityList = mutableListOf<Double>()
  val outcomes = list.distinct()

  outcomes.mapTo(probabilityList) {
    it.value.toDouble() * prob(list, it.value)
  }

  return probabilityList.sum()
}
```

The *expectedBuffer* is a version of the operator that allows chaining with the *buffer* operator, taking a list of events and computing their expected value.

```
fun <T : Number>EventStream<out
  ↪ List<NumericEvent<T>>>.expectedBuffer() :
  ↪ EventStream<NumericEvent<Double>> {
  val exp = this.observable.map {
    NumericEvent(computeExpectedValue(it))
  }

  return EventStream(exp)
}
```

3.1.4 Variance

Given a stream of events, with each event containing a numeric value, the *variance* operator accumulates these values in a list, and calculates the variance of the random variable X, which represents the set of numeric values in the event stream, as described by the mathematical expression

$$\mathrm{Var}(X) = \sum_{i=1}^{n} p_i \cdot (x_i - \mu)^2 \tag{3.4}$$

The implementation is as follows. As a new event enters the stream, the method uses the *scan* operator to iterate through every event that has entered in the stream, accumulating them in a list data structure. The filter operator is used to remove the starting value, which is an empty list. Finally, the map operator is used to compute the variance based on the values of all the events in the stream, using a helper function.

```
fun <T : Number>EventStream<out NumericEvent<T>>.variance() :
  ↪ EventStream<NumericEvent<Double>> {
  val variance = this.observable
      .scan(mutableListOf<NumericEvent<T>>(),
            { acc, ev ->
              acc.add(ev)
              acc
            })
      .filter { list -> list.size > 0 }
      .map { list ->
        NumericEvent(computeVariance(list))
      }

    return EventStream(variance)
}
```

```
private fun <T : Number> computeVariance(list:
  ↪ List<NumericEvent<T>>) : Double {
  val n = list.size
  var sum = 0.0
  var sumSq = 0.0
  var x: Double

  for (ev in list) {
    x = ev.value.toDouble()
    sum += x
    sumSq += x * x
  }

  return (sumSq - (sum * sum)/n)/n
}
```

The *varianceBuffer* is a version of the operator that allows chaining with the *buffer* operator, taking a list of events and computing their variance.

```
fun <T : Number>EventStream<out
  ↪ List<NumericEvent<T>>>.varianceBuffer() :
  ↪ EventStream<NumericEvent<Double>> {
  val variance = this.observable.map
    NumericEvent(computeVariance(it))
  }

  return EventStream(variance)
}
```

CHAPTER 4

Experiment

This chapter describes the experiment performed on the CEPlin library, which aims to compare the framework's event processing performance against a purely reactive approach. The following sections outline the steps that compose the experiment, based on Jain's framework for performance evaluation [6].

4.1 Purpose of experiment

The goal of this experiment is to analyze the runtime performance of the complex event processing capabilities of the CEPlin library in an Android application, and compare it with the performance of a purely reactive implementation, by running a simulation on both approaches. The simulation generates a continuous flow of information, which will then be processed by each version, and a report is presented describing the results.

It is believed that the computational overhead introduced by the CEPlin library is small enough not to present a major disadvantage over the former, supporting the adoption of the framework for its succinctness and ease of use.

4.2 Application context

The chosen setting for the experiment is a simulation based on real historical stock market data from different companies. The scenario implemented by the simulation consists of observing patterns within the time series information to detect trend reversal signals, that is, when a market experiences a change in direction of its prices (e.g. prices increase during a decreasing trend, and vice versa). Based on this, two patterns have been selected:

1. **Price trend direction:** Refers to stock prices gradually increasing or decreasing over time. In financial analysis, a *moving average* is a good indicator for identifying this pattern. The simulation applies a **simple moving average** (**SMA**), which is a calculation that computes a series of averages of different subsets of the data. A trend direction can then be determined if the averages are gradually increasing or decreasing.

2. **Candlestick chart pattern:** *Candlestick charts* are a type of chart that displays information of stock prices over a period of time. Each "candlestick" contains the opening and closing price of a stock, along with the highest and lowest prices of that time interval, which is typically a day. In this chart, there are a number of patterns that can be visually

identified; these represent a movement in prices that can predict a particular market sentiment. A candle with a hollow body (known as a *bullish candle*) signifies that the stock price closed higher than its opening value; a candle with filled body (known as a *bearish candle*) signifies the opposite. An engulfing candle pattern occurs when a hollow candle has a larger body than a previous filled one (and vice-versa), indicating a market reversal during a price trend. The *engulfing candle pattern* has been chosen for the simulation; this pattern occurs when a hollow candle has a larger body than a previous filled one (and vice-versa). Figure 4.1 illustrates this pattern.

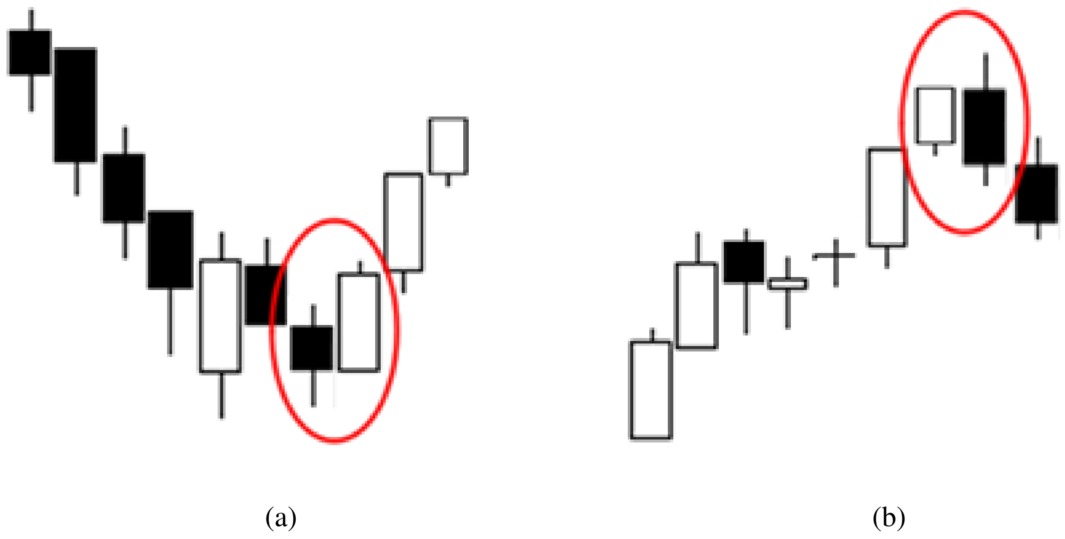

Figure 4.1: Two patterns in candlestick chart visualization. (a) Bullish engulfing pattern. (b) Bearish engulfing pattern.

This scenario allows the usage of one of the operators implemented in this work, as well as the definition of complex events to detect the patterns described above.

4.3 Performance metric

One performance metric was chosen to evaluate the event processing implementations: **response time**, also called **latency time** or **turnaround time**. According to [8], response time refers to the system's elapsed time from the point that a request is made by a user or an application to the response is returned to the user or the application. Li defines the response time metric in a CEP engine by using the following expression:

$$T_{response} = T_{out} - T_{n-in} \tag{4.1}$$

for a set of events $E_1, E_2, \cdots, E_n$ that enter the system in time series ($T_{n-in} > \cdots > T_{2-in} > T_{1-in}$), while T_{out} is the time a new event is triggered as a result of the transformation that occurs in the system. In the context of this work, for example, an *average* operator applied over the last ten events $(E_1, E_2, \cdots, E_{10})$ will produce an event E_{out} with a response time represented by the expression

$$T_{response} = T_{out} - T_1 \tag{4.2}$$

4.4 Workload characterization

According to [Jai], the workload can be real or synthetic. Because real workloads are generally not suitable for being used as test workloads, a synthetic workload was implemented, one which uses a dataset of historical stock market data retrieved from a Web-based application programming interface (API). This section describes the dataset, as well as the workload generator and the CEP rules that were defined during the event processing phase.

4.4.1 Dataset

As mentioned before, the dataset consists of real-world historical stock market data, which is represented as a time series spanning several years. Each company is comprised of a list of data points; each point contains stock information of a time period.

The dataset was retrieved from the AplhaVantage API, which provides various formats of historical and real-time data. The format chosen for this experiment is the *daily time series*, where each data point represents a day. The first few lines of the dataset are shown below:

```json
{
 "Meta Data": {
  "1. Information": "Daily Prices (open, high, low, close) and
   ↪ Volumes",
  "2. Symbol": "FB",
  "3. Last Refreshed": "2018-11-21",
  "4. Output Size": "Full size",
  "5. Time Zone": "US/Eastern"
 },
 "Time Series (Daily)": {
  "2018-11-21": {
   "1. open": "134.4000",
   "2. high": "137.1900",
   "3. low": "134.1300",
   "4. close": "134.8200",
   "5. volume": "25469735"
  },
  ...
 }
}
```

The data was modified to some degree; the only relevant information is the company's symbol and the time series itself. Furthermore, the time series was modified to list the data points from the earliest date, since the simulation will generate events in a sequential order. Lastly, the key-object format of a data point was modified, turning into a sole object, with the key now acting as a value. The modified dataset is as follows:

```
{
 "symbol": "FB",
 "time_series": [
  {
   "date": "2012-05-18",
   "open": "42.0500",
   "high": "45.0000",
   "low": "38.0000",
   "close": "38.2318",
   "volume": "573576400"
  },
  ...
 ]
}
```

The `date` field refers to the day in which the stock prices were recorded. The `open` and `close` fields refer to the opening and closing prices of the stock for that day, and the `high` and `low` fields denote the highest and lowest traded prices of the stock during the associated day.

4.4.2 Load generator

From the dataset described above, a load generator was implemented, which produces a continuous flow of events containing the time series information. The `TimeSeriesSimulator` class is responsible for creating a given number of event emitters (named `EquitySimulators`); each emitter triggers events from the time series of a given company.

For each time series (stored as a JSON file in the Android application's raw resources directory), a `EquitySimulator` is created, which then reads the file, parsing the information into a list data structure, which will be used to generate events when the simulation is started.

When the simulation starts, each `EquitySimulator` object creates a timer, which is a thread that executes a task after a given delay, and repeats that task for a given period of time. The timer's task will produce an event containing information from the first element of the data structure, repeat the process for the next element, and so on, until all the elements have been processed. To coordinate task execution across timers, a parameter controls the interval between each task, regulating the event triggering process, and can be adjusted to run simulations at a higher or lower event throughput.

To demonstrate in an example, if there are n `EquitySimulators`, each with a timer labeled $t_1, t_2, \cdots, t_n$ and the interval parameter is set to s seconds, all timers have their periods set to $s*n$ seconds, and each timer is delayed to $0*s, 1*s, \cdots, (n-1)*s$ seconds. Thus, timer t_1 starts immediately, timer t_2 starts after s seconds, timer t_3 starts after $2s$ seconds, and so on, ultimately interpolating n events over $s*n$ seconds.

4.4.3 CEP Rules

From the patterns outlined in section 4.2, six CEP rules have been defined - two referring to the *price trend direction* pattern, two more referring to the *candlestick chart* pattern, and the final two referring to combinations of these patterns. Each rule has a version that uses operators from the CEPlin library, while the other uses operators from the ReactiveX library, along with helper logic that mimics CEP behavior.

Indeed, the CEP versions are easier to understand, and require less lines of code. The `getSequenceObservable` function, seen on the ReactiveX versions in the following subsections, is a function that detects a sequence of values, according to a predicate that is passed to it:

```
private fun <T> getSequenceObservable(subject: Observable<T>,
  ↪ predicate: (T, T) -> Boolean, count: Int = 5, skip: Int =
  ↪ count): Observable<MutableList<T>>? {
  return subject.buffer(count, skip)
    .filter {
      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
    }
}
```

In the CEPlin version, the `sequence` operator is enough to achieve the same results:

```
val stream = eventManager.asStream()
      .sequence { a, b -> a == b }, 2, 1 }
      .subscribe {
        println("Two equal events detected.")
      }
```

The rules were implemented as follows:

4.4.3.1 Up Trend Rule

Rule that is satisfied when an SMA of a period is higher than the previous calculated SMA, which may indicate an increase in stock prices.

```
// CEPlin version
val upTrendRule = priceManager.asStream()
      // filter by company
      .filter { it.symbol == symbol }
      // calculate 10-day SMA
      .buffer(10, 1)
      .averageBuffer()
      // detect higher-than-previous SMAs
      .sequence({ a, b -> b.value > a.value }, 2, 1)

// ReactiveX version
val averagePrices = priceEvents
      // filter by company
      .filter { it.symbol == symbol }
      // calculate 10-day SMA
      .buffer(10, 1)
      .map {
        NumericEvent(it.sumByDouble
          ↪ {it.value}/it.size.toDouble())
      }

val upTrendRule =
      getSequenceObservable(averagePrices,
                            { a, b -> b.value > a.value },
                            2, 1)
```

4.4.3.2 Down Trend Rule

Rule that is satisfied when an SMA of a period is lower than the previous calculated SMA, which may indicate a decrease in stock prices.

```
// CEPlin version
val downTrendRule = priceManager.asStream()
      // filter by company
      .filter { it.symbol == symbol }
      // calculate 10-day SMA
      .buffer(10, 1)
      .averageBuffer()
      // detect lower-than-previous SMAs
      .sequence({ a, b -> b.value < a.value }, 2, 1)
```

```
// ReactiveX version
val averagePrices = priceEvents
      // filter by company
      .filter { it.symbol == symbol }
      // calculate 10-day SMA
      .buffer(10, 1)
      .map {
        NumericEvent(it.sumByDouble
          ↪ {it.value}/it.size.toDouble())
      }

val downTrendRule =
      getSequenceObservable(averagePrices,
                            { a, b -> b.value < a.value },
                            2, 1)
```

4.4.3.3 Engulfing Bullish Rule

Rule that specifies an engulfing bullish pattern, that is, when a bullish candle has a larger body than a previous bearish candle.

```
// CEPlin version
val engulfingBullishRule = priceManager.asStream()
      // filter by company
      .filter { it.symbol == symbol }
      // detect engulfing bullish pattern
      .sequence({ a, b ->
        a.close < a.open && b.open <= a.close && b.close >
          ↪ a.open
      }, 2, 1)

// ReactiveX version
val engulfingBullishRule =
      getSequenceObservable(
        // filter by company
        priceEvents.filter {
          it.symbol == symbol
        },
        // detect engulfing bullish pattern
        { a, b ->
          a.close < a.open && b.open <= a.close && b.close >
            ↪ a.open
        }, 2, 1)
```

4.4.3.4 Engulfing Bearish Rule

Rule that specifies an engulfing bearish pattern, that is, when a bearish candle has a larger body than a previous bullish candle.

```
// CEPlin version
val engulfingBearishRule = priceManager.asStream()
      // filter by company
      .filter {
        it.symbol == symbol
      }
      // detect engulfing bearish pattern
      .sequence({ a, b ->
        a.close > a.open && b.open >= a.close && b.close <
          ↪ a.open
      },
      2, 1)

// ReactiveX version
val engulfingBearishRule =
      getSequenceObservable(
        // filter by company
        priceEvents.filter {
          it.symbol == symbol
        },
        // detect engulfing bearish pattern
        { a, b ->
          a.close > a.open && b.open >= a.close && b.close <
            ↪ a.open
        },
        2, 1)
```

By combining these rules, two more complex patterns arise:

4.4.3.5 Down Trend Engulfing Bullish Event

When an engulfing bullish pattern occurs during a down trend, it is typically considered a reversal signal - that is, the market will possibly experience a positive turnaround.

```
// CEPlin version
engulfingBullishRule.merge(downTrendRule).subscribe {

  // complex event handling

}
```

```
// ReactiveX version
val mergedRules = Observable.merge(engulfingBullishRule,
  ↪ downTrendRule)
mergedRules.buffer(100, TimeUnit.MILLISECONDS, 2)
        .subscribe { bundle ->
          if (bundle.size == 2) {
            // complex event handling
          }
        }
```

4.4.3.6 Up Trend Engulfing Bearish Event

When an engulfing bearish pattern occurs during an up trend, it is typically considered a reversal signal - that is, the market will possibly experience a negative turnaround.

```
// CEPlin version
engulfingBearishRule.merge(upTrendRule).subscribe {
  // complex event handling
}

// ReactiveX version
val mergedRules = Observable.merge(engulfingBearishRule,
  ↪ upTrendRule)
mergedRules.buffer(100, TimeUnit.MILLISECONDS, 2)
        .subscribe { bundle ->
          if (bundle.size == 2) {
            // complex event handling
          }
        }
```

4.5 Monitor

The **monitor** is responsible for observing the simulation, storing information concerning the event processing library's activity. Events that are consumed and produced are the main concerns of the monitor; therefore, it is integrated in the load generator and the simulation itself, acting whenever an event is triggered.

A report is generated when the simulation is completed. In addition to the response time of the library, as previously stated, the monitor will also present the amount of events that were triggered during the simulation, separated by type, as well as the total execution time.

Factor	Levels
Number of companies (event emitters)	1, 3, 5
Dataset period (years)	1, 3, 5
Moving Average Period	10-day, 20-day
Interval between generated events (ms)	100, 250, 500

Table 4.1: Experiment factors and levels

4.6 Factors and Levels

This experiment aims to execute a series of simulations, and each simulation is determined by the values of its parameters, defined by [6] as *levels* and *factors*, respectively. Table 4.1 lists the factors and the levels that will be used for the experiment.

The number of companies represent the event emitters that will be created, producing events from a company's time series data. Real-world information from up to five companies (Apple, Amazon, Facebook, Google and Microsoft) will be used for the experiment.

The dataset period is a factor that indicates how much data from each company will be produced. Since each company contains years of data, each year comprises a subset, and can be increased by broadening the period.

The moving average period is the factor that affects the SMA calculation, which is based on a given window size (e.g., 10-day). A smaller window means that the trend direction will change frequently, and a larger window means the opposite. Thus, the amount of complex patterns detected may change according to the moving average period levels.

Lastly, the period between events represents the interval between each event produced by the load generator. As previously stated, this factor will affect the event throughput of the simulation.

CHAPTER 5

Results

This chapter presents the results of the experiment. Each simulation was executed with a different configuration, based on the levels described in the previous chapter; additionally, each simulation was executed under both CEP-oriented and purely reactive implementations. After each simulation, the monitor calculates the average response time of the detected complex patterns (down trend engulfing bullish and up trend engulfing bearish).

Finally, each simulation is executed ten times, in order to avoid variability between simulations. The repeated measures are then averaged and presented for each configuration.

5.1 Simulations

This section describes the simulations and their results.

5.1.1 First configuration

The first configuration uses one company (Google), for a dataset spanning a year of stock market data, a time period of 500 milliseconds between events, and a 10-day moving average window. The results are displayed in Table 5.1.

Implementation	Average Response Time (ms)
CEPlin Operators	4516.08
ReactiveX	4520.34
Execution time: 125551ms	
Equity Price Events: 251	
Down Trend Events: 92	
Up Trend Events: 149	
Engulfing Bearish Events: 8	
Engulfing Bullish Events: 9	
Down Trend Bullish Updates: 2	
Up Trend Bearish Updates: 3	
Total events: 515	

Table 5.1: Experiment results for the first simulation

5.1.2 Second configuration

The second configuration uses one company (Facebook), for a dataset spanning three years of stock market data, a time period of 250 milliseconds between events, and a 20-day moving average window. The results are displayed in Table 5.2.

Implementation	**Average Response Time (ms)**
CEPlin Operators	5325.53
ReactiveX	4870.22
Execution time: 190408ms	
Equity Price Events: 755	
Down Trend Events: 222	
Up Trend Events: 512	
Engulfing Bearish Events: 39	
Engulfing Bullish Events: 28	
Down Trend Bullish Updates: 5	
Up Trend Bearish Updates: 25	
Total events: 1587	

Table 5.2: Experiment results for the second simulation

5.1.3 Third configuration

The third configuration uses three companies (Apple, Amazon and Facebook), for a dataset spanning three years of stock market data, a time period of 250 milliseconds between events, and a 10-day moving average window. The results are displayed in Table 5.3.

Implementation	**Average Response Time (ms)**
CEPlin Operators	6769.80
ReactiveX	6875.32
Execution time: 567985ms	
Equity Price Events: 2265	
Down Trend Events: 851	
Up Trend Events: 1384	
Engulfing Bearish Events: 105	
Engulfing Bullish Events: 79	
Down Trend Bullish Updates: 22	
Up Trend Bearish Updates: 60	
Total events: 4769	

Table 5.3: Experiment results for the third simulation

5.1.4 Fourth configuration

The fourth configuration uses three companies (Facebook, Google and Microsoft), for a dataset spanning five years of stock market data, a time period of 100 milliseconds between events, and a 20-day moving average window.The results are displayed in Table 5.4.

Implementation	**Average Response Time (ms)**
CEPlin Operators	5745.25
ReactiveX	5741.77
Execution time: 379984ms	
Equity Price Events: 3315	
Down Trend Events: 1234	
Up Trend Events: 2018	
Engulfing Bearish Events: 132	
Engulfing Bullish Events: 125	
Down Trend Bullish Updates: 41	
Up Trend Bearish Updates: 77	
Total events: 6945	

Table 5.4: Experiment results for the fourth simulation

5.1.5 Fifth configuration

The final configuration uses five companies (Apple, Amazon, Facebook, Google and Microsoft), for a dataset spanning five years of stock market data, a time period of 100 milliseconds between events, and a 20-day moving average window.The results are displayed in Table 5.5.

Implementation	**Average Response Time (ms)**
CEPlin Operators	9540.00
ReactiveX	9540.49
Execution time: 631141ms	
Equity Price Events: 5829	
Down Trend Events: 2190	
Up Trend Events: 3536	
Engulfing Bearish Events: 244	
Engulfing Bullish Events: 202	
Down Trend Bullish Updates: 63	
Up Trend Bearish Updates: 143	
Total events: 12212	

Table 5.5: Experiment results for the final simulation

5.2 Discussion

The results produced by the simulations demonstrate that the implementation of rules using the CEPlin library incur little to no computational overhead when compared to the purely reactive implementation - with the former even slightly surpassing the response time of the latter in some simulations, as is the case for the first and third simulations, in tables 5.1 and 5.3, respectively. It can be safely assumed that, when building an event-driven application, the abstractions provided by the framework do not sacrifice efficiency, while at the same time reducing complexity and offering a larger set of features for event processing.

5.2.1 Comparison with CEPSwift

The performance analysis carried out for the CEPSwift library [7] provides a very different set of results, reporting a response time improvement of about 30 times over the purely reactive implementation's response time. However, it is unclear how the metric was calculated, and a similarity to the metric described here (by the equation 4.2) is difficult to determine.

The random nature of the simulation may also have an impact on the results reported. The load generator implemented may not be a suitable candidate for a test workload, since a given simulation for a version of the event processing logic cannot be reproduced for another version.

CHAPTER 6

Conclusion

This work proposed the extension of the CEPlin library by adding aggregate operators, increasing the scope of the framework, and by implementing a performance analysis framework for running simulations using real-world stock market data. It was determined that the computational overhead introduced by the the library is little to non-existing, establishing the framework as a good solution for event-driven applications without sacrificing efficiency. This is another step in expanding the potential of the RxCEP project, while at the same time reinforcing the need for improvement, in order for the project to become a standard CEP reference for general-purpose languages.

6.1 Limitations

As previously stated, there is room for further improvement in the CEPlin library, both within and outside the scope of this work. Some of the limitations that were identified include:

1. The operators implemented can only be applied to events that extend the `NumericEvent` class, and only execute on its numeric field. While the event is generic enough to allow any kind of number, it is still restrictive on the class type. Ideally, the operators should be used on any type of event, and perform calculations on a chosen numeric field; however, this may impact the performance of the library.

2. A broader analysis is possible, considering factors such as memory and CPU usage. Due to time constraints, a more complete evaluation was not feasible. The proposed experiment is suitable for further analysis, avoiding the need for a new one.

6.2 Future work

The CEPlin library is still in its early stages, and its further improvement is crucial for it to become a viable tool in Android applications. Some of the future work include:

1. A further expansion to include more, if not all, operators can evolve the library into a full product, fulfilling most of the requirements inherent to a CEP framework. Adding support for the different types of windows can be a good start, for instance.

2. A more sophisticated proof of concept can be very useful in identifying problems in the library, as well as maximize its potential in a real-world application. There is an

increasing number of opportunities for event-driven solutions in a mobile environment, and a CEP presence can increase it even further.

Bibliography

[1] Engineer Bainomugisha, Andoni Lombide Carreton, Tom van Cutsem, Stijn Mostinckx, and Wolfgang de Meuter. A survey on reactive programming. *ACM Comput. Surv.*, 45(4):52:1–52:34, August 2013.

[2] George Belo. Cepswift: Complex event processing framework for swift, 2017. (document), 1, 2.3

[3] Gianpaolo Cugola and Alessandro Margara. Processing flows of information: From data stream to complex event processing. *ACM Comput. Surv.*, 44(3):15:1–15:62, June 2012. (document), 1, 2.1, 2.1

[4] Yanlei Diao, Neil Immerman, and Daniel Gyllstrom. Sase+: An agile language for kleene closure over event streams, 2007. 2.1

[5] Opher Etzion and Peter Niblett. *Event Processing in Action*. Manning Publications Co., Greenwich, CT, USA, 1st edition, 2010.

[6] R. Jain. *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling*. May 1991. Winner of "1991 Best Advanced How-To Book, Systems" award from the Computer Press Association. 4, 4.6

[7] Hélmiton Júnior. Análise de performance de operadores no cepswift, 2018. 5.2.1

[8] Chunhui Li and Robert Berry. Cepben: A benchmark for complex event processing systems. In *Revised Selected Papers of the 5th TPC Technology Conference on Performance Characterization and Benchmarking - Volume 8391*, pages 125–142, Berlin, Heidelberg, 2014. Springer-Verlag. 4.3

[9] Alessandro Margara and Guido Salvaneschi. Ways to react: Comparing reactive languages and complex event processing. 2013. (document), 1, 2.1

[10] ReactiveX. Rxswift project. `https://github.com/ReactiveX/RxSwift`. Accessed: 2018-06-09. 1, 2.2

[11] RxCEP. Cep.js. `https://github.com/RxCEP/CEP.js`. Accessed: 2018-06-09. 2.3

[12] RxCEP. Ceplin. `https://github.com/CEPlin`. Accessed: 2018-06-09. 2.3

[13] RxCEP. Cepswift. https://github.com/RxCEP/CEPSwift. Accessed: 2018-06-09. 2.3

[14] RxCEP. Rxcep project. https://github.com/RxCEP. Accessed: 2018-06-09. 2.3

[15] Guido Salvaneschi, Gerold Hintz, and Mira Mezini. Rescala: Bridging between object-oriented and functional style in reactive applications. In *Proceedings of the 13th International Conference on Modularity*, MODULARITY '14, pages 25–36, New York, NY, USA, 2014. ACM. 2.3

[16] Andre Staltz. The introduction to reactive programming you've been missing. https://gist.github.com/staltz/868e7e9bc2a7b8c1f754. Accessed: 2018-06-09. (document), 2.2