Federal University of Pernambuco
Center of Informatics

BSc Computer Science

CEPSwift: Complex Event Processing Framework for Swift

Undergraduate thesis

George Belo Guedes

Recife
December, 2017

Federal University of Pernambuco
Center of Informatics

BSc Computer Science

CEPSwift: Complex Event Processing Framework for Swift

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.

Author: George Belo Guedes
(gbg@cin.ufpe.br)
Supervisor: Kiev Santos da Gama
(kiev@cin.ufpe.br)

Recife
December, 2017

To all those who went through my life,
and contributed in some way to my essence

Acknowledgements

It is really hard to thank everyone that helped me to get here. College was a scary period of my life and it was not easy to pass it through. Sometimes I thought about throwing it all in the air and giving up, but I ended up finding strength to go on. Some people were crucial during this long way, but I probably won't remember someone or won't have enough space to credit them all. Never the less, I would like to specially thank some people who were close to me all time along this journey: my parents, Cristina and Genilson, who never thought twice about investing in my education; my girlfriend, Marina, for all the affection and support; and my supervisor, Kiev, for introducing me to complex event processing and for all the patience and help during this work.

As Rocky Balboa said on Rocky movie "*Let me tell you something you already know. The world ain't all sunshine and rainbows. It is a very mean and nasty place and it will beat you to your knees and keep you there permanently if you let it. You, me, or nobody is gonna hit as hard as life. But it ain't how hard you hit; it's about how hard you can get hit, and keep moving forward. How much you can take, and keep moving forward. That's how winning is done. Now, if you know what you're worth, then go out and get what you're worth. But you gotta be willing to take the hit, and not pointing fingers saying you ain't where you are because of him, or her, or anybody. Cowards do that and that ain't you. You're better than that.*"

“*Don't look back. You're not going that way.”*

— Unknown

Abstract

Complex Event Processing (CEP) is a branch of event processing that has been gaining prominence in recent years. CEP allow to process event flows, enabling the definition and detection of high-level situations of interest, usually from a manipulation or combination of low-level simple events. CEP systems have great benefits such as being highly distributed and extremely loose coupled. Similarly to this kind of systems, reactive programming languages involves the same steps for detection, elaboration and propagation of changes to the interest parties. Despite the similarities, an integration between these two areas has been little explored. This work proposes the construction of a framework to perform Complex Event Processing in Swift, called CEPSwift, using as support a reactive programming framework.

Keywords: Complex Event Processing, Stream Processing, Event-Driven Architecture.

Resumo

Processamento de Eventos Complexos (*Complex Event Processing* - CEP) é um ramo de Processamento de Eventos que vem ganhando destaque nos últimos anos. CEP permite processar fluxo de eventos, possibilitando a definição e detecção de situações de interesse de mais alto nível, normalmente através de uma manipulação ou combinação de eventos simples de mais baixo nível. Sistemas CEP apresentam grandes benefícios como serem altamente distribuídos e extremamente desacoplados dos outros componentes. Similarmente a esses tipos de sistemas, linguagens de programação reativa apresentam as mesmas etapas para detecção, elaboração e propagação de mudanças para as partes interessadas. Apesar das semelhanças, uma possível integração entre as áreas vem sendo pouco explorada. Este trabalho propõe o desenvolvimento de uma biblioteca para processamento de eventos complexos em Swift intitulada CEPSwift, utilizando como suporte um framework de programação reativa.

Palavras-chave: Processamento de Eventos Complexos, Processamento de Streams, Arquitetura Orientada a Eventos.

List of Figures

List of Tables

List of Acronyms

CEP	Complex Event Processing
RL	Reactive Languages
API	Application Programming Interface
IDE	Integrated Development Environment
LOC	Lines of Code

Contents

1. Introduction

Event-Driven Architecture (EDA) is a computational software architecture in which the program flow is defined by events. Although the area has been studied since the 1990s, recently there has been a significant increase of interest on the part of the industry and academia [1]. This growth is mainly due to the increasing number of applications that are based on this principle, such as mobile applications, client-server architectures, web systems, chat systems, among others. This paradigm allows an application to be written in a simple way, defining a set of events and reactions according to the occurrence of each of them.

Parallel to the increase of interest in EDA, there was a notable growth in the area of Complex Event Processing (CEP) [1]. CEP is a technique used to process events, where the main goal is to define and detect situations of interest, usually through a combination of simple events. This way, it is possible to set rules and trigger actions when a particular pattern is encountered.

One of the first patterns of message propagation focused on notification filtering is the publish-subscriber pattern (pub-sub). The idea is that anyone who has an interest in receiving event notifications subscribes, not knowing who will publish. Similarly, event sources will post the occurrence of an information item on a channel, not knowing who will receive it, and who is subscribed will be notified. There is a crucial difference between this pattern and CEP: while the former allows only the enrollment in events of a certain type, individually [2], the second allows to take into consideration the history of events or even the relationship between different types of events, arriving from diverse sources. This way, CEP can be seen as an extension of the pub-sub pattern [3], allowing those who sign up to express rules that infer complex patterns by combining two or more events.

1.1 Motivation

In complex event processing, the composition of events is defined by the programmer. He defines how to select, manipulate and combine these events. Usually, existing frameworks for complex event processing are based on query languages to define rules (queries), which makes its usage unintuitive for people who do not have experience with SQL-like languages. It is crucial that a language for CEP might be expressive [4] and its API also be intuitive, making it possible to create rules and patterns even by people who are not experts in the field.

Efforts have been made in the area [7] in order to provide modeling and more intuitive languages.

Both reactive languages (RL) and CEP provide a declarative way of defining entities. Although there are some differences, both involve the same steps for the detection, elaboration and propagation of changes that occur in the sources to the interested parties. Still, there is little study in the area and the two communities are distant, with no knowledge exchange and an integration between both would benefit the two areas [4]. Also, according to [4], a possible approach for integration between the two areas is to add to the RLs the possibility to operate on past values and support to common CEP operations such as filtering, pattern detection and window concept.

1.2 Goals

This work proposes a framework called CEPSwift. The main goal is to make event generation and stream handling easier in Swift and also provide complex event processing capabilities. By the lack of studies done in the area and the complementary relationship between reactive languages and CEP [4], libraries that give support to reactive programming in Swift are reused to build the CEP solution.

Swift is an open-source language, widely used to develop mobile applications for iOS platform. These applications are inherently event driven. In addition, there is growing interest by the industry in server applications built in Swift [6].

Along with the construction of CEPSwift framework, this work has the following specific goals:

- Perform a study of complex event processing area;
- Establish a parallel between CEP and reactive languages;
- Implement and demonstrate a case of use of the CEPSwift framework through a practical application example;
- Provide CEPSwift as an open-source library framework for the community, encouraging contributions from others developers.

1.3 Structure of the document

This document is structured as follow:

- Chapter 2 gives a background of a few areas that are important to the understanding of this work: First, there is an overview about event-driven architecture, later complex event processing is discussed and last the relationship between reactive languages and complex event processing;
- Chapter 3 discusses about the framework proposal;
- Chapter 4 discusses about the implementation of the proposal;
- Chapter 5 illustrates two use cases of the framework;
- Chapter 6 discusses the conclusions and future work.

2. Background

This chapter discusses about the most important background topics involving CEP. First, there is a discussion about Event-driven Architecture; then, Complex Event Processing is discussed; finally, there is an overview about the relationship between Reactive Languages and CEP.

2.1 Event-driven Architecture

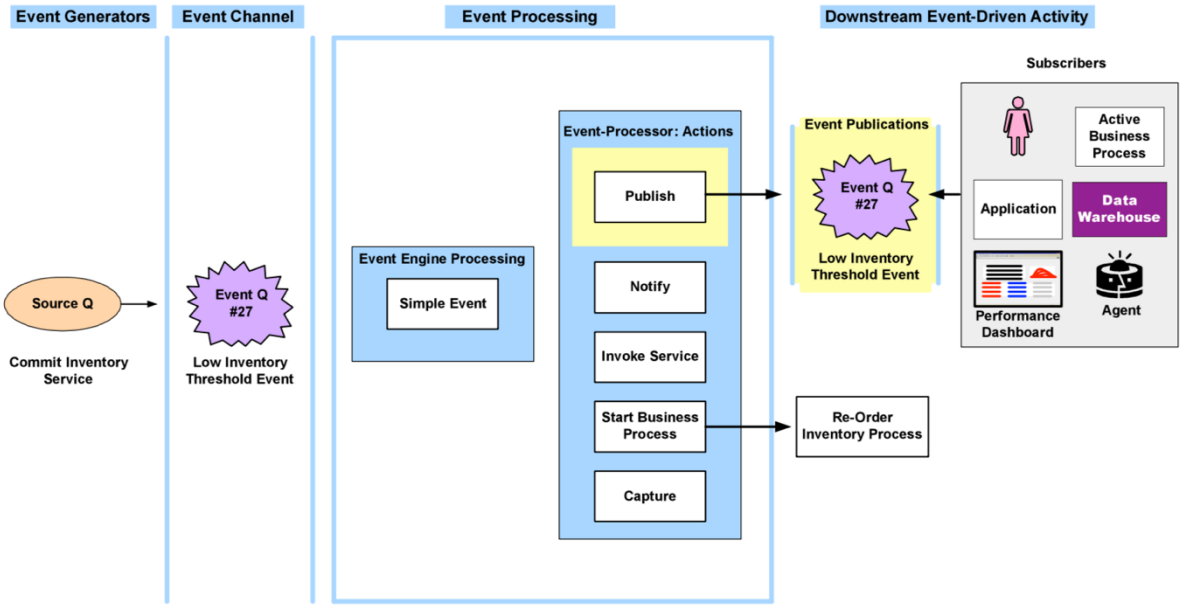
Event-driven Architecture is a computational software architecture where programs are event-based. The definition of an event is straightforward: an event is a generalization of a notable thing that happens and that is interest in the domain. For example, to an e-commerce company, each order is an event of interest. The term event is often used to refer to the event definition but also to each instance of an event. It is important to notice the temporal aspect present on this kind of architecture, in which each event is associated to a timestamp and the order of the events is defined by the time of their occurrences.

EDA applications can be built with four logical layers [14], each one with their own responsibilities in order to have a well modularized system:

- **Event generator:** This logical layer can sense the occurrence of something of interest and is responsible for representing this into an event. It is important to notice here that "something of interest" can be nearly anything that can be sensed. Examples of event generators are an e-commerce system or physical sensors;
- **Event channel:** This layer is responsible for transporting standard well formatted events between the event generator and the event processing layer;
- **Event processing:** The event processing layer receives the event and evaluates it against event processing rules to execute the proper reactions. These rules and reactions are defined by the interest parties. For example, in an e-commerce system, an event of "product ID with low stock" can trigger reactions like "Notify department";
- **Downstream event-driven activity:** Many activities can be initiated by a single event occurrence or a correlation of events. This can be a push by the event

processing or even a callback that is executed when the event of interest takes place.

Figure 1- Simple Event Processing Flow with layers exemplification



Source: Michelson, Brenda M. "Event-driven architecture overview." Patricia Seybold Group 2 (2006).

In the event-driven architecture there are three general styles of event processing: simple event processing, stream event processing and complex event processing. It is important to notice here that these three styles often appear together in an application that uses EDA and they are not in any way mutually exclusive:

- **Simple Event Processing:** In this style, when an event of interest occurs a proper reaction is initiated. It does not take into consideration correlation between events of the same type that had occurred before;
- **Stream Event Processing:** In stream event processing, there is an interest to keep receiving continuously events of a same type. Both events of interest or ordinary events can occur and all of them are stored in streams;
- **Complex Event Processing:** In complex event processing, there is an interest to extract a certain behavior from the occurrence of a combination of events from different sources and then react according. This style is the focus of this work.

The event-driven architecture has great benefits like being inherently highly distributed and extremely loose coupling. The highly distributed characteristic arises from the fact that anything can be seen as an event and they exist practically everywhere. So, when building an

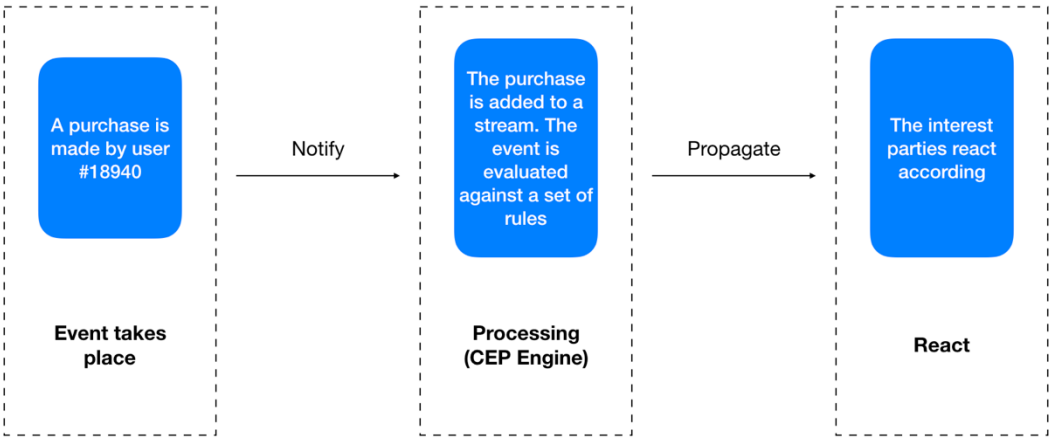
application, it is easy to have micro-services connected. The extremely loose coupling characteristic arises from the fact that the producer does not need to know who is consuming the events and neither the consumer needs to know who is producing the events. Typically, EDA have real time needs and thus the work and information flow happens asynchronously.

2.2 Complex Event Processing

The concepts of timeliness and flow processing are crucial to explain the need for a new class of systems that are capable of processing data-flows in real time [3]. In traditional database management systems (DBMSs), data is persistently stored and indexed before it is available for query and this does not fit the needs of real time event processing applications. Complex event processing arises from this requirement and can be seen as a branch of event processing [8] where the goal is to define and detect situations of interest, usually through a combination of simple events, and react to them as quickly as possible.

Events can relate to each other by many aspects such as time, causality or aggregation [9]. New events can be generated by observing the relationship between a set of events. These new events are often referred as complex events [9]. Thus, CEP provides a way of detect patterns by filtering, correlating, contextualizing and analyzing data provided by different sources, taking in consideration the order of the events. For example, in the context of a credit card company, each credit card purchase can be seen as an event. A purchase event can have many attributes such as a flag indicating if the purchase was online or face-to-face, the location of the beneficiary establishment and the timestamp. For each credit card client, there will be a stream of purchase events. The credit card company can use CEP to keep listening to purchases of each client and block the credit card if two face-to-face purchases happens in a short period of time and in a long range of distance, where the period of time and range of distance are variables defined by the system. Figure 2 illustrates the CEP mechanism by using the credit card company example.

Figure 2 - Complex Event Processing mechanism



According to Eckert and Bry [12], there are four aspects to be considered as requirements for an event query language:

- **Data extraction:** Events contain important data. It is crucial for a query language to be able to access this information in order to react to them, to enrich other data or even to generate new events;
- **Composition:** It is important to join individual events from different sources, combining their occurrence in order to obtain high-level situations;
- **Temporal relationship:** The temporal aspect is crucial in event processing systems. It should be possible to express in the rules temporal conditions such as defining an interval for the occurrence of an event or a sequence of events in a particular order;
- **Accumulation:** It should be possible to query for accumulator operators such as maximum, minimum or average over a certain finite slice (or window).

Additionally, two types of rules can be defined to perform CEP operations: deductive rules, where new events are defined based on event queries, regarding the occurrence and correlation of a set of events; and reactive rules, where it is specified how the system should react to the occurrence of single events or complex events, usually with a procedure that will be executed.

2.2.1 Operators

Complex event processing operators are the core of a CEP engine. It is through the operators that the rules will be expressed, identifying situations of interest by correlating events in different aspects. Firing conditions can be written using the CEP operators and when the conditions hold, actions defined will be taken. These conditions are usually defined as patterns that match portions of events in a certain stream, using logical operators, content and timing constraints.

CEP engines allow users to express rules in different paradigms and can even use more than one in the same system [3]. The most commons paradigms used are declarative, which usually includes languages derived from relational languages like relational algebra or SQL; imperative, which usually offers visual interfaces to define rules and detecting; and pattern-based, where rules are separately specified the conditions and procedures that will be executed.

Cugola and Margara in [3] define classes of CEP operators as illustrated in Table 1.

Table 1 – Complex event processing common operators. Summarized from Cugola and Margara, 2013 [4]

Kind of operator	Class of operator	Description
Single-Item operator	Selection operators	Selection operators can filter items if they satisfy a given constraint.
	Elaboration operators	These operators can transform information present on items.
Logical operators	Conjunction	A conjunction of items *I1, I2… In* is satisfied when all the items *I1, I2… In* have been detected.
	Disjunction	A disjunction of items *I1, I2… In* is satisfied when at least one of the information items *I1, i2… In* has been detected.
	Repetition	A repetition of an information item *I* degree *<a, b>* is satisfied when the item I is detected at least *a* times and not more than *b* times.
	Negation	A negation of an information item *I* is satisfied when *I* is not detected.
Sequences	Sequence operator	Sequences are used to capture arrival of a set of information items, but they take into consideration the order of arrival. A sequence defines an ordered set of information items *I1, I2… In* which is satisfied when all the items *I1, I2… In* have been detected in the specified order.

Kind of operator	Class of operator	Description
Windows	Fixed windows	This class of windows do not move and they size is fixed.
	Landmark windows	They have a fixed lower bound while the upper bound is variable, growing every time a new information arrives.
	Sliding windows	Sliding windows have a fixed size, with a fixed lower and upper bound and the window slides as a new information item arrives.
	Pane and tumble windows	They are a variant of sliding windows, in which the lower and upper bound mode by *k* elements, as *k* elements enter the system.
Flow management operators	Join operator	Used to merge two flows of information.
	Union	Merge two flows of information of the same type.
	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.
	Intersect	Takes two or more input flows and outputs only the items included in all of them.
	Remove-duplicate	Removes all duplicates from an input flow.
	Duplicate	Allow a single flow to be duplicated in order to use it as an input for different processing chains.
	Group-by	This operator is used to split information flows into partitions in order to apply the same operator (usually an aggregate) to the different partitions.
	Order-by	This operator is used to impose an ordering to the items of an input flow.
Flow creation operators	Flow creation	Some languages define explicit operators for creating new information flows from a set of items.
Aggregates	Detection aggregates	Are those used during the evaluation of the condition part of a rule.
	Production aggregates	Are those used to compute the values of information items in the output flow.

2.2.2 Applications

The increase of applications that use Event-driven Architecture and the need of monitoring IT systems due to legal, contractual or operational concerns has led to a significantly increase of event generation in computer systems. In this kind of application there is a need to manage and process the occurrence of events in an automatic and systematic way [12]. Complex event processing has been widely used for this purpose in many areas such as:

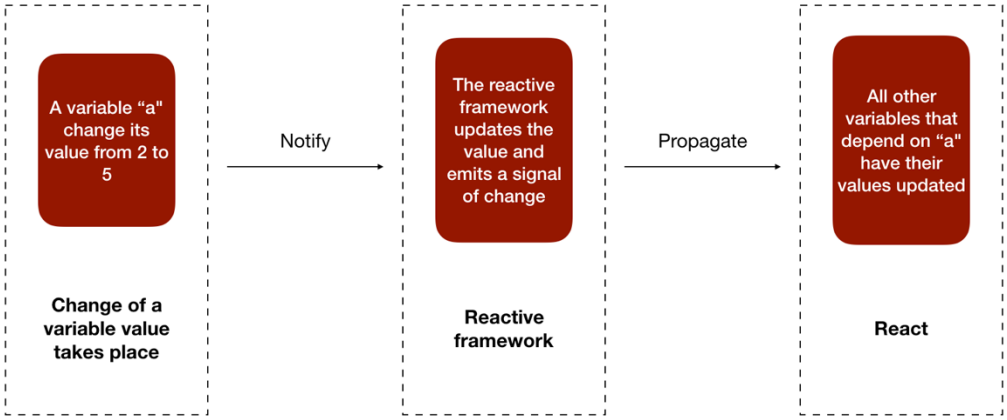
- **Business Activity Monitoring:** In business activity monitoring, the focus is to identify problems and opportunities as quickly as possible. The application of CEP in this area allows the extraction of performance indicators that can be used to improve the business and fix problems;
- **Sensor networks:** Sensors can read data from physical world. Often there is a need of combine data from different sources and analyze the behavior of a certain measure during a period of time;
- **Market data:** The prices of stock or commodity in market data can be considered as events. In trading algorithms, there is a need to analyze the prices continuously in order to recognize trends early and react to them automatically, excluding the need to have a person checking prices and making decisions.

2.3 Reactive languages and CEP

Reactive languages provide an abstraction to model values that change over time. These values can be anything such as variables, clicks, user inputs, properties, data structures and so on. They deal with data streams asynchronously and propagate the changes to those who are interested. For example, in a social network there is a “likes” counter for each post. This counter updates according to how many users clicked on the like button of that post. On the mobile application front-end, there will be a UI component that will display the value of the “likes” counter. Using reactive languages on the app, every time the counter is updated, the change is instantly reflected on the UI component. This is the reactive way to propagate signals of change. The RL mechanism is illustrated in Figure 3.

This solution contrasts with the Observer pattern and is considered as a better way of dealing with values that changes because it can give more composability and increases the readability of applications [4]. RLs emerged from purely functional languages but later were introduced in several other types of languages. Nowadays, there are various extensions that enable reactive programming in different programming languages. A very famous one that has been widely used in big companies such as Netflix is the ReactiveX [5] library, which provides a reactive framework for several mainstream programming languages.

Figure 3 - Reactive Languages mechanism



By comparing the CEP mechanism from Figure 2 and the RL mechanism from Figure 3 it is possible to notice that they both involve the same steps of execution. The main difference is that complex event processing focuses on event occurrences. Each time a new event is generated the CEP engine is notified and a set of rules is processed to check if a certain pattern is encountered. If it is, this is propagated to the interest parties and they can react accordingly. Reactive languages focus on the change of a value and not in event occurrences. Every time a value is changed the reactive framework updates the value and propagates a signal of change. All the variables that depend on the value that just changed have their values updated.

Margara and Salvaneschi in [4] identified five main steps on the reactive behavior:

- **Observation:** This phase is when a fact of interest takes place. In a customer service center, for example, observation happens each time a new case arrives;
- **Notification:** After the event happened, it is encoded to a notification;
- **Processing:** The notification triggers an action;
- **Propagation:** The result of the actions is propagated to the interested parties;
- **Reaction:** The interested parties receive the results and react according.

Both RLs and CEP go through the same five steps of the reactive cycle, although with some differences on each phase. In CEP engines, in the observation phase, the sources observe and propagate the occurrence of an event. In RLs, the focus is on a signal that represents the changes of a value. In CEP, event notifications happen explicitly and is usually pushed from the sources to the processing component. There is a set of rules that defines how to manipulate and combine events to produce the desired results. In contrast, in RLs, the

notifications are implicit. The processing phase is defined by an expression that specifies how the result should be based on the value change. The propagation also happens explicitly in CEP systems and implicit in RLs, but in both the results are propagated to all interest parties. After delivering the results, CEP systems do not impose any limitations on reactive phase. External clients receive the results and can react in any way they want; in RLs, the reaction is always defined by a change into a value.

Margara and Salvaneschi in [4] also provide a more detailed analysis of some general key aspects of CEP and RLs that are summarized in the table below:

Table 2 - Analysis of similarities and differences of key aspects. Summarized from Cugola and Margara, 2013 [4]

Aspect	Analysis
Language expressiveness	CEP systems work with the occurrence of events. New events are generated from the sources and they are time based. CEP rules extract patterns or behaviors from the combination or manipulation of these events often involving temporal aspect. In RL's, there are signals that represents values that can change over time. The goal is to react according to these changes but the temporal aspect does not have much importance as in CEP systems.
Composability	They both offer support for composability. In CEP rules, it is possible to compose rules to define the occurrence of new events that satisfy those rules. That way, it is possible to define high-level situations of interest from simple events occurrences. In RL's, instead of events the main interest is on signals that represents value changes. Often, they are composed into expressions that produce new signals.
Consistency	Consistency is a real problem in software systems. In CEP systems, the rules are always processed in timestamp order. This ensure that if an event A happened before an event B, A will be processed and propagated before B. But in the case of events that happened at the same time, there are no guarantees of in which order they will be processed. In RL's, there is a level hierarchy that dictates in which order the values should update, in order to ensure that the variables always hold valid values.
Performance	Performance is a key aspect of CEP systems because they have real-time or near real-time needs. That way, the community is often studying new ways to reduce processing time and increase throughput. In RL's, performance has not been a real issue.

Aspect	Analysis
Distribution	Distribution is another key aspect of CEP systems. They usually have a client-server architecture, where the CEP engine behaves like a server, receiving events, processing them and distributing events to other client components. This makes CEP systems highly distributed and extremely loose coupling. In the RL's context distribution has not been explored. Usually they are used as a programming solution to allow reactive programming.
Safety	CEP frameworks usually offers less guarantee on the format of the information. For example, existing frameworks use SQL to define rules. The queries are treated as strings and the compiler can not detected type inconsistencies. Moreover, there are a few studies that integrates event-driven programming and event composition, allowing event types and rules to be checked at compile time. In RL's, signals are often types that are integrated in the language. In languages that are statically typed, signals expressions are type-checked and the compiler ensures that the program works. That way, expressions that combine signals are type safe.
Interaction with Object-Oriented Features	In CEP systems, the integration between event-based programming in object oriented languages is starting to be studied by the community. In RL's, the signal definition was introduced in the context of funcional languages. The community has been creating proposals to integrate signals in OO languages, but there are some challenges in the area like how to detect and notify with mutable objects and how to deal with signals and inheritance, polymorphism and other OO abstractions.

Although there are differences between CEP and RLs, there is a lot in common between them: they involve the same execution steps and have a lot in common in some key aspects. But despite these similarities, [4] points it out that research in the areas have been carried separately by the two communities. The communication between them should be stimulated because each community can benefit from results and techniques successfully applied to each other. According to [4], one way to integrate them both is to extend RLs expressions to operate over past values of a signal and also support all common CEP operators.

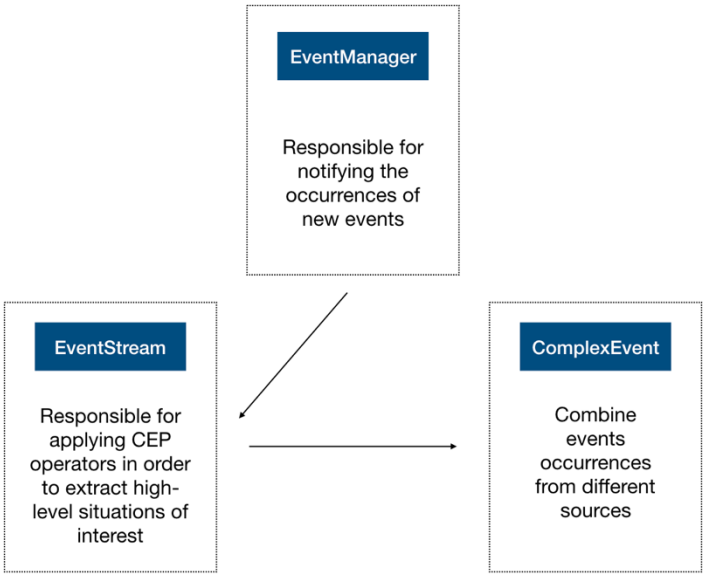
3. Proposal

This work proposes a framework called CEPSwift. The main goal is to make event generation and stream handling easier in Swift, and also provide complex event processing capabilities. The choice of Swift is also justified on the fact that, at the moment this work is being written, there are no complex event processing solutions built for Swift. Moreover, Swift is an open-source language that is continuously improving and is widely used to build mobile applications that, in their nature, are event-oriented. Also, there is a great increase of interest by the industry in server applications built in Swift [6]. The benefits of complex event processing are even bigger in server applications that often deal with a massive quantity of data and also require high performance, in real-time (or near real-time).

By the absence of studies in integrating reactive languages and CEP, the complementary relationship and the potential of integrating them [4], libraries that give support to reactive programming in Swift are reused to build the CEP solution. It is important to make it clear that CEPSwift, at first, is an implementation of just a subset of CEP operators because of the scope and time available for the work. Some other factors of choice are listed in the next chapter.

At first, the main entities and the API architecture were, in a high-level, modeled as follows to guide the development process:

Figure 4 - CEPSwift high-level architecture



The EventManager class was designed to be responsible for notifying the occurrences of new events. The EventStream class was designeded to be responsible for applying CEP operators. This class should enable easy composition of the rules in the same way reactive languages work. In order to combine events from different sources, another entity was created: ComplexEvent.

4. Implementation

This chapter presents an overview of the CEPSwift library implementation. First there is a discussion about the technologies used in the implementation process and later about the technical approach of the solution. All code is available in GitHub[1] as an open-source library under MIT license and on CocoaPods,[2] a dependency manager for Swift.

4.1 Technologies

All classes in CEPSwift library were written in Swift 4.0. Internally, RxSwift framework was used to add reactive programming capabilities to the developed API. RxSwift is a generic abstraction of computation expressed through an interface called Observable<Element>. The main idea is to deal with asynchronous operations and data streams. There are others frameworks that enable reactive programming in Swift but RxSwift was chosen because it is an open-source extension for Swift of the well-known reactive programming library ReactiveX [5] that is continuously improving with new releases.

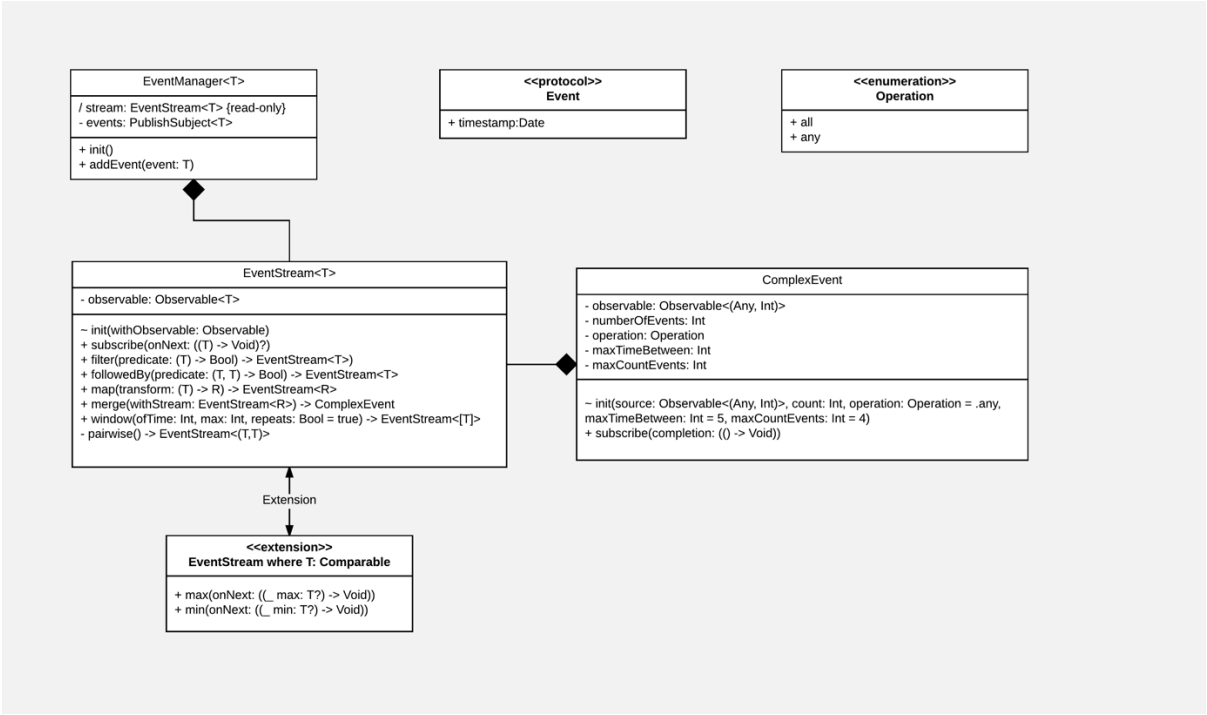
The IDE used in the project was Xcode, Apple's Official IDE for Mac and iOS development. Also, during the library implementation phase, GitHub was used for versioning the code and to host the source files. CocoaPods was used to add RxSwift as external dependency and, later, to distribute CEPSwift as a library. Also through CocoaPods, Quick framework was added to the project to help the writing of unit tests for each CEP operator implemented.

4.2 Technical Approach

CEPSwift library consists of four main entities: Event, EventManager, EventStream and ComplexEvent. There are other auxiliary classes, but they are not as important as these three to understand the architecture of the solution. Figure 5 illustrates an UML diagram of all CEPSwift classes and each class implementation and design decisions are explained in the following subsections.

[1] Available at <https://github.com/guedesbgeorge/CEPSwift>
[2] Available at <https://cocoapods.org/?q=cepswift>

Figure 5 - CEPSwift UML classes diagram



4.2.1 Event and EventManager

The Event entity is a protocol (Swift protocols are equivalent to interfaces). This decision was made because it makes sense to have an event protocol or, in other words, a protocol that every class that models a well-formatted event should conform to. To conform with the protocol, the class must have a timestamp attribute that should be initialized inside the constructor, that way keeping the temporal aspect. Other attributes can be added to the protocol in the future as needed like, for example, an identifier or a string describing the source. However, this must be done carefully to not cause backward compatibility issues.

The EventManager is a generic class that, as the name says, manages the occurrences of an event. The application class where the events will be generated should hold an instance of the EventManager<MyEvent>, where MyEvent is a model class that conforms to the Event protocol and represents the event of interest. Any time a new MyEvent occurs, it should be added to the EventManager by just calling the *addEvent* function. Using this instance, the user can get an EventStream instance, where he can describe rules in order to extract patterns of the event occurrences and high-level situations of interest.

4.2.2 EventStream and ComplexEvent

The idea of mixing reactive programming with complex event processing is to use the functional paradigm when creating rules and subscribing to events. In practice, by using the EventManager instance, the user can have access to an EventStream, which is a class that represents a stream with all occurrences of an event. Every time the user has an EventStream he can call a CEP operator function implemented in the library and this will return a new EventStream instance that holds only the events that satisfy the rule applied. The user can easily compose different rules by applying the operators one after the other, nesting the rules. Later, he can subscribe to that EventStream and pass a completion as parameter that will be executed every time a new event that satisfy that condition arrives.

This gives great power to an event-driven application because the class that knows about the occurrence of an event publishes them by adding to the EventManager. Other classes that need to extract high-level situations can define rules and trigger actions when the rules are satisfied by just holding an instance of the associated EventStream, keeping the code modularized and the all components independent.

The initial idea was to implement a protocol called Streamable with all CEP operators that the EventStream should conform, in order to make it easier change the internal support framework for reactive programming later in future tests. But this was not possible because Swift does not allow to create a protocol of a generic type and have inside the protocol functions that return the same type that conforms with the generic protocol. This was necessary because each CEP operator in EventStream returns a new instance of EventStream with all the events that satisfy the condition. The way of achieving that is by using a pattern called “type erasure”, but it is not intuitive and neither easy to implement in a short time, so this will be left as future work.

The ComplexEvent class is required in order to merge different instances of EventStream that can have different event types. When merging two different EventStreams, the result is a ComplexEvent that allows the subscription by passing a callback function that will be executed when all the events happens in a certain interval of time.

4.2.3 Implemented operators

The implemented operators are available through the EventStream class. The choice of which ones would be implemented in the library involved factors such as available time, implementation complexity and operator relevance through analysis of others CEP frameworks. The classes of CEP operators listed in Table 1 were analyzed and other library implementations were studied. The implemented operators are illustrated in Table 3, grouped by their respective classes.

Table 3 - Implemented operators

Kind of operator	Operator	Description
Single-Item operators	Filter	The filter operator can filter just events that satisfy a predicate that takes in consideration the attributes of the event. For example, a hypothetical event class that models a location update has 3 attributes: latitude, longitude and speed (in m/s). One can filter only events that has speed higher than 4 m/s. RxSwift already has this operator implemented, and the implementation was basically a wrap of RxSwift function.
	Map	The map operator creates a new EventStream by mapping each element of the original EventStream. RxSwift also has this operator implemented, and the implementation was a wrap of RxSwift function.
Flow management operators	Merge	The merge operator merges two different event EventStreams and returns a new object called ComplexEvent, which can notify when the events happen together in a certain interval of time; RxSwift only allow to merge observables of the same type. This is different from the requirements of CEP because often there is a need to merge streams of different event types. The implementation of this operator was a manipulation of RxSwift observables in order to achieve the desired result.
Aggregates	Maximum and Minimum	If the event class conforms with the comparable protocol, these functions returns the maximum and minimum event, respectively. RxSwift does not have maximum and minimum operators built in. The solution was to manipulate the observable class in order to calculate the maximum or minimum values.
Windows	Window	The window operator groups the events in windows of a given parameter size. RxSwift already has this operator implemented, and the implementation was a wrap of RxSwift function.

Kind of operator	Operator	Description
Sequences	FollowedBy	The followedBy operator can filter only events that satisfy a predicate that takes in consideration the event that happened immediately before. Using this, one can check, for example, if the speed of a location update is continuously increasing. This operator gives the power to operate over past values and RxSwift does not have this implemented. The CEPSwift API implementation had to manipulate the observable class in order to achieve the result.

5. Use case

This chapter describes two use cases of the CEPSwift library. First, a simple one that detects shake movements and later a more difficult one that detects if the user is walking. For each of them, there is a discussion about the problem and the requirements involved and then the solution using the developed API is presented. Also, there is a brief comparison of how the solution looks like if it was implemented without using the library, just by using pure reactive programming.

5.1 Shake movements detection

The problem addressed in this use case is simple: the detection of a simple shake movement by using accelerometer data from the mobile device. The accelerometer sensor provides the acceleration force value along the *x*, *y* and *z* axis and it is available from Apple CoreMotion native library.

The main idea of the solution is to check if there is a big difference between two consecutive accelerometer readings. This difference is set through a threshold value. In this use case, each accelerometer reading is considered as an event. Figure 6 shows the definition of a model class that conforms with the event protocol that models the event of interest.

Figure 6 - Model class that represents the accelerometer data reading

```
class AccelerationEvent: Event {
    var timestamp: Date
    var acceleration: CMAcceleration

    init(data: CMAcceleration) {
        self.timestamp = Date()
        self.acceleration = data
    }
}
```

After defining the event model class, an instance of the EventManager should be created. Every time a new acceleration reading arrives, the value should be added to the manager (Figure 7).

Figure 7 - EventManager instance for AccelerometerEvent

```
let motion = CMMotionManager()
let manager = EventManager<AccelerationEvent>()

override func viewDidLoad() {
    setRules()
    motion.accelerometerUpdateInterval = 0.1
    motion.startAccelerometerUpdates(to: .main) { (data, error) in
        guard error == nil else { return }
        guard let data = data else { return }
        self.manager.addEvent(event: AccelerationEvent(data: data.acceleration))
    }
}
```

One can now define a high-level situation "*device shake detected*" by looking for two consecutive accelerometer force readings with a difference between them that exceeds a threshold previously defined. Using CEPSwift library, the rules can be defined by applying the followedBy operator and checking if the absolute value of the difference between the previous and the current accelerometer reading is higher than 1. The rule definition is illustrated in Figure 8.

Figure 8 - Rules definition using CEPSwift to detect shake moviments

```
func setRules() {
    manager.asStream().followedBy { (fst, snd) -> Bool in
        abs(fst.acceleration.x - snd.acceleration.x) > 1 ||
        abs(fst.acceleration.y - snd.acceleration.y) > 1 ||
        abs(fst.acceleration.z - snd.acceleration.z) > 1
        }.subscribe { (event) in
            print("You shaked too fast!")
    }
}
```

Figure 9 shows how the code would be if the same rules were written by using pure reactive programming with RxSwift, to contrast with how they look by using CEPSwift in Figure 8. For simplicity, the lines of code (LOC) metric was used to compare both codes. The solution using CEPSwift has 7 LOC and the solution that uses pure RxSwift has 21 LOC, three times greater. The reason why the code is longer lies on the fact that reactive frameworks does not make it possible to work with previous values easily. In order to achieve this behavior, the observable structure from RxSwift API had to be manipulated.

Figure 9 - Rules for shake detection using pure RxSwift

```
func setRules() {
    var previous:CMAcceleration? = nil
    let observableWithPreviousValue = manager.asObservable().filter { element in
        if previous == nil {
            previous = element
            return false
        } else {
            return true
        }
        }
        .map { (element:CMAcceleration) -> (CMAcceleration,CMAcceleration) in
            defer { previous = element }
            return (previous!, element)
    }

    _ = observableWithPreviousValue.filter { (arg) -> Bool in
        let (fst, snd) = arg
        return abs(fst.x - snd.x) > 1 ||
            abs(fst.y - snd.y) > 1 ||
            abs(fst.z - snd.z) > 1
        }.subscribe { (event) in
            print("You shaked too fast!")
    }
}
```

5.2 Walk movements detection

The problem addressed in this use case is an iOS application capable of identifying if a user is walking or if he is not walking by combining data from the pedometer sensor and GPS, both data available from Apple native libraries. The application must guarantee that the user is walking and not driving a car or riding a bicycle, for example. The difficulty arises from the fact that, if it is considered only the pedometer data, one can easily just shake the phone and this will make the pedometer sensor increment the number of steps even if the user is standing still, because it is based on motion. On the other hand, if it is considered only GPS speed data, one can get on a car on a slow speed and this will seem like the user is walking.

The main idea of the proposed solution is to combine the data occurrences from both sensors. In this case, two different kind of events are defined. First, a pedometer data event, which is a basically an update of the pedometer sensor from the native iOS CoreMotion library. This event holds attributes such as number of steps, distance and start date of the measurement. The model class that represent this first type of event is defined in Figure 10 and conforms with the Event protocol. The other kind of event is a location data event, which is basically an update of the CoreLocation GPS data, and holds attributes such as latitude, longitude and speed (Figure 11).

Figure 10 - Model class that represents the pedometer update event

```
class PedometerEvent: Event {
    var timestamp: Date
    var data: CMPedometerData

    init(data: CMPedometerData) {
        self.timestamp = data.startDate
        self.data = data
    }
}
```

Figure 11 - Model class that represents the pedometer location event

```
class LocationEvent: Event {
    var timestamp: Date
    var data: CLLocation

    init(data: CLLocation) {
        self.timestamp = data.timestamp
        self.data = data
    }
}
```

After defining the model classes that represents the event of interest, an instance of EventManager for both events should be created on the application view controller class. Then, whenever we receive a pedometer update or a GPS data update we create an event and add to the EventManager (Figure 12).

Figure 12 - EventManager instances for PedometerEvent and LocationEvent

```
// Creating the EventManager for both events
var pedometerEvents = EventManager<PedometerEvent>()
var locationEvents = EventManager<LocationEvent>()

// Receive GPS updates
func locationManager(_ manager: CLLocationManager, didUpdateLocations locations: [CLLocation]) {
    guard let location = manager.location else { return }
    // Discard if horizontal accuracy is too bad
    guard location.horizontalAccuracy < 20.0 else { return }

    // Add the event to the LocationEvent
    locationEvents.addEvent(event: LocationEvent(data: location))
}

// Receive pedometer updates
func pedometerHandler(_ pedometerData: CMPedometerData?, _ error: Error?) {
    guard let data = pedometerData else { return }

    // Add the event on the PedometerEvent
    self.pedometerEvents.addEvent(event: PedometerEvent(data: data))
}
```

One can define a high-level situation of interest "*user is walking*" by checking if the pedometer data keeps increasing the number of steps continuously. This can be easily done by using the followedBy operator, passing as parameter a predicate that checks if the current number of steps is higher than on the event immediately before. But, as mentioned before, this does not guarantee that the user is walking because he may just be shaking his phone. Combined with this we want to check if we also receive an update from the GPS data and check if the speed is between an established threshold, that delimits a normal walking speed for human beans. This can be achieved by using the filter operator. The merge operator can be used to combine the occurrences of both events.

Another high-level situation of interest is "*user is not walking*". This can be easily extract by checking if the GPS speed data is less than a minimum walking speed, which means that the user is probably standing still, or higher than a maximum walking speed, which means that the user is probably running, cycling or even driving a car. This also can be achieved by using the filter operator.

The rules that detect both high-level situations of interest described before are illustrated on Figure 13.

Figure 13 - Rules definition using CEPSwift library to detect walk movements

```
func setRules() {
    // This rule assure that the person is walking in a regular walk speed
    let walkingRule1 = locationEvents.asStream().filter(predicate: {$0.data.speed > 0.2 && $0.data.speed < 1.8})

    // This rule assure that the number of steps is increasing
    let walkingRule2 = pedometerEvents.asStream().followedBy { (fst, snd) -> Bool in fst.data.numberOfSteps.intValue < snd.data.numberOfSteps.intValue }

    // When our first and second rules happen, user is walking! Let's set our backgroung to blue!
    walkingRule1.merge(withStream: walkingRule2).subscribe { self.setBlueBackground() }

    // When the speed is too low the user probably stopped walking and when the speed is too high, the user is probably in car or bicycle
    let stopWalkingRule = locationEvents.asStream().filter(predicate: {$0.data.speed < 0.2 || $0.data.speed > 1.8})

    /// When our stopWalkingRule occurs, the user isn't walking! Let's set our background to red!
    stopWalkingRule.subscribe { (location) in self.setRedBackground() }
}
```

Figure 14 shows how it would be the code if the same rules were written by using pure reactive programming with RxSwift. The solution using CEPSwift has 5 LOC and the solution that uses pure RxSwift has 35 LOC, seven times greater.

Figure 14 - Rules to detect walk movements using pure RxSwift

```
func setRules() {
    let walkingRule1 = locationEvents.asObservable().filter({$0.speed > 0.2 && $0.speed < 1.8})

    var previous:CMPedometerData? = nil
    let observableWithPreviousValue = pedometerEvents.asObservable().filter { element in
        if previous == nil {
            previous = element
            return false
        } else {
            return true
        }
        }
        .map { (element:CMPedometerData) -> (CMPedometerData,CMPedometerData) in
            defer { previous = element }
            return (previous!, element)
    }

    let walkingRule2 = observableWithPreviousValue.filter { (fst, snd) -> Bool in fst.numberOfSteps.intValue < snd.numberOfSteps.intValue }

    let mappedRule1 = walkingRule1.map({Events.location($0)})
    let mappedRule2 = walkingRule2.map({Events.pedometer($0.0, $0.1)})

    let mergedObservables = Observable.merge(mappedRule1, mappedRule2)
    _ = mergedObservables.buffer(timeSpan: RxTimeInterval(5), count: 4, scheduler: MainScheduler.instance).subscribe({ (buffer) in
            guard let events = buffer.element else { return }
            var locationHappened: Bool = false
            var pedometerHappened: Bool = false
            for event in events {
                switch event {
                case .location(_):
                    locationHappened = true
                case .pedometer(_):
                    pedometerHappened = true
                }
            }

            if(locationHappened && pedometerHappened) {
                self.setBlueBackground()
            }
        })

    let stopWalkingRule = locationEvents.asObservable().filter({$0.speed < 0.2 || $0.speed > 1.8}).subscribe({ (location) in self.setRedBackground() })
```

5.3 Discussion

In this chapter two use cases of the CEPSwift library were shown. First, a simpler one to detect shake movements was discussed. Even though it is a simple application, CEPSwift library made it easier to define rules and detect high-level situations, in contrast with using just a reactive framework to build the same solution, where the implementation would be much polluted with boilerplate code. Later, a second use case was discussed to detect walk movements. This use case was more complicated than the previous one because the proposed solution combined events from different sources and composed different rules to guarantee that the user is walking. As previously noted, the solution using just the reactive framework had a higher LOC than using CEPSwift library.

It is important to notice that in both use cases the followedBy operator was used. This operator allows to handle and manipulate previous events occurrences and this behavior is not usually built in reactive programming. CEPSwift library implements this behavior by manipulating the observable class of RxSwift and exposing only the function and the desired result, making CEP operation easier in Swift. In the future, other metrics besides LOC such as

complexity can be used to compare the solutions using CEPSwift and pure reactive programming.

6. Conclusion

Complex Event Processing is a technique used to process events, where the main goal is to define and detect situations of interest, usually through a combination of simple events. This way, it is possible to set rules and trigger actions when a particular pattern is encountered. Reactive languages and CEP have a complementary relationship and both involve the same execution steps and have a lot in common in key aspects.

6.1 Considerations

The main goal of this work was to implement common complex event processing operations in Swift in order to facilitate stream handling and event generation in Swift. The proposed API is called CEPSwift. First, it was given a background of event-driven architecture, CEP and the complementary relationship between reactive languages and complex event processing.

Despite the similarities presented between RLs and CEP, the research in both areas have been carried separately by the two communities. One way to integrate them both is to extend RLs expressions to operate over past values of a signal and also support all common CEP operators. This approach was explored during the implementation of CEPSwift library.

The choice of which operators would be implemented in the library involved factors such as available time, implementation complexity and operator relevance. The classes of CEP operators listed in table 1 were analyzed and other libraries implementations were studied. Finally, it was implemented six operators: filter, followedBy, map, merge, window and max and min.

Two use cases of CEPSwift library were demonstrated in order to illustrate practical examples of the API usage. The same solution for the proposed problems in the use cases were written without the library and using only RxSwift. In both cases the code using only RxSwift was longer (in terms of lines of code) and consisted basically on boilerplate code.

6.2 Future work

Usability test and performance test

The proposed API is a tool that offers support for stream handling, event generation and complex event processing operations in Swift. Therefore, it is crucial to conduct usability tests in order to check if the library usage and methods are intuitive enough. These tests can show what aspects need to be improve and can guide the work that need to be done in new releases.

Complex event processing has real-time (or near real-time) requirements. Performance tests need to be conducted in order to verify if this requirement is being met. Other reactive libraries can be integrated to the project as support framework in order to compare with RxSwift performance.

New operators

Complex event processing has many classes of operators that make stream handling easier and helps to extract high-level situations, as described in table 1. Six operators were implemented but there is plans of implementing more in new releases.

Swift server-side support

Swift is an open-source language that is continuously improving and is widely used to build mobile applications that, in their nature, are event-oriented. But also, there is a great increase of interest by the industry in server applications built in Swift. The benefits of complex event processing are even bigger in server applications that often deal with a massive quantity of data and also requires high performance, in real-time (or near real-time). Therefore, changes need to be done in the API in order to add compatibility with Swift server-side and scalability is an important factor that should be focused and analyzed.

Use of design metrics to measure code quality

Design metrics such as Weighted Methods per Class (WMC), Coupling Between Object classes (CBO), Response For a Class (RFC) and Number of Children of a Class (NOC) are capable of measure the quality of an object-oriented code [13]. These metrics can be used to analyze CEPSwift software quality and also to indicate possible refactoring in new releases. Also, studies involving complexity and other metrics besides LOC can be conducted to compare the use case solutions that use CEPSwift and pure RxSwift to define CEP rules.

7. References

[1] PASCHKE, Adrian; KOZLENKOV, Alexander; BOLEY, Harold. A homogeneous reaction rule language for complex event processing. arXiv preprint arXiv:1008.0823, 2010.

[2] AGUILERA, Marcos K. et al. Matching events in a content-based subscription system. In: Proceedings of the eighteenth annual ACM symposium on Principles of distributed computing. ACM, 1999. p. 53-61.

[3] CUGOLA, Gianpaolo; MARGARA, Alessandro. Processing flows of information: From data stream to complex event processing. ACM Computing Surveys (CSUR), v. 44, n. 3, p. 15, 2012.

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

[5] ReactiveX, RxSwift Project. Retrieved November 28th 2017, from <https://github.com/ReactiveX/RxSwift>.

[6] Swift.org, Server APIs Project. Retrieved November 28th 2017, from <https://swift.org/server-apis/>.

[7] TEYMOURIAN, Kia; PASCHKE, Adrian. Enabling knowledge-based complex event processing. In: Proceedings of the 2010 EDBT/ICDT Workshops. ACM, 2010. p. 37.

[8] KOTA, Venkata Krishna et al. Secure Complex Event Processing Framework. In: Advance Computing Conference (IACC), 2017 IEEE 7th International. IEEE, 2017. p. 156-160.

[9] ROBINS, D. Complex event processing. In: Second International Workshop on Education Technology and Computer Science. Wuhan. 2010. p. 1-10.

[10] Luckham, David. 2002. The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems. Pearson Education, Inc.

[11] SABOOR, M.; RENGASAMY, R. Designing and developing Complex Event Processing Applications. Sapient Global Markets, 2013.

[12] ECKERT, Michael; BRY, François. Complex event processing (CEP). Informatik-Spektrum, v. 32, n. 2, p. 163-167, 2009.

[13] BASILI, Victor R.; BRIAND, Lionel C.. ; MELO, Walcélio L. A validation of object-oriented design metrics as quality indicators. IEEE Transactions on software engineering, v. 22, n. 10, p. 751-761, 1996.

[14] MICHELSON, Brenda M. Event-driven architecture overview. Patricia Seybold Group, v. 2, 2006.

Signatures

_____

George Belo Guedes
(Author)

_____

Kiev Santos da Gama
(Supervisor)