Universidade Federal de Pernambuco
Centro de Informática

Graduação em Ciência da Computação

Extensão e avaliação funcional da biblioteca CEPlin

Jonathan Gomes dos Santos

Trabalho de Graduação

Recife
Dezembro de 2018

Universidade Federal de Pernambuco
Centro de Informática

Jonathan Gomes dos Santos

Extensão e avaliação funcional 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 a obtenção do grau de Bacharel em Ciência da Computação.

Orientador*: Kiev Santos da Gama*

Recife
Dezembro de 2018

Agradecimentos

Um agradecimento especial à minha esposa, que me ajudou muito com sua compreensão todo o tempo, meus pais e minha irmã que me apoiaram durante o desenvolvimento deste trabalho. Agradeço aos amigos que estiveram sempre me encorajando. Agradeço também de forma especial a Kiev pela compreensão e por me guiar neste trabalho.

Resumo

Dada a presença de aplicações reativas estar em crescimento, a grande necessidade de bibliotecas para auxiliar os desenvolvedores neste fim gerou o surgimento de CEPlin, uma biblioteca em Kotlin para esse fim. Mas embora tenha surgido esta biblioteca, ela não foi completamente desenvolvida, o que mostrou a necessidade de extensão da mesma para a inclusão de novos operadores. Neste trabalho o objetivo foi implementar e validar uma parte importante e muito útil desses operadores: os operadores de controle de fluxo.

Palavras-chave: Programação Reativa, CEP, Programação Orientada a Eventos, Operadores de Controle de Fluxo, Testes.

Abstract

With the presence of reactive applications growing, the large need for libraries to assist developers in this regard has led to the creation of CEPlin, a framework written in Kotlin for this purpose. But although this library has been developed, it has not been fully built, which has shown the need to extend it to include new operators. In this work the objective was to implement and validate an important and very useful part of these operators: flow control operators.

Keywords: Reactive Programming, CEP, Event-Driven Programing, Flow Control Operators, Tests.

Lista de Figuras

Lista de Tabelas

Lista de Acrônimos

CEP	Complex Event Processing
RL	Reactive Language
RLs	Reactive Languages
IFP	Information Flow Processing
Rx	ReactiveX
FP	Functional Programming

Sumário

Capítulo 1

Introdução

O processamento de fluxo de informação tem sido uma necessidade cada vez mais comum nas aplicações que surgem a cada dia. Por conta disso ferramentas dos mais variados tipos passaram a aparecer com o intuito de suprir essa demanda, utilizando-se de modelos de processamento iguais ou muito semelhantes [1].

Entre as áreas surgidas por conta disso, veio a existir a de processamento de eventos complexos (Complex Event Processing, CEP). Em sistemas CEP, o funcionamento é dado pela filtragem e combinação de eventos com o intuito de fornecer às partes interessadas os resultados em um nível mais alto. Geralmente esses sistemas utilizam linguagens de consulta similares a SQL ou linguagens com encadeamento de funções, o que acaba se tornando pouco intuitivo [1].

Outro campo surgido das mesmas necessidades é o das Linguagens reativas (RLs), linguagens estas que tratam de valores que variam com o tempo [5], propagando a mudança de forma implícita. A grande vantagem de RLs é o fato de que as mesmas se integram ao ambiente de desenvolvimento, fornecendo recursos úteis ao desenvolvedor [2].

Dado isso, temos uma grande similaridade entre CEP e RLs em sua forma de processamento. O fluxo que ambas as abordagens seguem tem vários pontos em comum e cada uma possui uma vantagem interessante, o que levou à motivação inicial para essa pesquisa.

1.1 Motivação

O que motivou esse trabalho foi a necessidade de melhorias na primeira biblioteca em Kotlin [7] para CEP utilizando RLs, chamada CEPlin [5], com suporte ao sistema Android. A ferramenta é baseada nos princípios de CEP e escrita em Kotlin utilizando funcionalidades das bibliotecas *RxJava* [9] e *RxKotlin* [10], que são bibliotecas da família *ReactiveX* [8] que fornecem as funcionalidades de RLs, porém, a mesma possui limitações no tocante à quantidade de operadores disponíveis e a falta de testes dos operadores, o que poderia garantir seu funcionamento correto dos mesmos, algo fundamental a uma biblioteca deste tipo.

1.2 Objetivos

Tendo como base o que foi implementado na biblioteca CEPlin, pode ser notado que nem todos os operadores mais comuns foram implementados por limitação de tempo. Isso abriu margem para trabalhos futuros com relação a ela.

1.2.1 Objetivos gerais

Este trabalho se foca em uma parte do que foi deixado para o futuro em sua implementação, tendo como objetivos gerais:

1. Ampliar a expressividade da biblioteca, permitindo que ela possua mais funcionalidades e maior flexibilidade de uso.

2. Permitir testes simplificados dos operadores, simulando o aspecto temporal da emissão de eventos, para que tais operadores possam ser validados de forma simulada, diferente do que ocorre atualmente, onde a validação é totalmente manual.

1.2.2 Objetivos Específicos

Para Atingir os objetivos gerais, foram definidos objetivos específicos, para guiar melhor o desenvolvimento da implementação:

1. Desenvolver os operadores de gerenciamento de fluxo, pois ainda não existem na biblioteca e são operadores muito importantes em aplicações, pelo fato do fluxo de dados em aplicações ser algo muito importante nos mesmos.
2. Desenvolver um novo componente relacionado aos testes, que simule um ambiente real para a execução dos mesmos, de forma a garantir que o aspecto temporal seja respeitado e que permita a criação de testes repetíveis, evitando possíveis problemas do teste manual de operadores.
3. Efetuar testes dos operadores implementados utilizando a engine de testes desenvolvida, a fim de avaliar se os operadores funcionam de forma semelhante à teoria, segundo descrito em [1].

Capítulo 2

Fundamentação

Neste capítulo, teremos a introdução de conceitos importantes relativos a este trabalho no tocante a CEP, RLs e sua integração. Os operadores também serão discutidos e a linguagem utilizada. Por fim, teremos uma visão geral da biblioteca CEPlin, que será estendida, visando um maior entendimento dos conceitos da mesma e de como foi implementada.

2.1 Complex Event Processing

Tendo seu surgimento da década de 90, *Complex Event Processing (CEP)* nasceu da necessidade de gerenciamento de eventos dada a sua ocorrência em sistemas [4] com o objetivo de permitir o uso de lógicas mais complexas a partir de combinações entre esses eventos. Deste momento até os dias atuais vem sendo utilizado em diversas áreas como gerenciamento de regras de negócio em aplicações, aprendizagem de máquina e sistemas distribuídos, o que despertou o interesse e promoveu o surgimento de diversas ferramentas voltadas para esta área [6][10][13][14].

O funcionamento de sistemas CEP é semelhante ao de sistemas *Publisher-Subscriber*, onde os consumidores se inscrevem em canais para serem notificados da ocorrência de eventos de interesse. A grande vantagem no uso de CEP é que os eventos podem ser filtrados, compostos e encadeados para representar situações de interesse mais complexas, que melhor representem situações reais [1].

Esse encadeamento, composição e outros tipos de combinações e filtragens que podem ocorrer em CEP são resultado de regras predefinidas que recebem múltiplos eventos e emitem um evento único de resposta caso a regra seja satisfeita.

2.2 Fluxo de processamento CEP

O conceito de CEP está em um grande domínio chamado de Information Flow Processing (IFP), que possui conceitos em comum, como definido por Cugola em [1]. Nessa definição temos as etapas do IFP dadas como:

1. ***Signaling***, etapa onde os eventos são detectados.
2. ***Triggering***, etapa o evento é vinculado a uma regra específica.
3. ***Evaluation***, etapa em que o evento é avaliado de acordo com a regra à qual foi atrelado.
4. ***Scheduling***, etapa em que a ordem de execução das regras é definida.
5. ***Execution***, etapa em que os eventos executam de fato.

Vale salientar que as etapas mostradas anteriormente permitem uma execução cíclica, ou seja, dada a execução de um evento, este poderá gerar um novo evento como resultado, evento este que entra novamente no fluxo, reiniciando o ciclo.

2.3 Reactive Language

Reactive Language (RL) ou linguagem reativa é um tipo de linguagem de programação de permite a manipulação de *Streams*, fornecendo uma abstração maior no tocante à linguagem e deixando o programador mais livre para focar nas regras de negócio e na lógica do sistema. Linguagens deste tipo servem como ferramentas de propagação de mudanças sofridas por dados ao longo do tempo.

Este tipo de linguagem surgiu a partir da necessidade de aplicações com código mais enxuto, conciso e focado na lógica, para desenvolvimento de aplicações de baixo tempo de resposta, o que culminou no *Reactive Manifesto* [3]. Dado isso, surgiu uma API para padronizar as operações desta natureza que fosse comum a múltiplas linguagens, servido de

referência para o desenvolvimento de bibliotecas com essa finalidade. E assim surgiu *ReactiveX* (Rx) [8], provendo reatividade em múltiplas linguagens utilizando o padrão Observer como base.

2.4 Relação CEP - RL

Rls e CEP possuem várias similaridades, como ressaltado em [2], o que mostrou a possibilidade do desenvolvimento de ferramentas que utilizam-se de ambos para o desenvolvimento de sistemas e aplicações reativas [1][6].

Comparando os dois domínios [2][5], podemos observar as fases envolvidas no processamento e sua relação com o que foi analisado anteriormente para CEP neste capítulo.

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

Figura 2.1- Comparação entre fases de processamento CEP e Rls [5]

A Figura 2.1, demonstra, de forma sucinta, as similaridades entre as duas abordagens e fica clara a relação entre ambas no tocante ao suporte que pode ser fornecido a aplicações reativas. CEP trabalha sobre eventos, processando-os, gerando novos eventos como resultado e re-encaminhados tais novos eventos novamente ao fluxo de processamento, o que permite a composição de seu próprio fluxo para a avaliação de regras mais complexas. Similarmente RLs trabalham sobre a emissão de valores para realização de filtragens e combinações mais complexas utilizando-se do padrão *Observable* para permitir a geração de novos fluxos a partir de tal emissão de valor e com isso a possibilidade de composição da mesma forma que CEP [2].

Além disso, ambas as abordagens permitem processamento assíncrono e trabalham com dados de múltiplas origens, o que é muito útil em aplicações reativas.

2.5 Programação Funcional

A Programação Funcional (Functional Programming ou FP) surgiu na década de 50, e desde então teve sua base em ideias da década de 30, como o *lambda cauculus*. É um paradigma de programação diferente de outros, pois se baseia na avaliação de funções matemáticas, a fim de evitar estados mutáveis em contraste com a programação imperativa [15].

Ao longo do tempo linguagens funcionais como *Lisp*, *Erlang* e *Haskell* surgiram e foram sendo utilizadas por programadores, sendo estas, linguagens puramente funcionais. Atualmente várias linguagens se utilizam deste paradigma, principalmente as mais recentes, muitas das quais vem surgindo como *linguagens multiparadigma*, como no caso da linguagem Kotlin [7].

2.6 Biblioteca CEPlin

Este trabalho tem como foco a extensão de uma biblioteca preexistente, a biblioteca CEPlin [5], e para tal é necessário um entendimento da mesma para que o trabalho de ampliação de suas possibilidades seja possível.

A biblioteca em questão foi a primeira escrita em Kotlin, linguagem que vem tendo um grande crescimento em contribuições da comunidade [12] e é a linguagem oficial adotada pelo Google para o desenvolvimento de aplicações para o sistema Android, o que torna o desenvolvimento de ferramentas nesta linguagem cada vez mais importante.

Outro aspecto que a torna uma linguagem poderosa é o fato de ser uma linguagem que consegue trabalhar com vários paradigmas, como FP, o que é mais uma vantagem para a biblioteca no tocante à sua versatilidade de uso.

Seu desenvolvimento teve o intuito de prover o poder de programação reativa de forma simplificada com o uso de CEP ao sistema Android. Para isso, foi desenvolvida sobre bibliotecas de RLs preexistentes nas linguagens Java e Kotlin, mais especificamente as da

família *ReactiveX* [8] (*RxJava* e *RxKotlin* [9][10]) que já implementam os princípios de RLs, deixando para CEPlin a implementação dos operadores CEP.

Em comparação com outras bibliotecas existentes, CEPlin possui diferenças pelo uso de RL e FP, e pelo fato de ser voltada a aplicações móveis para o sistema Android. Possui muitas semelhanças se equiparando à existente CEPSwift [6], porém, a mesma foi desenvolvida para a linguagem Swift e o sistema iOS. No mundo Java, podemos ressaltar Apache Flink CEP [13], que utiliza FP, mas não RL e Asper [14], que não se utliza de FP ou RL, enquanto CEPlin utiliza-se de ambas, obtendo ganho em abstração de código. De todas as citadas, apenas Asper é compatível com o sistema Android, foco de CEPlin, o que mostra a grande necessidade de desenvolvimento em bibliotecas para esse sistema.

A biblioteca possui alguns componentes principais que possibilitam seu funcionamento [5], onde cada um desses possui sua própria função, de forma a complementarem-se para o funcionamento da biblioteca:

1. ***Event***: Entidade que representa eventos CEP. É a unidade básica utilizada no processamento dos eventos na biblioteca.
2. ***EventManager***: Classe que gerencia a os eventos, permitindo sua adição e obtenção do *Stream* de eventos que ela gerencia, permitindo controle sobre *Streams* importantes utilizados em aplicações CEP..
3. ***EventStream***: Representa o Stream de ocorrência de eventos. É onde os operadores estão implementados e onde ocorre a execução dos operadores. Por ser o principal componente da biblioteca, provém uma interface de utilização dos operadores implementados para a aplicação dos mesmos sobre *Streams* e geração de novos *Streams* de acordo com o operador utilizado.
4. ***ComplexEvent***: Classe utilizada para mesclagem de eventos de tipos diferentes, gerando *Streams* que podem ser re-inseridos ao fluxo de processamento CEP.

Embora tenha muito trabalho já realizado, a biblioteca teve seu escopo reduzido por questão de tempo, o que não permitiu o desenvolvimento de todos os operadores CEP definidos por Cugola e Margara [1], o que abriu margem para trabalhos futuros e serviu de

motivação para este trabalho, com o objetivo de melhorar a biblioteca aumentando a quantidade de operadores implementados e, portanto, seu poder e expressividade.

2.7 Testes em CEPlin

Dada a necessidade de validação do desenvolvimento, a realização de testes se torna imprescindível, o que é uma grande defasagem da biblioteca CEPlin [5]. Atualmente para a execução de testes na mesma é necessário criar toda a estrutura necessária manualmente, o que se torna cansativo, repetitivo e improdutivo. Além disso, os testes se tornam bastante complexos para o entendimento.

Outro problema é o aspecto temporal das operações CEP, o que acaba por dificultar os testes pela falta de formas simples de gerar dados para os testes que respeitem este aspecto durante a execução. Dados estes aspectos, uma outra necessidade da biblioteca foi descoberta na necessidade de um mecanismo que facilite os testes dos operadores.

2.8 Proposta

A proposta deste trabalho é aumentar a funcionalidade da biblioteca CEPlin, implementando operadores importantes e ainda não existentes na mesma e validar o trabalho com a utilização de testes dos operadores implementados em streams, visto que, dado o caráter temporal do processamento de eventos complexos, testes unitários simples não são suficientes. Com isso teremos os novos operadores implementados e testados de modo que sejam tidos como uma extensão válida da biblioteca CEPlin, incrementando suas possibilidades como desejado.

Estes operadores serão desenvolvidos seguindo a arquitetura proposta em [1], mantendo assim a compatibilidade com os operadores preexistentes. Para a realização de testes, será desenvolvido o componente *EventStreamSimulator*. O componente permitirá a simulação da emissão de eventos a partir de listagens simples, facilitando a criação de testes, além disso, a ordem das listas utilizadas poderá ser manipulada para refletir o aspecto

temporal das emissões de cada evento, permitindo simulações mais precisas em relação a este aspecto.

Dadas estas necessidades, o desenvolvimento segue pela criação dos operadores, que serão desenvolvidos onde já existem os demais dento do código da biblioteca, o *EventStreamSimulator* será desenvolvido como um novo módulo e os testes serão desenvolvidos utilizando o simulador recém criado em um arquivo separado, com testes repetíveis, podendo ser alterados e permitindo a repetição dos mesmos de forma exata a qualquer momento posteriormente.

Capítulo 3

Implementação

Neste capítulo, temos a descrição de como foi feita a implementação do que foi proposto neste trabalho, visando deixar clara a forma como foi pensado e implementado cada operador e a descrição da engine de testes e casos testados para a validação dos novos operadores.

3.1 Mapeamento de Operadores

Para dar início ao trabalho, foi necessário visualizar o conjunto de operadores a serem implementados e definir suas assinaturas antes do desenvolvimento dos mesmos. O subconjunto dos operadores escolhido foi o *Flow Management Operators*, que operam no gerenciamento do fluxo dos eventos, não modificando os mesmos, apenas trabalhando sobre a forma que o fluxo de eventos funciona.

Tais operadores são muito necessários para operações de ordenação e filtragem, algo muito comum em aplicações de todos os tipos, o que levou à escolha dos mesmos. Outro fator importante na escolha deste subconjunto de operadores é o fato de que outros também comuns em aplicações já estavam implementados na biblioteca, levando à escolha dos mais importantes ainda não implementados na mesma para desenvolvimento neste trabalho.

Na Tabela 3.1 podemos ver os operadores do subconjunto escolhido e algumas informações sobre os mesmos na vertente teórica [1] e sua nomenclatura existente ou escolhida para desenvolvimento na biblioteca CEPlin [5].

Tipo	Operador Teórico	Operador CEPlin	Implementação
Flow Management	Join	merge(stream:)	Existente em CEPlin
	Union	union(stream:)	Existente em CEPlin
	Except	not(stream:)	A ser implementado
	Intersect	intersect(stream:)	A ser implementado
	Remove-Duplicate	distinct()	A ser implementado
	Duplicate	-	-
	Order By	orderBy(comparison:)	A ser implementado
	Group By	groupBy(comparison:)	A ser implementado

Tabela 3.1 - Operadores de fluxo CEP

A operação de *Duplicate* acabou não tendo necessidade de implementação pelas propriedades inerentes à linguagem *Kotlin*. O fato do *EventStream* da biblioteca ser armazenado em uma variável já permite a duplicação da mesma, o que torna o operador desnecessário. Os operadores Join e Union já foram implementados na versão inicial de CEPlin, o que implica na falta de necessidade de implementação neste caso.

3.2 Implementação de Operadores

Na versão inicial da biblioteca CEPlin [5] os operadores *Join* e *Union* já haviam sido implementados e, como explicado anteriormente, não existe a necessidade do operador *Duplicate* por propriedades inerentes à linguagem de programação escolhida, após uma revisão dos operadores existentes, foi percebida a existência de alguns operadores de fluxo, o que deixa apenas os operadores presentes na Tabela 3.2 a cargo da atual implementação.

Tipo	Operador Teórico	Operador CEPlin	Tipo de Implementação
Flow Management	Except	not(stream:)	Implementação Completa
	Intersect	intersect(stream:)	Implementação Completa
	Remove-Duplicate	distinct()	*Wrap RxJava* [9]
	Order By	orderBy(comparison:)	Implementação Completa
	Group By	groupBy(comparison:)	Implementação Completa

Tabela 3.2 - Operadores de fluxo CEP implementados

Na sequência deste capítulo, temos as descrições de todos o operadores implementados, incluindo figuras referentes a seus respectivos funcionamentos. Cada figura poderá conter de 2 a 3 linhas, cada uma representando um *EventStream* e um retângulo nomeado representando o operador executado. As linhas estarão descritas como:

1. ***linha A***: Representa o *Stream* de onde a operação é executada. É o primeiro *Stream* a ser considerado pelo operador.
2. ***linha B***: Representa o *Stream* que é recebido como parâmetro por operadores CEP implementados. É o Stream considerado para a comparação com o *Stream* da ***linha A*** em operadores que trabalham com dois *Streams*.
3. ***linha C***: Representa o *Stream* retornado pelo operador. É a resposta das funções implementadas na biblioteca e o que deve ter similaridade com a resposta prevista no operador teórico [1].

3.2.1 Operador *not(stream:)*

Este operador equivale ao operador *Except* definido por Cugola [1]. Assim como outros operadores de fluxo, este se comporta de maneira similar a operações sobre conjuntos.

O operador compara dois *EventStreams*, o em que o operador foi executado e o passado como parâmetro e retorna a diferença entre os dois como um novo *EventStream*. Esta diferença segue exatamente a mesma noção da diferença entre conjuntos, retornando apenas os eventos presentes no *Stream* que executa o operador e não no que foi passado como parâmetro. O funcionamento do operador pode ser visualizado na Figura 3.1.

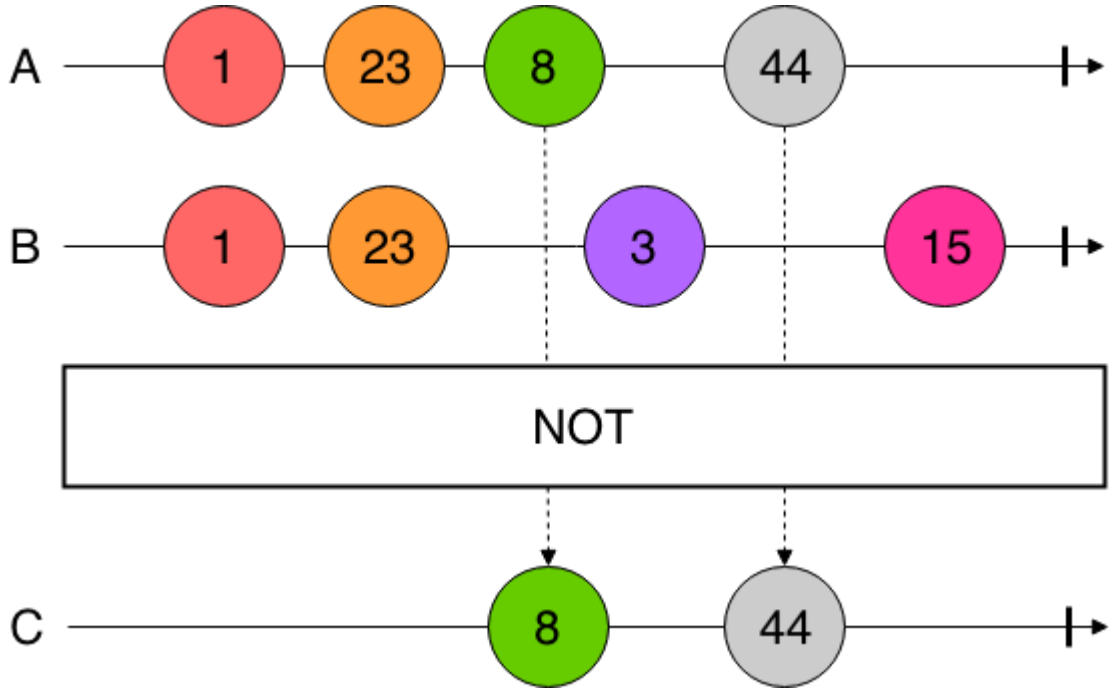

Figura 3.1 - Funcionamento do operador *not(stream:)*

Esta abordagem permite que, pelo fato de ser gerado um *EventStream* como retorno do operador, ele possa ser encadeado com outras operações e, pelo fato de ser um novo *Stream* e não uma modificação do original, não existe problemas de perda de dados além de permitir a possibilidade de executar operações entre o *Stream* original e o gerado como retorno do operador. A implementação do operador pode ser encontrada no Apêndice A.1.

O operador combina dois *EventStreams* de modo que sempre que um evento é emitido pelo primeiro, ele é combinado com os eventos emitidos pelo segundo através do operador *withLatestFrom* presente na biblioteca *RxKotlin*. Isso permite que a presença do evento seja verificada no segundo *Stream*, fazendo com que o mesmo possa ser removido do *Stream* final caso exista em ambos e força a termos apenas os existentes no primeiro *Stream* somente.

3.2.2 Operador *intersect(stream:)*

Este operador equivale ao operador *Intersect* definido por Cugola [1]. Assim como outros operadores de fluxo, este se comporta de maneira similar a operações sobre conjuntos.

O operador compara dois *EventStreams*, o em que o operador foi executado e o passado como parâmetro e retorna a intersecção entre os dois como um novo *EventStream*. Esta diferença segue exatamente a mesma noção da intersecção entre conjuntos, retornando apenas os eventos presentes em ambos os *Streams* e removendo os eventos duplicados. O funcionamento do operador pode ser visualizado na Figura 3.2.

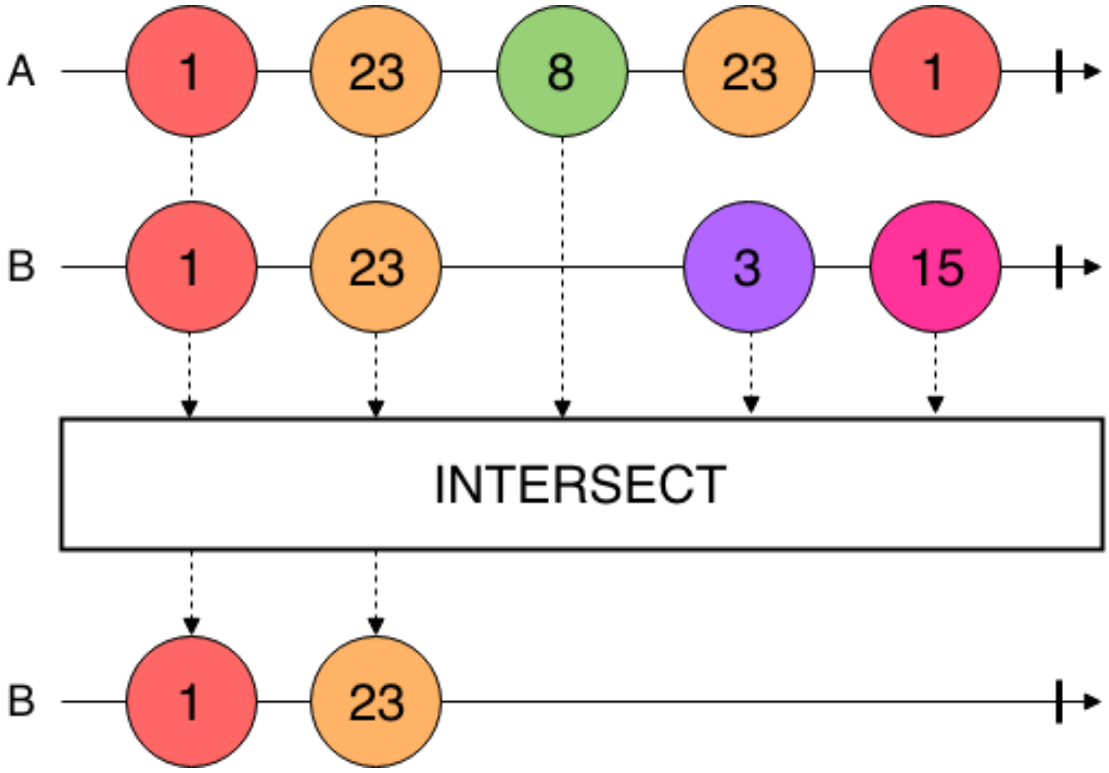

Figura 3.2 - funcionamento do operador *intersect(stream:)*

Esta abordagem permite que, pelo fato de ser gerado um *EventStream* como retorno do operador, ele possa ser encadeado com outras operações e, pelo fato de ser um novo *Stream* e não uma modificação do original, não existe problemas de perda de dados além de permitir a possibilidade de executar operações entre o *Stream* original e o gerado como retorno do operador. A implementação do operador pode ser encontrada no Apêndice A.2.

3.2.3 Operador *distinct()*

Este operador equivale ao operador *Remove-Duplicate* definido por Cugola [1]. O operador retorna apenas uma instância de cada evento, como encontrado similarmente em linguagens de consulta tal qual SQL.

O operador verifica os eventos existentes em um *EventStream* e retorna apenas um instância de cada evento encontrado como um novo *EventStream*. O funcionamento do operador pode ser visualizado na Figura 3.3.

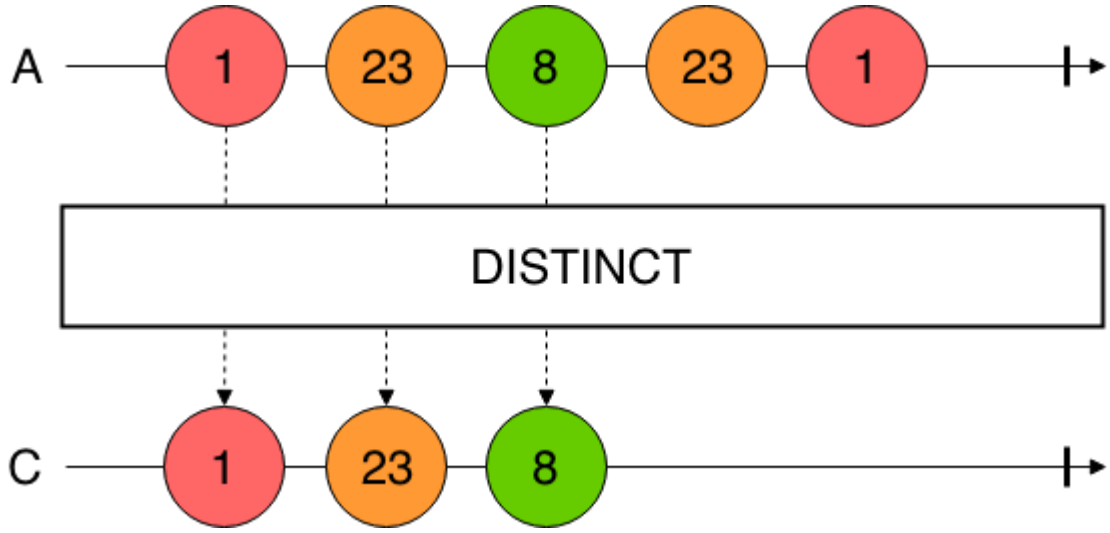

Figura 3.3 - Funcionamento do operador *distinct()*

Esta abordagem permite que, pelo fato de ser gerado um *EventStream* como retorno do operador, ele possa ser encadeado com outras operações e, pelo fato de ser um novo *Stream* e não uma modificação do original, não existe problemas de perda de dados além de permitir a possibilidade de executar operações entre o *Stream* original e o gerado como retorno do operador. A implementação do operador pode ser encontrada no Apêndice A.3.

O operador foi implementado como um *Wrap* do método de mesmo nome existente em RxJava, pois a implementação existente funciona corretamente e esta abordagem pôde explicitar o operador CEP correspondente na biblioteca CEPlin [5].

3.2.4 Operador *orderBy(comparison:)*

Este operador equivale ao operador *Order By* definido por Cugola [1]. O operador retorna os eventos ordenados por um critério de comparação, como encontrado similarmente em linguagens de consulta tal qual SQL.

O operador verifica os eventos existentes em um *EventStream* e processa o critério de comparação passado por parâmetro, retornando os eventos na ordem definida pelo comparador como um novo *EventStream* que possui a listagem ordenada dos eventos. O funcionamento do operador pode ser visualizado na Figura 3.4.

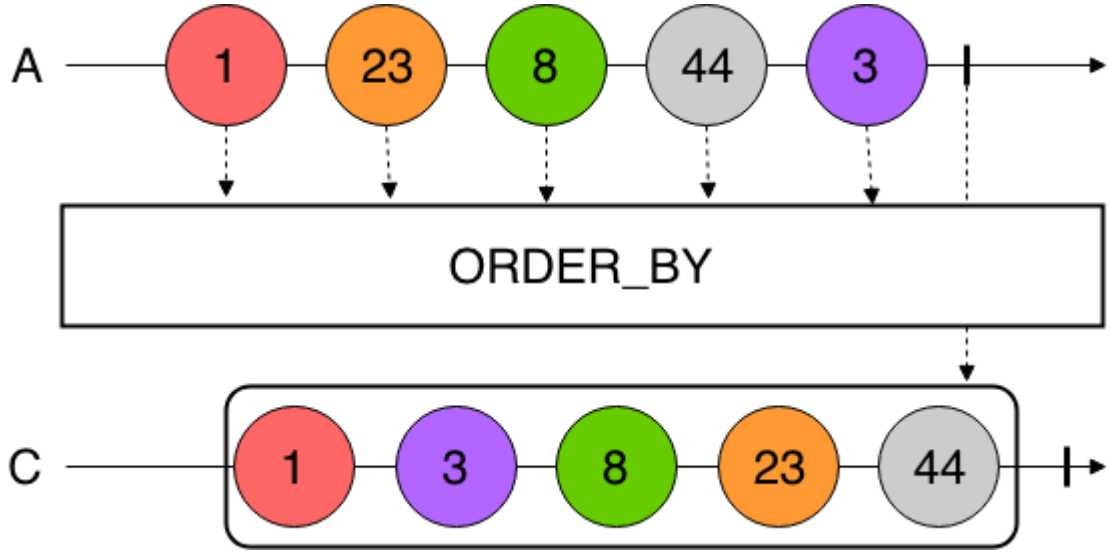

Figura 3.4 - funcionamento do operador *orderBy(comparison:)*

Esta abordagem permite que, pelo fato de ser gerado um *EventStream* como retorno do operador, ele possa ser encadeado com outras operações e, pelo fato de ser um novo *Stream* e não uma modificação do original, não existe problemas de perda de dados além de permitir a possibilidade de executar operações entre o *Stream* original e o gerado como retorno do operador. A implementação do operador pode ser encontrada no Apêndice A.4.

3.2.5 Operador *groupBy(comparison:)*

Este operador equivale ao operador *Group By* definido por Cugola [1]. O operador retorna os eventos agrupados por um critério de comparação, como encontrado similarmente em linguagens de consulta tal qual SQL.

O operador verifica os eventos existentes em um *EventStream* e processa o critério de comparação passado por parâmetro, agrupando os eventos da forma definida pela função de comparação como um novo *EventStream*. O funcionamento do operador pode ser visualizado na Figura 3.5.

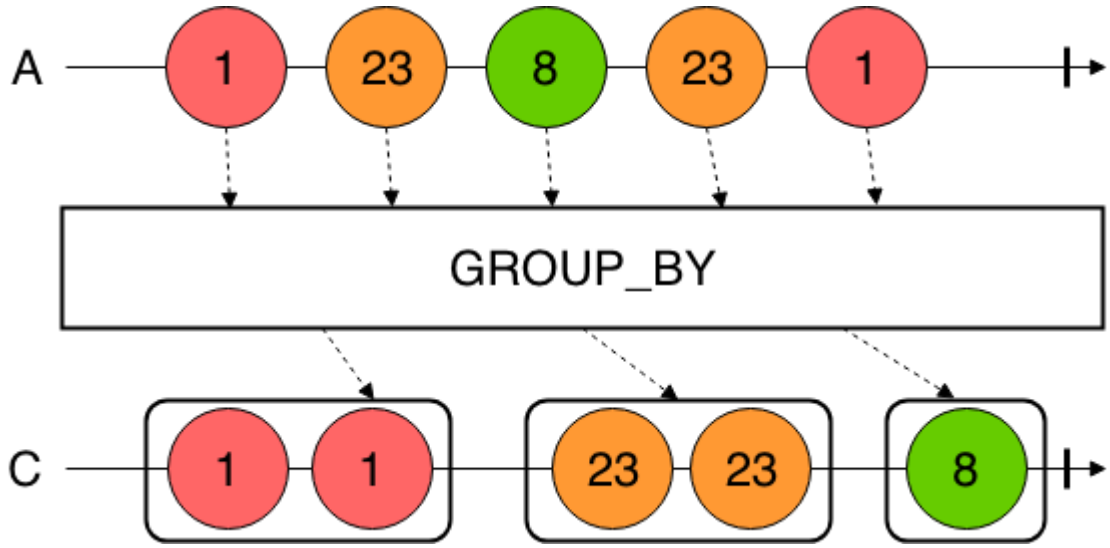

Figura 3.5 - funcionamento do operador *groupBy(comparison:)*

Esta abordagem permite que, pelo fato de ser gerado um *EventStream* como retorno do operador, ele possa ser encadeado com outras operações e, pelo fato de ser um novo *Stream* e não uma modificação do original, não existe problemas de perda de dados além de permitir a possibilidade de executar operações entre o *Stream* original e o gerado como retorno do operador. A implementação do operador pode ser encontrada no Apêndice A.5.

3.3 Engine de testes

Para a avaliação dos operadores criados foi desenvolvido um componente que auxilia na criação de testes, com o intuito de facilitar o teste dos mesmos levando em conta o aspecto temporal das operações CEP.

O componente chamado *EventStreamSimulator* foi desenvolvido para este fim, e auxiliou posteriormente nos testes dos operadores. Ele se utiliza do componente *EventManager* existente na biblioteca para gerenciar as emissões de eventos, simulando o que ocorre na realidade em aplicações Android.

3.3.1 EventStreamSimulator

O EventStreamSimulator é composto por funções de simulação de recebem os parâmetros necessários para simular a ocorrência dos eventos, e possuindo variações das funções para simular diferentes tipos de operadores. As funções desenvolvidas foram:

1. ***simulate***: Esta tem três variantes, para simulação de Operações com diferentes tipos e quantidades de parâmetros e respostas esperadas.
   1.1. Operações sobre um único *Stream*, como *distinct()*
   1.2. Operações sobre dois *Streams*, como intersect(stream:)
   1.3. Operações sobre um *Stream* e uma função, como *groupBy*(comparison:)
2. ***simulateCompare***: Esta função foi feita com intuito de facilitar o retorno da simulação de operações entre um *Stream* e um *Comparable*, como no caso do operador *orderBy*(comparison:)

O fluxo da simulação em testes ocorre como demonstrado na Figura 3.6.

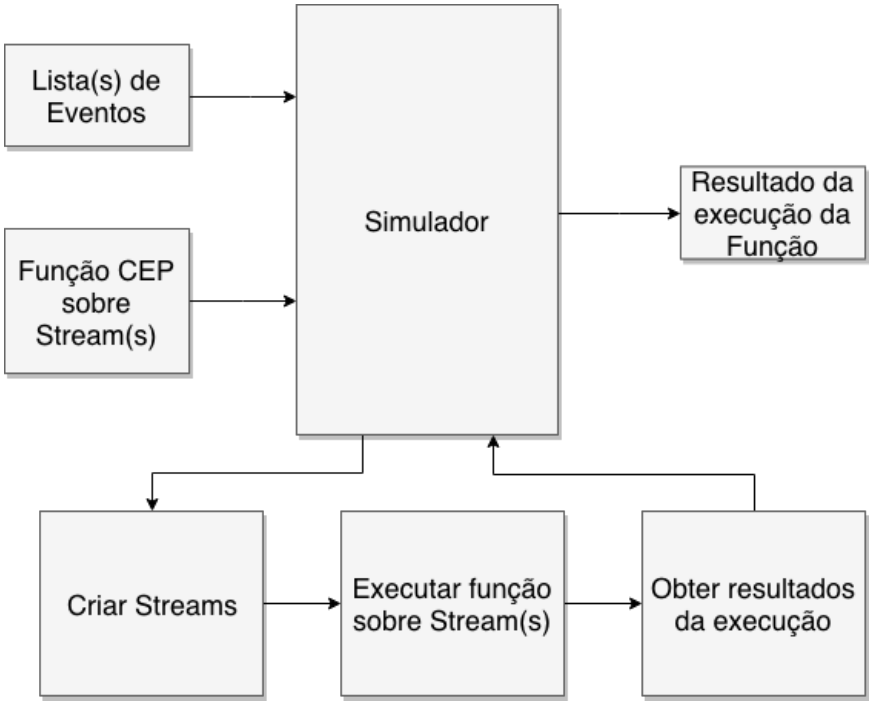

Figura 3.6 - Fluxo de funcionamento do *EventStreamSimulator*

Como explicitado na Figura 3.6, após receber os eventos e a função passados pelo teste implementado, o simulador cria Streams a partir do eventos, executa a função que gerencia estes Streams e obtém a partir destes, os resultados do processamento para que possam ser visualizados pelo teste facilmente em forma de listas ou mapas simples e possam ser comparados com o resultado esperado sem necessidade de operações complexas. O código do simulador pode ser encontrado no Apêndice B.1.

Capítulo 4

Testes e Avaliação

Para fazermos a avaliação do trabalho desenvolvido, vamos verificar se o que foi feito atende aos objetivos específicos descritos na sessão 1.2.2 deste documento, onde pode-se notar a necessidade de três partes do desenvolvimento:

1. Criação dos operadores, junto aos preexistentes no mesmo local.
2. Criação da engine de testes, como um novo componente que passará a estar disponível juntamente à biblioteca como parte integrante da mesma.
3. Criação de testes dos operadores, utilizando a engine de testes desenvolvida para simular o aspecto temporal sem a necessidade de testes manuais.

Dados estes aspectos, a avaliação pode ser dada a partir dos testes realizados, de forma a comparar seus resultados com o esperado de acordo com o funcionamento teórico dos operadores [1] da seguinte forma:

1. **Executar testes desenvolvidos**. Os testes consistem em criar entradas para a simulação e uma função sobre um operador a ser validado. Também no teste definimos a saída esperada de acordo com o que ocorreria na descrição teórica do operador [1]. Ao final, os testes verificam se a saída da simulação condiz com a expectativa.
2. **Verificar resultados dos testes**. O resultado de cada teste deve retornar sucesso, mostrando que condição testada de acordo com a expectativa definida na teoria é reproduzida corretamente no operador testado.

Os testes foram realizados utilizando a biblioteca de testes padrão do Android, a JUnit 4 [11], que junto com o novo componente se mostrou eficaz na realização dos testes. Foram

testados os operadores de fluxo implementados neste trabalho, com o intuito de garantir seu funcionamento de forma correta.

A realização dos testes foi feita de forma simples, passando eventos e funções ao simulador e comparando os resultados com um padrão esperado. O fluxo da execução dos testes foi idêntico ao que é comumente utilizado em ferramentas de teste, o que consiste em *setup* de dados, execução de testes e asserção de resultados, comparando a saída do simulador com o resultado esperado previamente descrito na fase de *setup*.

Como os operadores de fluxo não modificam os eventos do *Stream*, o tipo de evento utilizado nos testes se tornou indiferente, o que levou à escolha de eventos mais simples para facilitar o desenvolvimento de testes. Os eventos utilizados foram do tipo *IntEvent*, já existentes na biblioteca CEplin [5]. tais eventos emitem valores inteiros e o evento pode ser comparado com outro através deste valor.

Os testes foram feitos com o intuito de validar os novos operadores, portanto, a delimitação de tal escopo foi necessária visando a criação dos testes de forma concisa. Após a delimitação da metodologia a ser utilizada nos testes e dos componentes que seriam utilizados nos mesmos, juntamente com o *EventStreamSimulator* recém implementado, foram feitos os casos de testes, que foram desenvolvidos segundo o que foi previamente definido.

```
class IntEvent(val value: Int) : Event, Comparable<IntEvent> {

    override val timeStamp = Date()

    override fun compareTo(other: IntEvent): Int {
        return this.value.compareTo(other.value)
    }

    override fun equals(other: Any?): Boolean {
        if (other is IntEvent) {
            return this.value == other?.value
        }
        return super.equals(other)
    }

    override fun hashCode(): Int {
        return value!!.hashCode()
    }

    override fun toString(): String {
        return value.toString()
    }
}
```

Figura 4.1 - *IntEvent* presente na biblioteca CEPlin

Como mostrado na Figura 4.1, o evento do tipo *IntEvent* existente na biblioteca já supriu as necessidades de teste, o que dispensou o desenvolvimento de novos tipos de eventos para o uso específico nos testes da mesma.

4.1 Testes desenvolvidos

Esta seção descreve os testes desenvolvidos segundo o que foi delimitado anteriormente. Os testes foram feitos para os novos operadores, e serão descritos a seguir.

4.1.1 Operador *not(stream:)*

O operador *not(stream:)*, que é a implementação em CEPlin [5] do operador *Except* [1], como descrito nas Tabelas 3.1 e 3.2, é um operador que recebe um parâmetro, que é um segundo *Stream* para comparação. Ele opera sobre o próprio *Stream* que o executa e, usando o recebido como parâmetro, retorna um novo *Stream* contendo apenas os eventos existentes somente no *Stream* que executou a operação e não no passado como parâmetro, ou seja, caso um evento com as mesmas características seja encontrado no segundo *Stream*, ao executar este operador, o *Stream* retornado não o conterá. O funcionamento do operador pode ser encontrado na seção 3.2.1, no Capítulo 3 deste documento, e a implementação do teste executado pode ser encontrada a seguir.

```
fun exceptEvents() {
    val events1 = listOf(
            IntEvent(10),
            IntEvent(10),
            IntEvent(5),
            IntEvent(12),
            IntEvent(12)
    )

    val events2 = listOf(
            IntEvent(10),
            IntEvent(10),
            IntEvent(12),
            IntEvent(12)
    )
```

```
    val expected = listOf(
            IntEvent(5)
    )

    val simulator = EventStreamSimulator<IntEvent>()
    val output = simulator.simulate(events1, events2, ::exceptFunction)

    assert(expected == output)
}

private fun exceptFunction(stream1: EventStream<IntEvent>, stream2:
EventStream<IntEvent>): EventStream<IntEvent> {
    return stream1.not(stream2)
}
```

4.1.2 Operador *intersect(stream:)*

O operador *intersect(stream:)*, que é a implementação em CEPlin [5] do operador *Intersect* [1], como descrito nas Tabelas 3.1 e 3.2, é um operador que recebe um parâmetro, que é um segundo *Stream* para comparação. Ele opera sobre o próprio *Stream* que o executa e, usando o recebido como parâmetro, retorna um novo *Stream* contendo apenas os eventos existentes em ambos, ou seja, caso um evento com as mesmas características não seja encontrado no segundo *Stream*, ao executar este operador, o *Stream* retornado não o conterá. Além disso ele também retorna apenas uma instância de cada evento, o que implica em remoção de eventos duplicados existentes. O funcionamento do operador pode ser encontrado na seção 3.2.2, no Capítulo 3 deste documento, e a implementação do teste executado pode ser encontrada a seguir.

```
fun intersectEvents() {
    val events1 = listOf(
            IntEvent(10),
            IntEvent(10),
            IntEvent(5),
            IntEvent(12),
            IntEvent(12)
    )
```

```
    val events2 = listOf(
            IntEvent(10),
            IntEvent(10),
            IntEvent(12),
            IntEvent(12)
    )

    val expected = listOf(
            IntEvent(10),
            IntEvent(12)
    )

    val simulator = EventStreamSimulator<IntEvent>()
    val output = simulator.simulate(events1, events2, ::intersectFunction)

    assert(expected == output)
}

private fun intersectFunction(stream1: EventStream<IntEvent>, stream2:
EventStream<IntEvent>): EventStream<IntEvent> {
    return stream1.intersect(stream2)
}
```

4.1.3 Operador distinct()

O operador *distinct()*, que é a implementação em CEPlin [5] do operador *Remove-Duplicates* [1], como descrito nas Tabelas 3.1 e 3.2, é um operador que não recebe parâmetros. Ele opera sobre o próprio *Stream* que o executa e retorna um novo *Stream* contendo apenas os eventos não repetidos, ou seja, caso mais de um evento com as mesmas características seja encontrado, ao executar este operador, o *Stream* retornado conterá apenas uma instância do mesmo. O funcionamento do operador pode ser encontrado na seção 3.2.3, no Capítulo 3 deste documento, e a implementação do teste executado pode ser encontrada a seguir.

```
fun distinctEvents() {
    val events = listOf(
            IntEvent(10),
            IntEvent(10),
            IntEvent(5),
            IntEvent(12),
            IntEvent(12)
    )

    val expected = listOf(
            IntEvent(10),
            IntEvent(5),
            IntEvent(12)
    )

    val simulator = EventStreamSimulator<IntEvent>()
    val output = simulator.simulate(events, ::distinctFunction)

    assert(expected == output)
 }

private fun distinctFunction(stream: EventStream<IntEvent>): EventStream<IntEvent> {
     return stream.distinct()
}
```

4.1.4 Operador *orderBy(comparison:)*

O operador *orderBy(comparison:)*, que é a implementação em CEPlin [5] do operador *Order By* [1], como descrito nas Tabelas 3.1 e 3.2, é um operador que recebe um parâmetro, que é uma função de comparação. Ele opera sobre o próprio *Stream* que o executa e, usando a função recebida como parâmetro, retorna um novo *Stream* contendo os eventos ordenados segundo o critério de ordenação passado. O funcionamento do operador pode ser encontrado na seção 3.2.4, no Capítulo 3 deste documento, e a implementação do teste executado pode ser encontrada a seguir.

```
fun orderEvents() {
    val events = listOf(
            IntEvent(10),
            IntEvent(11),
            IntEvent(5),
            IntEvent(1),
            IntEvent(12)
    )

    val expected = listOf(
            IntEvent(1),
            IntEvent(5),
            IntEvent(10),
            IntEvent(11),
            IntEvent(12)
    )

    val simulator = EventStreamSimulator<IntEvent>()
    val output = simulator.simulateCompare(events, ::orderByFunction)

    assert(expected == output)
}

private fun orderByFunction(stream: EventStream<IntEvent>):
EventStream<List<IntEvent>> {
     return stream.orderBy { event ->
         event.value
     }
}
```

4.1.5 Operador *groupBy(comparison:)*

O operador *groupBy(comparison:)*, que é a implementação em CEPlin [5] do operador *Group By* [1], como descrito nas Tabelas 3.1 e 3.2, é um operador que recebe um parâmetro, que é uma função de comparação. Ele opera sobre o próprio *Stream* que o executa e, usando a função recebida como parâmetro, retorna um novo *Stream* contendo os eventos agrupados segundo o critério de agrupamento passado, retornando o mapeamento entre o critério utilizado e uma listagem de eventos que obedecem tal critério. O funcionamento do

operador pode ser encontrado na seção 3.2.5, no Capítulo 3 deste documento, e a implementação do teste executado pode ser encontrada a seguir.

```
fun groupEvents() {
    val events = listOf(
            IntEvent(10),
            IntEvent(10),
            IntEvent(5),
            IntEvent(12),
            IntEvent(12)
    )

    val expected = mapOf<Int, List<IntEvent>>(
            Pair(10, listOf(
                    IntEvent(10),
                    IntEvent(10)
            )),
            Pair(5, listOf(
                    IntEvent(5)
            )),
            Pair(12, listOf(
                    IntEvent(12),
                    IntEvent(12)
            ))
    )

    val simulator = EventStreamSimulator<IntEvent>()
    val output = simulator.simulate(events, ::groupByFunction)

    assert(expected == output)
}

private fun groupByFunction(stream: EventStream<IntEvent>): EventStream<Map<Int,
List<IntEvent>>> {
     return stream.groupBy { event ->
         event.value
     }
}
```

4.2 Resultados obtidos

Após o desenvolvimento dos operadores, da engine de testes e dos testes a serem realizados, a execução dos mesmos foi feita, a fim de avaliar possíveis problemas em quaisquer dos componentes supracitados.

Como os testes criados dizem respeito apenas aos novos operadores, não se pode afirmar a corretude ou desempenho dos demais, no entanto, os testes dos operadores de fluxo implementados ocorreram com sucesso, testando os casos propostos sem quaisquer erros ou falhas, demonstrando que os mesmos funcionam conforme o que foi proposto.

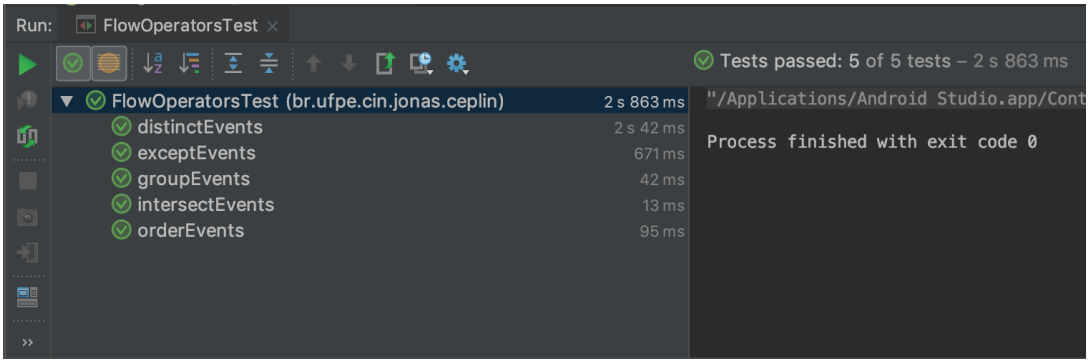

Figura 4.2 - Execução dos testes dos operadores de fluxo

Embora os testes realizados tenham sido focados nos novos operadores, a necessidade de implementação de testes mais completos, que possam cobrir todos os operadores já existentes na biblioteca é grande, tendo em vista a necessidade de validação dos mesmos, o que poderá garantir que todos estejam corretos ou mostrar as falhas existentes que devem ser corrigidas em futuras versões.

Capítulo 5

Conclusão e Trabalhos Futuros

5.1 Conclusão

Neste trabalho, a biblioteca *CEPlin* [5] foi estendida para incluir operadores de controle de fluxo ainda inexistentes. O trabalho de verificação de similaridades entre os operadores preexistentes das bibliotecas *RxJava* e *RxKotlin* [9][10], nas quais a biblioteca se baseia foi o início para chegar à implementação. A partir disso e do estudo do funcionamento dos operadores de fluxo CEP foi possível ampliar o poder de expressão da ferramenta de forma a incluir a possibilidade de lidar com esses tipos de operadores.

Também foram desenvolvidos uma engine de testes e testes relativos aos operadores de fluxo, auxiliando na validação destes. A engine de testes foi criada como um componente a parte, que permite seu uso na simulação de *Streams* de eventos, facilitando a criação de testes, trazendo simplicidade aos mesmos e simulando o aspecto temporal das operações, deixando a cargo do testador apenas os *setups* e asserções necessários nos testes.

Pode ser verificado ao fim desse trabalho que o que foi implementado condiz com o que foi proposto, de forma a possibilitar CEPlin a lidar com operações que antes não lidava, tendo assim seu poder e expressividade ampliados, permitindo uma maior abrangência em tipos de aplicações que podem fazer uso da mesma em seu desenvolvimento.

5.2 Trabalhos futuros

Apesar de ter desenvolvido novas funcionalidades para a biblioteca, alguns pontos ainda podem ser melhorados e outras coisas podem ser acrescidas a partir do ponto em que este trabalho termina. Algumas delas são:

1. Desenvolvimento de mais operadores, a fim de completar toda a gama existente na literatura dos operadores CEP.
2. Ampliação dos testes, validando todos os operadores existentes, dado que este trabalho validou apenas os operadores implementados no mesmo.
3. Avaliação de performance, visando comparar o uso da biblioteca com o de outras e descobrir se a performance obtida pelo código e o consumo de recursos é melhor ou pior que as bibliotecas existentes.
4. Desenvolvimento de aplicações do mundo real, visando utilizar CEPlin em aplicações que possam ser usadas pelo público em geral e verificar sua adoção por desenvolvedores.

Apêndice A

Implementação dos operadores

A.1 Operador *not(stream:)*

```
fun not(stream: EventStream<T>): EventStream<T> {
    val streamAccumulated =
EventStream<MutableList<T>>(stream.accumulator().observable.startWith(ArrayList
<T>()))
    val filtered = this.observable.withLatestFrom(streamAccumulated.observable).filter
{ (event, accumulated) ->
        !accumulated.contains(event)
    }.map {
        it.first
    }
    return EventStream<T>(filtered)
}
```

A.2 Operador *intersect(stream:)*

```
fun intersect(stream: EventStream<T>): EventStream<T> {
    val streamAccumulated =
EventStream<MutableList<T>>(stream.accumulator().observable.startWith(ArrayList
<T>()))
    val filtered = this.observable.withLatestFrom(streamAccumulated.observable).filter
{ (event, accumulated) ->
        accumulated.contains(event)
    }.map {
        it.first
    }.distinct()
```

```
    return EventStream<T>(filtered)
}
```

A.3 Operador *distinct()*

```
fun distinct(): EventStream<T> {
    return EventStream<T>(this.observable.distinct())
}
```

A.4 Operador *orderBy(comparison:)*

```
fun <R : Comparable<R>> orderBy(comparison: ((T) -> R)): EventStream<List<T>> {
    val ordered = this.accumulator().map {
        it.sortedBy(comparison)
    }.observable
    return EventStream(ordered)
}
```

A.5 Operador *groupBy(comparison:)*

```
fun <R> groupBy(comparison: ((T) -> R)): EventStream<Map<R, List<T>>> {
    val grouped = this.accumulator().map {
        it.groupBy(comparison)
    }.observable
    return EventStream(grouped)
}
```

Apêndice B

Engine de testes

B.1 EventStreamSimulator

```
class EventStreamSimulator<T : Event> {
  private var primaryEventManager = EventManager<T>()
  private var secondaryEventManager = EventManager<T>()

  fun simulate(events: List<T>, function: ((stream: EventStream<T>) ->
EventStream<T>)): List<T> {
    val result = ArrayList<T>()

    function(primaryEventManager.asStream()).subscribe {
      result.add(it)
    }

    eventsFromEntries(events, primaryEventManager)
    return result
  }

  fun simulate(events1: List<T>, events2: List<T>, function: ((stream1:
EventStream<T>, stream2: EventStream<T>) -> EventStream<T>)): List<T> {
    val result = ArrayList<T>()

    function(primaryEventManager.asStream(),
secondaryEventManager.asStream()).subscribe {
      result.add(it)
    }
```

```
        eventsFromEntries(events2, secondaryEventManager)
        eventsFromEntries(events1, primaryEventManager)

        return result
    }

    fun simulateCompare(events: List<T>, function: ((stream: EventStream<T>) ->
EventStream<List<T>>)): List<T> {
        var result = listOf<T>()

        function(primaryEventManager.asStream()).subscribe {
            result = it
        }

        eventsFromEntries(events, primaryEventManager)
        return result
    }

    fun <R> simulate(events: List<T>, function: ((stream: EventStream<T>) ->
EventStream<Map<R, List<T>>>)): Map<R, List<T>> {
        var result = mapOf<R, List<T>>()

        function(primaryEventManager.asStream()).subscribe {
            result = it
        }

        eventsFromEntries(events, primaryEventManager)
        return result
    }

    private fun eventsFromEntries(entries: List<T>, manager: EventManager<T>) {
        entries.forEach {
            manager.addEvent(it)
```

```
        }
      }
    }
```

Referências Bibliográficas

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

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

[3] The Reactive Manifesto. https://reactivemanifesto.org

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

[5] LINS, Jonas. CEPlin: A Complex Event Processing Framework for Kotlin. (TG - CIn-UFPE). 2018. Projeto disponível em <https://github.com/RxCEP/CEPlin>.

[6] GUEDES, George. CEPSwift: Complex Event Processing Framework for Swift (TG - CIn-UFPE). 2017. Projeto disponível em <https://github.com/RxCEP/CEPSwift>.

[7] Kotlinlang.org, Kotlin Refference. Disponível em <https://kotlinlang.org/docs/reference/>. Acessado em 24 de agosto de 2018.

[8] ReactiveX - An APIfor asynchronous programing with observable streams. Disponível em <http://reactivex.io>. Acessado em 24 de agosto de 2018.

[9] ReactiveX, RxJava Project, Reactive Extensions for the JVM. Disponível em <https://github.com/ReactiveX/RxJava>. Acessado em 24 de agosto de 2018.

[10] ReactiveX, RxKotlin Project, Kotlinn Extensions for RxJava. Disponível em <https://github.com/ReactiveX/RxKotlin>. Acessado em 24 de agosto de 2018.

[11] JUnit 4 - A simple Framework to write repeatable tests. Disponível em <https://junit.org/junit4/>. Acessado em 11 de outubro de 2018.

[12] Github - Projects | The state of the Octoverse (2018). Disponível em <https://octoverse.github.com/projects.html>. Acessado em 12 de dezembro de 2018.

[13] FlinkCEP - Complex event processing for Flink. Disponível em <https://ci.apache.org/projects/flink/flink-docs-stable/dev/libs/cep.html>. Acessado em 12 de dezembro de 2018.

[14] Asper - Esper for Android. Disponível em <https://github.com/mobile-event-processing/Asper>. Acessado em 12 de dezembro de 2018.

[15] Pratt, Terrence, W. and Marvin V. Zelkowitz. *Programming Languages: Design and Implementation*. 3rd ed. Englewood Cliffs, New Jersey: Prentice Hall, 1996.