Universidade Federal de Pernambuco
Centro de Informática

Graduação em Ciência da Computação

Extensão e avaliação funcional da biblioteca CEPSwift

Filipe Nogueira Jordão

Trabalho de Graduação

Recife
18 de dezembro de 2018

Universidade Federal de Pernambuco
Centro de Informática

Filipe Nogueira Jordão

Extensão e avaliação funcional da biblioteca CEPSwift

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

Orientador: *Prof. Kiev Santos da Gama*

Recife
18 de dezembro de 2018

Resumo

Complex Event Processing ou CEP, é uma área da computação que estuda a detecção de situações complexas a partir da detecção e composição de eventos mais simples. Dado o grande crescimento da área nos últimos anos surgiram diversas bibliotecas que oferecem suporte para a implementação de tais sistemas. Similarmente, tivemos o surgimento do conceito de programação reativa, onde a partir de fluxos de mudança de valores conseguimos mapear, filtrar e combina-los em outros. Devido as similaridades presentes em ambas as áreas, foi desenvolvido uma nova biblioteca CEP baseada nos conceitos de programação reativa, voltada para a linguagem *Swift* derivando seu nome, *CEPSwift*, da mesma. Esse trabalho se propõem a expandir as capacidades funcionais da mesma e realizar o desenvolvimento do ferramental e testes necessários para a sua validação funcional.

Palavras-chave: CEP, Reactive, RX, ReactiveX, Swift, iOS, Testes

Abstract

Complex Event Processing or CEP is an area of computation that studies the detection of complex situations from the detection and composition of simpler events. Given the great increase of the area in the last years several libraries have appeared that support the implementation of such systems. Similarly, we had the emergence of the concept of reactive programming, where from the flows of change of values which can be mapped, filtered and combined in others. Due to the similarities present in both areas, a new CEP library was developed based on the concepts of reactive programming, aimed at Swift and then deriving its name, CEPSwift, from it. This work proposes to expand the functional capabilities of the library and perform the development of the tooling and tests necessary for its functional validation.

Keywords: CEP, Reactive, RX, ReactiveX, Swift, iOS, Tests

Sumário

Lista de Figuras

Lista de Tabelas

CAPÍTULO 1

Introdução

Aplicações dependentes do processamento de fluxos de informação têm se tornado cada vez mais comuns, como sistema financeiros, sistemas meteorológicos. Essa demanda fomentou o surgimento de diversas ferramentas, como *Apache Flink, Esper e Drools Fusion*, cada uma sendo caracterizada por diferenças em suas arquiteturas, modelo de dados e regras, Porém seguindo um modelo muito similar de processamento [4].

Processamento de eventos complexos (Complex Event Processing, CEP), é uma área surgida dessa demanda. Sistemas CEP funcionam através da filtragem e combinação de eventos mais simples, com o intuito de detectar eventos de mais alto nível, notificando as partes interessadas da ocorrência do mesmo. Esses sistemas comumente utilizam uma linguagem de consulta, similar a SQL.

Linguagens Reativas (RLs) por sua vez, tem como principal funcionalidade o tratamento de valores que mudam ao longo do tempo [12], oferecendo um modelo de propagação de mudança similar ao padrão *Observer*[6]. Porém, ao contrário do padrão citado, isso é feito de forma declarativa e o processo de notificação das partes interessadas é feito de maneira implícita. Além disso, o processo de desenvolvimento utilizando RLs é mais simplificado, integrando-se diretamente ao ambiente utilizado como parte da aplicação, e oferece todos os recursos de análise estática do mesmo [8]. Similarmente a CEP, RLs são capazes de receber mudanças de valores de múltiplas fontes, aplicar operações de filtragem e combinação sobre valores e "notificar"mudanças para as partes interessadas, caracterizando um fluxo de processamento muito similar ao visto em sistemas CEP.

1.1 Motivação

Ferramentas de processamento de eventos complexos tem tido uma crescente demanda, entretanto até recentemente não existia nenhuma ferramenta com suporte a aplicações *iOS*. Em 2017, foi desenvolvida a primeira ferramenta para tal plataforma, conhecida como *CEPSwift*[3]. A ferramenta trouxe inovações na área ao oferecer uma abordagem reativa, diferentemente do que é feito por outras ferramentas que utilizam linguagens de consulta para o processamento de eventos. A biblioteca faz uso de tecnologias em ascensão, sendo estas a linguagem *Swift* e a biblioteca *RxSwift*, uma biblioteca da família *ReactiveX*[12].

1.2 Objetivos

Devido à limitação de tempo durante o desenvolvimento da biblioteca *CEPSwift*, seu escopo foi reduzido, o que deixou margem para a sua continuação em novos trabalhos[3]. Este trabalho dá andamento aos trabalhos futuros relacionados a essa biblioteca. Em um primeiro momento foi realizada a implementação de um subconjunto de operadores CEP, com o intuito de expandir o poder de expressividade da biblioteca. Além disso, foi realizado o desenvolvimento de uma ferramenta que permita a realização de testes funcionais, visto que, dado o caráter temporal do processamento de eventos complexos, testes unitários não são suficientes para a implementação dos mesmos. Por fim, utilizaremos a ferramenta desenvolvida para validar funcionalmente a biblioteca através do desenvolvimento de uma *suite* de testes automatizados.

CAPÍTULO 2

Fundamentação

Neste capítulo iremos realizar uma contextualização de temas e áreas que serão de interesse desse projeto. Como se trata da extensão de uma biblioteca já existente, faremos uma breve análise do que a compõem, além de uma revisão de suas motivações, isso incluirá uma breve discussão sobre sistemas de processamento de eventos complexos e programação reativa. Por fim, iremos tratar sobre teste de software, sua motivação e tipos.

2.1 Complex Event Processing

2.1.1 História e Motivação

Complex Event Processing (CEP) é uma área de estudo que tem como objetivo a criação de sistemas capazes de, a partir da combinação eventos mais simples, extrair noções mais complexas. Surgida durante a década de 90, essa área nasceu a partir do estudo da arquitetura orientada a eventos, onde o fluxo de operação de um sistema é definido a partir da ocorrência de eventos [10]. Sendo utilizado em diversas áreas como no mercado financeiro e sistemas previsão meteorológica, a área passou por um grande aumento de interesse que culminou para o surgimento de diversas ferramentas [5].

2.1.2 Funcionamento

Sistemas CEP tem sua origem muito ligada a sistemas *Publisher-Subscriber*, onde consumidores se inscrevem em canais/tópicos de interesse e são notificados pelo sistema caso um evento de interesse ocorra. Sistemas CEP levam esse funcionamento a outro patamar, visto que permite composições de eventos, seja através da filtragem, coincidência, ocorrência de uma dada sequência, ou outras operações, que acabam por criar melhores representações de eventos reais [4]

Como dito anteriormente, sistemas CEP são capazes de detectar eventos abstratos a partir da composição de eventos mais simples. Isso é feito a partir da utilização de um sistema de regras que a partir de regras pré-definidas realiza as composições gerando novos eventos que pode por sua vez ser processados por um outro conjunto de regras. Cada sistema CEP apresenta um sistema de regras próprio, sendo muito comum a utilização de linguagens de consulta similares a SQL como utilizado na linguagem Esper, um exemplo pode ser observado na figura 2.1, nesse exemplo é realizada uma consulta no fluxo de entrada *AutoIdRFIDExample*, de modo que é aplicado uma janela de 60 segundos e cujos eventos devem conter o comando *READ_PALLET_TAGS_ONLY*, por fim os eventos detectados são agrupados pelo seu campo

ID.

```
select ID as sensorId, sum(countTags) as numTagsPerSensor
from AutoIdRFIDExample.win:time(60 seconds)
where Observation[0].Command = 'READ_PALLET_TAGS_ONLY'
group by ID
```

Figura 2.1 Código *Esper*

2.1.2.1 Fluxo de processamento

A existência de diversos modelos de sistema CEP propiciou a construção de diversas arquiteturas e sistemas de regras. Entretanto, um estudo feito por Cugola [4] definiu uma classe de sistemas chamada *Information Flow Processing* que engloba uma série de sistemas similares, incluindo sistemas CEP. O estudo demonstrou que esses sistemas compartilhavam um mesmo conjunto de etapas de processamento, isto é compartilhavam os mesmo conceitos centrais. Essas etapas são:

- *Signaling*, nesta etapa é realizada a detecção dos eventos.
- *Triggering*, etapa responsável por vincular um dado evento com um conjunto de regras de processamento.
- *Evaluation*, realiza a avaliação da "gatilho"atrelado ao evento na etapa anterior.
- *Scheduling*, define a ordem de execução das regras definidas.
- *Execution*, repassa às partes responsáveis pela execução os eventos detectados.

Além das etapas acima, sistemas CEP permitem a reentrada de eventos no fluxo de processamento, isto é, dado que um ou mais eventos sejam detectados e processados por um conjunto de regras gerando um novo evento, esse novo evento pode reentrar no fluxo de processamento gerando novos eventos e assim por diante [4].

2.1.3 Garantias

Dado o caráter crítico de aplicações CEP, tais sistemas devem oferecer um conjunto de garantias funcionais. Nesta seção iremos tratar dos dois principais pontos que serão levados em conta nesse trabalho: Consistência e Tempo de reação [8]. Tais características serão as bases para os testes e avaliação funcional da biblioteca.

- **Consistência**: por se tratar de sistemas de natureza distribuída, aplicações CEP devem se mostrar resistentes à inconsistências causadas por problemas de concorrência, tal característica é primordial nesses sistemas para evitar detecção e consequentemente reação errônea à um evento [8].

- **Tempo de reação**: pela natureza temporal dos sistemas desenvolvidos é primordial que o tempo entre a detecção de um evento de interesse e a notificação das partes interessadas seja baixo, dessa forma precisamos garantir que a emissão e notificação seja realizada logo após a detecção dos mesmos [8].

  [8].

2.2 Programação Reativa

2.2.1 História e Motivação

O mercado cada vez mais exige que aplicações e serviços estejam sempre disponíveis e com baixíssimo tempo de resposta, além disso esses sistemas estão sempre em evolução necessitando que os mesmos sejam flexíveis o suficiente diante de novas funcionalidades e demandas [1]. Esses novos requisitos influenciaram a busca por soluções arquiteturais que dessem suporte ao desenvolvimento dessas novas aplicações.

Dado esse cenário ocorreu a confecção do *Reactive Manifesto* [1]. Um documento que descrevia características que um sistema deveria seguir para se adequar às demandas do mercado, tais características são:

1. *Responsive*, sistemas reativos respondem rapidamente. Sistemas com essas característica se tornam mais fáceis de se trabalhar, erros surgem mais rapidamente e são tratados imediatamente tornando a experiência do usuário final mais fluida e confiável.

2. *Resilient*, tais sistemas devem ser capazes de resistir a falhas de modo que continuem responsivos. Erros são tratados internamente em cada componente isolando falhas a sistemas específicos.

3. *Elastic*, devem ser capazes de se ajustar dinamicamente a mudanças de *workload*, alocando recursos em momentos de maior demanda e desalocando em momentos de baixa demanda.

4. *Message Driven*, a comunicação entre componentes é vital em tais sistemas. A utilização de mensagens para efetuar comunicações reduz o acoplamento entre os componentes. Além disso é possível realizar a comunicação de erros através de canais de mensagens, oferecendo assim uma maneira de delegar o tratamento de erros a outro componente.

Tal documento abriu caminho para a criação de várias ferramentas que implementariam as características descritas. Programação Reativa se mostrou como uma possível solução para esses problemas. Seguindo os conceitos definidos no *Reactive Manifesto*, surgiram diversas ferramentas capazes de desenvolver sistemas que reagem a ações do usuário e eventos exteriores de maneira simplificada, graças a utilização de conceitos presentes em linguagens funcionais [12]. A biblioteca de maior destaque no meio é a *ReactiveX*[12].

2.2.2 ReactiveX

ReactiveX é uma uma biblioteca multi-plataforma que oferece uma API unificada entre as linguagens suportadas, *C, Java, JavaScript, Swift, Scala, Clojure*, facilitando a reutilização de conceitos entre sistemas desenvolvidos em tecnologias diferentes.

A biblioteca se baseia no uso de *Observables*, componentes que representam um fluxo de valores que são emitidos ao longo do tempo. Esses componentes são o resultado da combinação de características presentes nos padrões *Observer*, *Iterator* e Programação Funcional [12].

Observables são agnósticos à estratégia de assincronismo utilizada, de modo que o desenvolvedor não precisa se preocupar com condições de corrida ou outros problemas inerentes a concorrência. Além disso, tais componentes permitem a uma facilidade na hora de compor operações, isso ocorre visto que todas operações aplicadas a um *Observable* gera um novo *Observable* com os resultados do operador sobre o *Observable* original.

2.2.3 Relação com *Complex Event Processing*

Criadas e desenvolvidas por duas comunidades distintas. Linguagens Reativas e Processamento de Eventos Complexos apresentam uma série de similaridades, tanto em relação ao problema que buscam solucionar como características de funcionamento. Nessa sessão realizaremos uma breve discussão dessas similaridades, apontando as principais características que aproximam tais áreas uma da outra [8].

Como visto anteriormente, sistemas CEP operam sobre a ocorrência de eventos. Tais sistemas se baseiam no momento de ocorrência dos eventos processados para executar alguma ação, isto é, dependendo do intervalo de ocorrência entre eventos, ou da sequência de ocorrência dos mesmos, uma dada regra de processamento pode ou não ser executada. Similarmente, Programação Reativa leva em consideração o momento de emissão de um valor para realizar filtragens, composições, mapeamentos entre outras operações [8].

Sistemas CEP tem a capacidade de, ao gerar um novo evento a partir do processamento de eventos de entrada, realizar novos processamentos sobre seus resultados, de modo que possa realizar a composição de seus próprios fluxos de processamento na geração de novos eventos. Programação reativa, ao utilizar um operador sobre um dado fluxo de entrada (*Observable*) gera um novo fluxo. Dessa maneira é possível realizar processamentos sobre resultados de processamentos de maneira similar ao que ocorre em sistemas CEP.

Outro ponto relevante a se destacar é que ambos permitem o processamento de dados vindos de origens distintas e de forma assíncrona. Esse ponto é crucial visto qualquer mapeamento entre as áreas deve levar em consideração essa característica indispensável para ambos.

2.2.4 Sumário da seção

Nesta seção fizemos uma breve discussão em relação a Sistemas Reativos, expondo as principais características e necessidades desses sistemas. Além disso, fizemos uma breve análise sobre uma classe de ferramentas surgida dessas demandas, a Programação Reativa. Por fim, realizamos uma breve comparação com sistemas CEP, apresentando pontos de convergência entre ambas tecnologias.

2.3 Biblioteca *CEPSwift*

O crescimento de interesse na área e a ausência de bibliotecas voltadas para Processamento de Eventos complexos no ecossistema Apple culminaram no criação da biblioteca *CEPSwift*. A biblioteca desenvolvida pelo então estudante de computação George Belo, se baseou nos estudos feitos por Cugola e Margara [4], [8]. A biblioteca foi desenvolvida a partir da biblioteca *RxSwift*, uma biblioteca da família *ReactiveX*, de modo que operadores CEP fossem construídos a partir dos operadores presentes em *ReactiveX*.

2.3.1 Funcionamento

A biblioteca é composta por três principais componentes: *EventManager*, *EventStream*, *ComplexEvent*, sendo cada um desses responsável por um conjunto de funcionalidades bem específicas.

1. *EventManager*, componente responsável por realizar a criação e gerenciamento da emissão de eventos em um dado *EventStream*. Tal componente fornece na sua interface publica uma única função, *addEvent*, que permite a inserção de novos eventos no *EventStream* gerenciado pelo mesmo.

2. *EventStream*, principal componente presente na biblioteca. Tal componente é uma representação de um dado fluxo de eventos, fornecendo a capacidade de gerar novos fluxos (*EventStreams*) a partir da aplicação de operadores. Tal componente oferece na sua interface pública uma série de operadores que, como dito anteriormente, permitem a criação de novos *Streams* a partir dele.

3. *ComplexEvent*, componente responsável por representar a composição de eventos de natureza distinta, isto é, eventos de tipos diferentes. Sua instanciação é feita através da aplicação do operador *merge* presente no componente *EventStream*.

Tanto um *EventStream* como um *ComplexEvent* permitem que outros componentes da aplicação se inscrevam na ocorrência de eventos definidos nos mesmos.

2.3.2 Limitações

A biblioteca foi desenvolvida como um Trabalho de Conclusão de Curso e devido a isso teve seu escopo de desenvolvimento limitado [3]. Dessa forma, foram escolhidos um subconjunto dos operadores definidos por Cugola e Margara [4]. Além disso, a biblioteca só permite a criação de eventos complexos a partir de dois *EventStreams* distintos, de modo que o poder de expressividade dos mesmos se torna limitado. Por fim, devido ao caráter temporal e assíncrono da biblioteca, tornou-se difícil a realização de testes realmente eficazes para constatar o funcionamento de seus operadores [3].

2.3.3 Sumário da seção

Neste seção fizemos uma breve discussão sobre a biblioteca CEPSwift, apontando as razões que motivaram a sua criação e seu funcionamento. Por fim, foram expostas as limitações da mesma, tais limitações serão abordadas nesse trabalho.

2.4 Testes

2.4.1 Testes Unitários

Testes unitários são uma técnica utilizada para realizar validação funcional de código. De maneira geral, um teste pode ser definido como uma função capaz de executar uma outra dado uma entrada e verificar se os resultados obtidos durante a execução equivalem aos resultados esperados[9]. Entretanto, para um teste ser considerado um teste unitário ele deve ser composto por um série de características, sendo a principal, o fato do teste estar avaliando uma unidade independente[9]. Ao contrário do que se pensa, testes unitários não só são aplicados sobre funções e métodos, podendo ser aplicados sobre classes, e até módulos inteiros desde que o mesmo seja independente de outros componentes externos e o resultado do teste possa ser avaliado sem a necessidade de acessar artefatos não públicos, diferenciando-se assim de testes de integração, onde os testes são aplicados sobre componentes que apresentam dependências externas como bancos de dados e relógio da máquina[9].

Em linhas gerais um conjunto de testes pode ser considerado testes unitários se seguirem as seguintes *guidelines*:

1. Automatizado e repetível.
2. Fácil de se implementar.
3. Relevante no futuro.
4. Execução rápida.
5. Independência de outros testes.
6. Fácil diagnostico de falhas.
7. Executável por qualquer pessoa.

2.4.2 Testes de integração

Diferentemente dos testes unitários, testes de integração buscam avaliar o funcionamento de componentes integrados, isto é, utilizando dependências externas reais[9]. Dessa forma, tais testes apresentam resultados menos claros que testes unitários, uma vez que teremos diversos pontos de falha que fogem do controle do componente testado não deixando claro o que de fato ocorreu de errado. Além disso, testes de integração se tornam mais difíceis de se desenvolver

visto que todo *setup* depende da inicialização de outros componentes reais[9] o que pode resultar em maiores tempos de execução e consequentemente o tempo necessário para detecção da falha.

No desenvolvimento de aplicações comumente são desenvolvidas as chamadas piramides de testes. Tais estruturas são compostas por camadas de testes de diferentes tipos, unitários, integração, interface de modo que na base da piramide, consequentemente em maior quantidade, temos testes unitários, seguidos por testes de integração e de interface. Tal estrutura é feita de modo que tenhamos uma maior quantidade de testes que sejam rápidos, mais simples e que sejam mais precisos em relação as falhas em relação a testes mais complexos, lentos e menos claros.

2.4.3 Testes em *Streams*

A realização de testes em sistemas de processamento de *streams* sofre pela dificuldade de alimentar tais sistemas com os dados necessários para a realização dos testes[7]. Isso ocorre uma vez que esses sistemas realizam seu processamento através da recepção de dados em tempo real, ou seja, sobre um espaço de tempo variado[7]. Dessa forma para realização de tais testes é necessária a utilização de algum ferramental externo que seja capaz de alimentar tais sistemas com esses dados de forma repetível e fiel aos dados históricos armazenados[7]. A dependência de tal integração com componentes externos para a realização dos testes faz com que os mesmos sejam caracterizados como testes funcionais e não unitários[9], tornando assim a avaliação e execução dos mesmos mais lenta e mais suscetível a falhas externas.

2.5 Proposta

Este trabalho se propõem a expandir as capacidades da biblioteca CEPSwift e validar a mesma através do desenvolvimento de testes unitários. Isso se dará a partir da implementação de melhorias em componentes já existentes, como *EventStream* e *ComplexEvent*, de modo a garantir a qualidade das funcionalidades oferecidas pelos mesmos. Além disso, será realizado a expansão de suas capacidades através do desenvolvimento de um subconjunto de operadores que ainda não estão presentes na biblioteca, *Flow Management Operators*. Tais operadores serão desenvolvidos seguindo o modelo arquitetural proposto originalmente[3], mantendo assim a compatibilidade com outros operadores e facilitando o aprendizado de novos componentes por usuários antigos. Para a realização de testes, será desenvolvido o um conjunto de simuladores que permitam a simulação da emissão de eventos em momentos específicos, permitindo assim simular o aspecto temporal dos mesmos. Por fim utilizaremos os simuladores desenvolvidos para a criação de uma *suite* de testes capaz de avaliar o funcionamento dos operadores presentes na mesma.

CAPÍTULO 3

Implementação

3.1 Melhorias na biblioteca CEPSwift

Durante a análise da biblioteca foi detectada uma limitação em relação a criação de um *ComplexEvent*. *ComplexEvent* é o componente responsável por realizar a composição de *EventStreams* de tipos distintos, de modo a permitir a detecção da ocorrência de eventos em um conjunto de *streams* dada uma janela temporal. A criação de tal componente é realizada através da operação *merge(with:)*, tal operador recebe dois *EventStreams* e realiza a composição dos dois em um *ComplexEvent*. Porém, uma vez criado não é possível compor esse *ComplexEvent* com mais nenhum *stream*. Além disso, a operação *merge(with:* só permite a composição de dois *streams* de modo que finda por limitar a expressividade da biblioteca.

```
public func merge<R>(with stream: EventStream<R>) -> ComplexEvent {
    let newNumberOfEvents = self.numberOfEvents + 1
    let streamObservable = stream.observable.map { elem in
       return (elem as Any, newNumberOfEvents)
    }

    let newObservable = Observable.merge([self.observable,
                                          streamObservable])
    self.numberOfEvents = newNumberOfEvents

    return ComplexEvent(source: newObservable, count: newNumberOfEvents)
}
```

Figura 3.1 Operador *merge(with:)* original

3.1.1 Solução

Para solucionar a limitação detectada, foram desenvolvidos o operador *asComplexEvent()*, presente no componente *EventStream*, e o operador *merge(with:)*, presente no componente *ComplexEvent*.

O operador *merge(with:* presente no *ComplexEvent* permite a criação de um novo *ComplexEvent* a partir da composição de um *ComplexEvent* com um *EventStream*. Dessa forma, permitimos a composição de mais *EventStreams* em um *ComplexEvent*. Além disso por manter a mesma assinatura do operador de mesmo nome presente em um *EventStream* se torna mais

intuitivo o encadeamento de operações de *merge*.

```
public func merge<R>(with stream: EventStream<R>) ->
  ↪ ComplexEvent {
    let newNumberOfEvents = self.numberOfEvents + 1
    let streamObservable = stream.observable.map { elem in
       return (elem as Any, newNumberOfEvents)
    }

    let newObservable = Observable.merge([self.observable,
                                          streamObservable])
    self.numberOfEvents = newNumberOfEvents

    return ComplexEvent(source: newObservable, count:
      ↪ newNumberOfEvents)
}
```

Figura 3.2 Operador *merge(with:)* melhorado

Também foram detectados problemas nos operadores *Max* e *Min*. Ambos operadores não seguiam o modelo de computação presente nos demais operadores, em vez de gerar um novo *EventStream* contendo os eventos filtrados era realizado um *subscribe* nesses eventos, impedindo que mais operadores fossem encadeados. Com o intuito de resolver essa falha foram feitas algumas alterações na implementação a fim de manter o mesmo padrão dos demais operadores.

```
public func max() -> EventStream<T> {
    return EventStream<T>(withObservable:
        self.observable
            .scan([]) { lastSlice, newValue in
                return Array(lastSlice + [newValue])
            }
            .map { $0.max() }
            .filter { $0 != nil}
            .map { $0! })
        .dropDuplicates()
}
```

Figura 3.3 Operador *max* melhorado

```
public func min() -> EventStream<T> {
    return EventStream<T>(withObservable:
        self.observable
            .scan([]) { lastSlice, newValue in
                return Array(lastSlice + [newValue])
            }
            .map { $0.min() }
            .filter { $0 != nil}
            .map { $0! })
        .dropDuplicates()
}
```

Figura 3.4 Operador *min* melhorado

3.2 Operadores

Nesta seção será discutido o subconjunto de operadores que foram adicionados à biblioteca. Faremos uma breve descrição do funcionamento de cada operador e trataremos de detalhes de implementação.

Os operadores desenvolvidos nesse trabalho são os operadores classificados por Cugola [4] como *Flow Management Operators*. Tais operadores são responsáveis por realizar operações relacionadas ao fluxo de eventos em si, isto é, eles não realizam mapeamentos, alterações ou emissão de novos eventos, sendo responsáveis apenas por gerenciar os eventos que fazem parte de um dado fluxo.

Devido às peculiaridades e padrões inerentes à linguagem, alguns operadores sofreram mudanças nos seus nomes. Isso foi realizado para tornar a biblioteca mais intuitiva a programadores experientes na plataforma. O mapeamento pode ser encontrado na tabela a seguir:

Tabela 3.1 Mapeamento de operadores

Tipo	Operador CEP	Operador CEPSwift
FlowManagement	Except	not(in:)
	Union	union(with:)
	Intersect	intersect(with:)
	Remove-Duplicate	dropDuplicates()
	Duplicate	————————————
	Group-By	grouped(by:)
	Order-By	ordered(by:)

Pode se perceber que não foi desenvolvido um operador para a operação *Duplicate*. Essa escolha se deu visto que qualquer *EventStream* pode ser armazenado em uma variável. Isso permite que o conjunto de operações aplicadas sobre um dado *EventStream* possa ser reutilizado para geração de novos *EventStreams* que compartilham essa mesma origem. Dessa forma, o

operador *Duplicate* se torna obsoleto, uma vez que sua função é permitir a reutilização de valores geradores pela *Engine* na geração de outros fluxos de eventos.

3.2.1 Operador not(:in)

Equivalente ao operador *Except* definido por Cugola [4]. Assim como outros operadores categorizados como *Flow Management Operators*, o operador *not(in:) (Except)*, se comporta de maneira similar a operações sobre conjuntos. O *not(in:* recebe opera sobre dois *EventStreams*, de forma que ambos *EventStreams* são tratados como conjuntos de eventos e as operações aplicadas são responsáveis por gerar um novo conjunto de eventos. O *not(in:)* realiza uma operação de diferença entre dois *EventStreams* de modo que é gerado um novo *EventStream* contendo apenas os eventos presentes no primeiro e que não estão presentes no segundo.

Assim como outros operadores presentes na biblioteca, foi seguido o padrão de retorno das funções onde sempre é retornado um novo *EventStream*. Tal estratégia permite o encadeamento de operações sem a preocupação com a ocorrência de efeitos colaterais, uma vez que sempre geramos um novo *EventStream*, sem nenhuma relação com o original, ao invés de modificar o *EventStream* cujo operador foi aplicado. Segue implementação do operador:

```
public func not(in stream: EventStream<T>) -> EventStream<T> {
    let obsAcc = stream.accumulated()
        .observable
        .startWith([])

    let newObservable = self.observable
        .withLatestFrom(obsAcc) { (event, acc) -> (T, [T]) in
            return (event,acc)
        }.filter { (event, acc) -> Bool in
            return acc.filter { $0 == event }.count == 0
        }
        .map { $0.0 }

    return EventStream<T>(withObservable: newObservable)
}
```

Figura 3.5 Operador *not(in:)*

O operador foi desenvolvido através da combinação dos dois *EventStreams* de modo que sempre que um evento é emitido pelo primeiro, ele é combinado com o conjunto de eventos emitidos até então pelo segundo através do operador withLatestFrom presente na biblioteca RxSwift. Tal combinação permite que no momento de emissão de um evento pelo primeiro *EventStream* seja feita a verificação da sua presença no segundo, dessa maneira pode se remover aqueles que não condizem com a operação, isto é, estão presentes no segundo. Para realização de tal combinação foi criado um operador auxiliar chamado *accumulated()* que para todo evento

emitido por um *stream* é emitido um *array* com todos os eventos até então.

3.2.2 Operador union(with:)

Operador equivalente ao operador *Union* definido por Cugola [4]. Permite a combinação de dois *EventStreams* de mesmo tipo de modo que seja realizada a união dos mesmos, isto é, serão emitidos apenas eventos únicos. Dessa maneira se realizamos a união entre dois *EventStreams* teremos a emissão de todos os eventos que forem emitidos por um dos dois desde que ele não tenha sido emitido ainda, removendo assim eventos duplicados.

A implementação de tal operação foi feita de maneira muito simples. É realizada uma operação de *merge* utilizando a biblioteca *RxSwift* que gera um *Observable* contendo os eventos presentes nos dois *streams*, após isso aplicamos a operação *dropDuplicates()*, também desenvolvida nesse trabalho, que realiza a remoção de eventos repetidos. Dessa maneira geramos um *EventStream* que emite apenas elementos únicos de ambos *streams* pais.

```
public func union(with stream: EventStream<T>) ->
  ↪ EventStream<T> {
    let newObservable = Observable.merge([self.observable,
                                          stream.observable])

    return EventStream<T>(withObservable:
      ↪ newObservable).dropDuplicates()
}
```

Figura 3.6 Operador *union(with:)*

3.2.3 Operador *dropDuplicates()*

Operação responsável por remover eventos repetidos de um *stream*. De acordo com a especificação *ReactiveX* existe uma operação já definida(*Distinct*) que realizaria essa mesma funcionalidade [12], ou seja, não haveria a necessidade de realizar a implementação da mesma neste trabalho, apenas realizaríamos um mapeamento. Entretanto, a implementação para *Swift* (*RxSwift*) não oferece esta operação até o momento, de modo que foi necessário realizar a implementação da mesma.

A implementação desse operador se baseia na verificação evento a evento se o mesmo faz parte do conjunto de eventos emitidos até então pelo *stream*. Isso é feito utilizando o operador *accumulated()*, para gerar um *Observable* que emite todos os eventos até então emitidos pelo *stream*, e combinando seu resultado com o *EventStream* original utilizando a operação *withLatestFrom*. Dessa maneira teremos um *Observable* que emite a tupla (Evento, TodosEventos), em seguida, removemos as tuplas cujo membro Evento aparece mais de um vez no conjunto TodosEventos. Por fim, realizamos um mapeamento de tal *Observable* transformando as tuplas (Evento, TodosEventos) em apenas Evento.

```
public func dropDuplicates() -> EventStream<T> {
    let accObservable = self.accumulated().observable
    let newObservable = self.observable
        .withLatestFrom(accObservable) { (event, acc) ->
         ↪ (T,[T]) in
            return (event, acc)
        }
        .filter { (event, acc) -> Bool in
            return acc.filter { $0 == event }.count == 1
        }
        .map { $0.0 }

    return EventStream<T>(withObservable: newObservable)
}
```

Figura 3.7 Operador *dropDuplicates*

3.2.4 Operador *intersect(with:)*

Operador responsável por realizar a combinação entre dois *EventStreams* de modo que o *EventStream* resultante emita apenas eventos presentes na intersecção das emissões de ambos *streams* de entrada, isto é, eventos que estejam presentes em ambos os *streams*.

Similarmente ao operador *not(in:)*, foi necessária realizar a verificação da existência de um dado evento emitido por um *stream* no outro. Porém, ao contrario do *not(in:)*, que tem interesse em capturar apenas os eventos emitidos pelo primeiro *stream*,

```
public func intersect(with stream: EventStream<T>) ->
  ↪ EventStream<T> {
    let selfAcc = self.accumulated()
    let streamAcc = stream.accumulated()

    let selfInStream = self.observable
        .withLatestFrom(streamAcc.observable,
                        resultSelector: self.isElem)
        .filter { $0 != nil }

    let streamInSelf = stream.observable
        .withLatestFrom(selfAcc.observable,
                        resultSelector: self.isElem)
        .filter { $0 != nil }

    let newObservable = Observable.merge([selfInStream,
                                          streamInSelf])
        .map { $0! }

    return EventStream<T>(withObservable: newObservable)
        .dropDuplicates()
}
```

Figura 3.8 Operador *intersect(with:)*

3.2.5 Operador ordered(by:)

Esse operador busca emitir os eventos gerados por um *EventStream* de forma ordenada, seguindo uma dada regra de ordenamento.

A sua implementação faz uso do operador *accumulated()* e realiza um mapeamento sobre os valores de modo a ordena-los utilizando a função de ordenamento recebida.

```
public func ordered(by comparison: @escaping (T,T) -> Bool) ->
  ↪ EventStream<[T]> {
    return self.accumulated().map { events -> [T] in
        return events.sorted(by: comparison)
    }
}
```

Figura 3.9 Operador *ordered(by:)*

3.2.6 Operador group(by:)

Operador responsável por gerar agrupamentos de eventos processados pelo *EventStream* de entrada de acordo com uma regra especificada pelo usuário do mesmo. Sua implementação faz uso do operador *scan* e realiza o agrupamento sobre um dicionário, adicionando novos eventos no mesmo de acordo com a chave gerada pela função passada na sua chamada.

```
public func group<K>(by keyed: @escaping (T) -> K) ->
  ↪ EventStream<[K: [T]]> {
    let newObservable = self.observable
        .scan([K: [T]]()) { (groups, elem) in
            let key = keyed(elem)
            let newGroup = [key: [elem]]

            return groups.merging(newGroup) { $0 + $1 }
    }

    return EventStream<[K: [T]]>(withObservable:
      ↪ newObservable)
}
```

Figura 3.10 Operador *group(by:)*

3.3 Engine de testes

Nesta seção será discutido o desenvolvimento do ferramental necessário para a realização de simulações e testes dos operadores propostos neste trabalho. Essa necessidade surgiu devido a natureza do problema a ser avaliado, depender da alimentação em tempo real dos eventos de entrada torna o desenvolvimento dos testes mais dificil[7].

Dado os problemas inerentes a tais sistemas e devido à utilização da biblioteca *RxSwift*, foi realizada uma investigação de como são realizados testes para sistemas que à utilizam. O resultado mais interessante foi uma biblioteca auxiliar a *RxSwift* chamada *RxTest*, tal biblioteca oferece um conjunto de extensões que buscam solucionar os problemas inerentes a testar tais sistemas. Para os propósitos desse trabalho iremos nos focar no componente *TestScheduler* uma vez que ele oferece os recursos necessários para o desenvolvimento dos nossos testes[12].

Em *RxSwift* um *Scheduler* é um componente que abstrai como uma operação será executada, de modo que podemos definir a estratégia a ser utilizada, isto é, em qual fila de processamento, de maneira paralela ou sequencial, etc. Além disso, esse componente permite a criação de *Observables* capazes de emitir um conjunto predefinido de eventos em momentos específicos. Por fim, ao utilizar a operação *start* podemos aplicar uma série de operações sobre os *Observables* criados a partir do mesmo, recebendo como resultado o conjunto de eventos resultantes das mesmas de maneira síncrona[12].

Dada ao fato da biblioteca *CEPSwift* ser desenvolvida de forma que isole a utilização de

componentes *RxSwift* do usuário, decidi por desenvolver dois *Wrappers* (*EventStreamSimulator*, *ComplexEventSimulator*) em torno dos componentes presentes na biblioteca *RxTest* de modo a permitir que desenvolvedores criem testes utilizando apenas os conceitos introduzidos pelo uso de *CEPSwift*. Dessa maneira, além de manter as interfaces oferecidas pela biblioteca agnósticas às dependências da mesma[3], facilitamos o desenvolvimento de testes de sistemas que a utilizem.

3.3.1 Simuladores

Os componentes *EventStreamSimulator* e *ComplexEventSimulator* permitem a execução de operações sobre um conjunto de eventos pré-definidos, cada evento possui um *timestamps*, a partir desses dados de entrada utilizamos os *schedulers* virtuais oferecidos pela biblioteca *RxTest* para tornar a execução da simulação síncrona e realizar o agendamento da emissão dos eventos, desse modo se torna possível simular o caráter temporal da emissão e detecção de eventos, característica essencial no desenvolvimento de testes de aplicações dessa natureza. Ambos os simuladores, são implementados de maneira similar e oferecem uma *API* consistente, divergindo apenas nos componentes simulados por cada um: *EventStreams* e *ComplexEvents* respectivamente.Como dito anteriormente, ambos os componentes tem interfaces similares expondo uma função *simulate* que recebe os eventos de entrada e uma *closure* retornando o *EventStream* ou *ComplexEvent* gerado a partir dos fluxos de entrada.

Com o intuito de melhorar a legibilidade foi criado um conjunto de *TypeAliases* 3.11. Tais tipos são utilizados nas assinaturas das funções presentes nos simuladores, a utilização dos mesmos facilita a identificação de tipos de entrada comuns entre funções e reduzindo o tamanho das assinaturas das mesmas. Isso pode ser melhor observado na implementação dos operadores A.

```
public typealias EventEntry<T> = (time: Int, event: T)

public typealias StreamsToComplex<T,K> = (EventStream<T>,
  ↪ EventStream<K>) -> ComplexEvent
public typealias StreamToComplex<T> = (EventStream<T>) ->
  ↪ ComplexEvent
public typealias StreamToStream<T,K> = (EventStream<T>) ->
  ↪ EventStream<K>
public typealias StreamsToStream<T,K> = (EventStream<T>,
  ↪ EventStream<T>) -> EventStream<K>
```

Figura 3.11 *TypeAliases*

3.3.2 Comparação

A utilização dos simuladores permite a criação de testes simples e mais completos, permitindo especificar o momento de emissão de cada evento. Ao contrário do que acontece numa implementação de testes sem o uso dos mesmos como pode ser visto a seguir. Para que o teste do

operador *map* seja realizado é necessária a criação de um *EventManager* e manualmente emitir cada evento. Essa abordagem não permite que controlemos o momento de emissão de cada evento, apenas a sua ordem, para que esse controle seja obtido caberia ao usuário desenvolver alguma forma de agendamento de eventos.

```
let manager = EventManager<IntEvent>()
let originalEvents = [0, 1, 2, 3].map(IntEvent.init)
let expectedEvents = [0, 10, 20, 30].map(IntEvent.init)
var events = [IntEvent]()

manager.stream
    .map(transform: { IntEvent(value: $0.value * 10) })
    .subscribe(onNext: { event in
        events.append(event)
    })

originalEvents.forEach(manager.addEvent)

it("Should output \(events.count) events") {

    ↪ expect(events.count).toEventually(equal(expectedEvents.count))
}

it("Should output the expected events") {
    expect(events).toEventually(equal(expectedEvents))
}
```

Figura 3.12 Teste sem uso dos simuladores

O mesmo não acontece ao utilizar um dos simuladores, o próprio simulador se encarrega de gerar os *EventStreams* necessários para a simulação. Além disso, é possível definir os *timestamps* de ocorrência de cada evento, simulando a ocorrência simultânea e espaçamento temporal de eventos. Por fim, permite que tudo seja executado de maneira síncrona, não necessitando o uso de artifícios como a função *toEventually* presente na biblioteca *Nimble* 4.

```
context("When mapping a EventStream") {
    let input = [
        (time: 1, event: IntEvent(value: 0)),
        (time: 3, event: IntEvent(value: 1)),
        (time: 5, event: IntEvent(value: 2)),
        (time: 5, event: IntEvent(value: 3)),
        ]

    let expectedOutput = [
        (time: 1, event: IntEvent(value: 0)),
        (time: 3, event: IntEvent(value: 10)),
        (time: 5, event: IntEvent(value: 20)),
        (time: 5, event: IntEvent(value: 30)),
        ]

    func mapTimesTen(_ stream: EventStream<IntEvent>) ->
     ↪ EventStream<IntEvent> {
        let resultStream = stream.map { IntEvent(value:
         ↪ $0.value * 10) }

        return resultStream
    }

    let simulator = EventStreamSimulator<IntEvent>()
    let output = simulator.simulate(with: input,
                                    handler: mapTimesTen)

    it("Should output \(expectedOutput.count) events") {
        expect(output.count).to(equal(expectedOutput.count))
    }

    it("Should output the expected events") {
        for (idx, elem) in expectedOutput.enumerated() {
            let outputElem = output[idx]

            expect(outputElem.time).to(equal(elem.time))
            expect(outputElem.event).to(equal(elem.event))
        }
    }
}
```

Figura 3.13 Teste com uso dos simuladores

CAPÍTULO 4

Avaliação Funcional

Com o intuito de validar a atual implementação da Biblioteca, foi elaborado um conjunto de testes, utilizando os simuladores desenvolvidos, que cubram todos os operadores presentes na mesma. Nesse capítulo iremos documentar os mesmos da seguinte maneira, primeiro definiremos qual operador e contexto está sendo avaliado, em seguida teremos uma representação do cenário de teste descrito por meio de um *Marble Diagram*, diagrama comumente utilizado para realizar a descrição de fluxos e operadores de sistemas reativos, por fim analisaremos os resultados obtidos no teste, explorando comportamentos inesperados.

Com o intuito de deixar a *suite* de testes o mais clara possível utilizaremos as Bibliotecas *Quick* e *Nimble*[11] para descrever cada caso testado, bibliotecas de teste baseadas em *Behaviour Driven Development* [11], onde cada caso de teste representa um comportamento esperado da unidade testada descrevendo o mesmo e as características que satisfazem o teste. A escolha das mesmas se deve as ferramentas de contextualização oferecidas, que permitem um entendimento mais rápido dos testes.

Para a realização dos casos de teste foram criados dois tipos básicos de eventos, *IntEvent* 4.1 e *StringEvent* 4.2. Tais eventos, são constituídos por um valor, que varia de acordo com o tipo do mesmo, e implementam os protocolos *Equatable* e *Comparable*. A implementação de tais protocolos é utilizada para facilitar a verificação dos resultados, permitindo uma comparação direta entre dois eventos quaisquer. Também teremos a implementação do protocolo *CustomStringConvertible* que permitirá uma melhor visualização dos eventos resultantes dos testes.

```
class IntEvent: Event, Comparable, Equatable,
 ↪ CustomStringConvertible {
    var timestamp: Date
    var value: Int

    var description: String {
        get {
            return "IntEvent -> \(value)"
        }
    }

    init(value: Int) {
        self.timestamp = Date()
        self.value = value
    }

    static func <(lhs: IntEvent, rhs: IntEvent) -> Bool {
        return lhs.value < rhs.value
    }

    static func ==(lhs: IntEvent, rhs: IntEvent) -> Bool {
        return lhs.value == rhs.value
    }
}
```

Figura 4.1 Implementação *IntEvent*

```
class StringEvent: Event, Equatable {
    var timestamp: Date
    var value: String

    init(value: String) {
        self.timestamp = Date()
        self.value = value
    }

    static func ==(lhs: StringEvent, rhs: StringEvent) -> Bool
      ↪ {
        return lhs.value == rhs.value
    }
}
```

Figura 4.2 Implementação *StringEvent*

Cada caso de teste será composto de três partes, *setup* 4.3, *execution* 4.4, *assertion* 4.5.

A etapa de setup é responsável pela criação de objetos e configuração do cenário. Os casos de testes desenvolvidos nesse trabalho utilizarão essa etapa para a criação dos eventos de entrada e de saída.

```
let input = [
    (time: 1, event: IntEvent(value: 0)),
    (time: 3, event: IntEvent(value: 1)),
    (time: 5, event: IntEvent(value: 2)),
    (time: 5, event: IntEvent(value: 3)),
]

let expectedOutput = [
    (time: 1, event: IntEvent(value: 0)),
    (time: 3, event: IntEvent(value: 10)),
    (time: 5, event: IntEvent(value: 20)),
    (time: 5, event: IntEvent(value: 30)),
]

func mapTimesTen(_ stream: EventStream<IntEvent>) ->
 ↪ EventStream<IntEvent> {
    let resultStream = stream.map { IntEvent(value: $0.value *
     ↪ 10) }

    return resultStream
}
```

Figura 4.3 Etapa de *setup*

A etapa de *execution* é responsável por executar as operações a serem testadas. Utilizaremos essa etapa para executar a simulação dos fluxos, gerando os eventos resultantes da aplicação do operador sob a entrada.

```
let simulator = EventStreamSimulator<IntEvent>()
let output = simulator.simulate(with: input,
                                handler: mapTimesTen)
```

Figura 4.4 Etapa de *execution*

A etapa de assertion é responsável por checar se os resultados obtidos durante a etapa de *execution* estão de acordo com os resultados esperados definidos na etapa de *setup*. Essa etapa será basicamente idêntica para todos os casos de teste elaborados nesse trabalho, uma vez que todos as simulações resultarão em uma lista de eventos. Além disso, estamos avaliando as mesmas características para todos os operadores, isto é, precisão temporal e consistência. Dessa forma, podemos utilizar a mesma estratégia de avaliação para todos os operadores.

```
it("Should output \(expectedOutput.count) events") {
    expect(output.count).to(equal(expectedOutput.count))
}

it("Should output the expected events") {
    for (idx, elem) in expectedOutput.enumerated() {
        let outputElem = output[idx]

        expect(outputElem.time).to(equal(elem.time))
        expect(outputElem.event).to(equal(elem.event))
    }
}
```

Figura 4.5 Etapa de *assertion*

4.1 Casos de teste

4.1.1 Operador *map*

O operador *map* é responsável por realizar transformações sobre eventos emitidos por um *EventStream*, isto é, ele não irá realizar exclusão de elementos mas sim a transformação de um evento em outro. Um exemplo seria a aplicação do operador sobre um fluxo de entrada emissor de *IntEvents* transformando o tais eventos através da multiplicação dos mesmos. O cenário utilizado para realizar a validação do operador é o descrito anteriormente, um fluxo de *IntEvents* de entrada sendo multiplicado por dez. O diagrama4.6 a seguir representa a entrada e saída esperada utilizada neste cenário, para cada evento emitido pelo fluxo de entrada teremos a emissão de um novo evento cujo valor é igual ao original multiplicado por dez. Sua implementação pode ser encontrada no apêndice B.1.

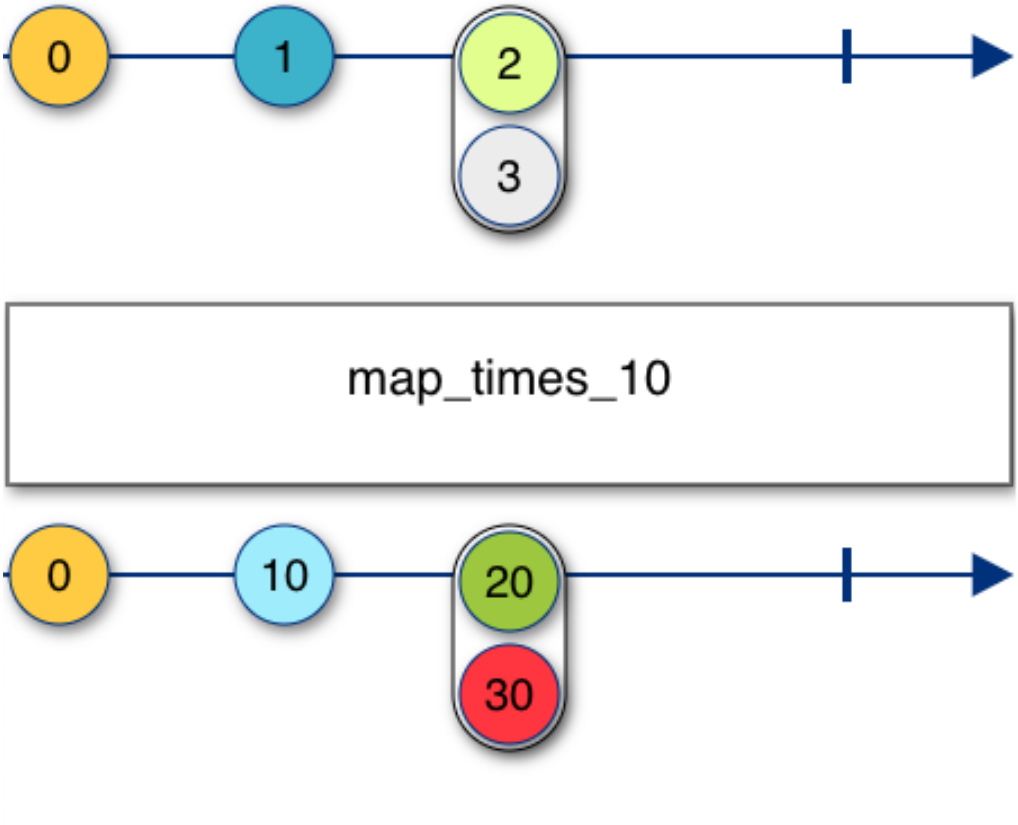

Figura 4.6 Diagrama cenário *mapping*

4.1.2 Operador *filter*

O operador *filter* é responsável por realizar incluir ou excluir elementos de um *EventStream*, para cada evento emitido pelo fluxo de entrada é realizada uma verificação se tal evento se enquadra na regra utilizada, em caso positivo teu evento é emitido pelo fluxo de saída. Para a realizar a validação do operador teremos o seguinte cenário. Como entrada teremos um fluxo contendo diversos eventos numéricos, alguns ocorrendo ao mesmo tempo que outros, e como regra de filtragem verificaremos se um evento é par, o resultado esperado será um novo fluxo contendo apenas os eventos incluídos na regra. O cenário descrito pode ser observado na figura a seguir4.7, eventos ocorridos no mesmo momento são representados por uma pilha vertical. Sua implementação pode ser encontrada no apêndice B.2.

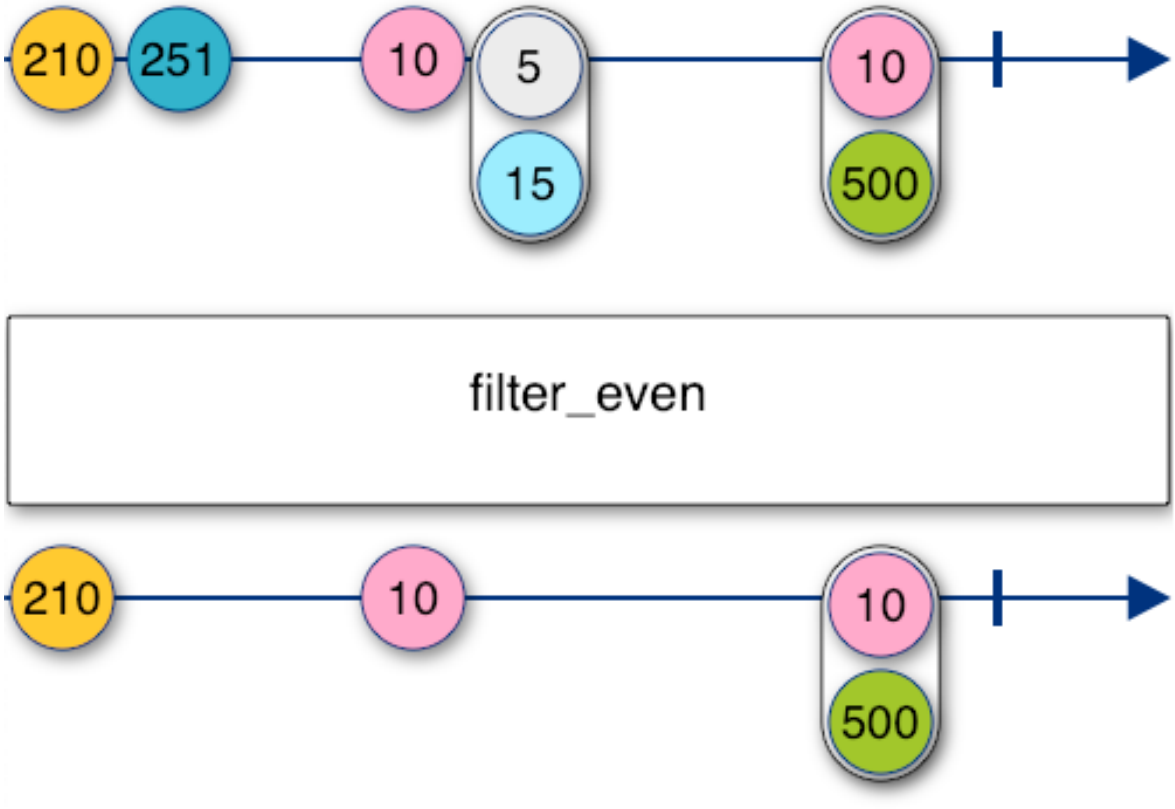

Figura 4.7 Diagrama cenário *filtering*

4.1.3 Operador *max*

O operador *max* é responsável por realizar a filtragem dos máximos globais emitidos por um *EventStream*, isto é ele deve ser capaz de verificar para toda emissão de um evento se este evento é maior do que todos os eventos já emitidos anteriormente. Para realizar a validação de tal operador foi desenvolvido o seguinte cenário de teste, teremos como entrada um conjunto de *IntEvents* de diversos valores, e ao aplicar o operador devemos obter como saída apenas os valores que se caracterizarem como um máximo global. Tal cenário está descrito na figura a seguinte 4.8 onde teremos representados o fluxo de entrada e o fluxo de saída após a aplicação do operador. Sua implementação está disponível no apêndice B.3

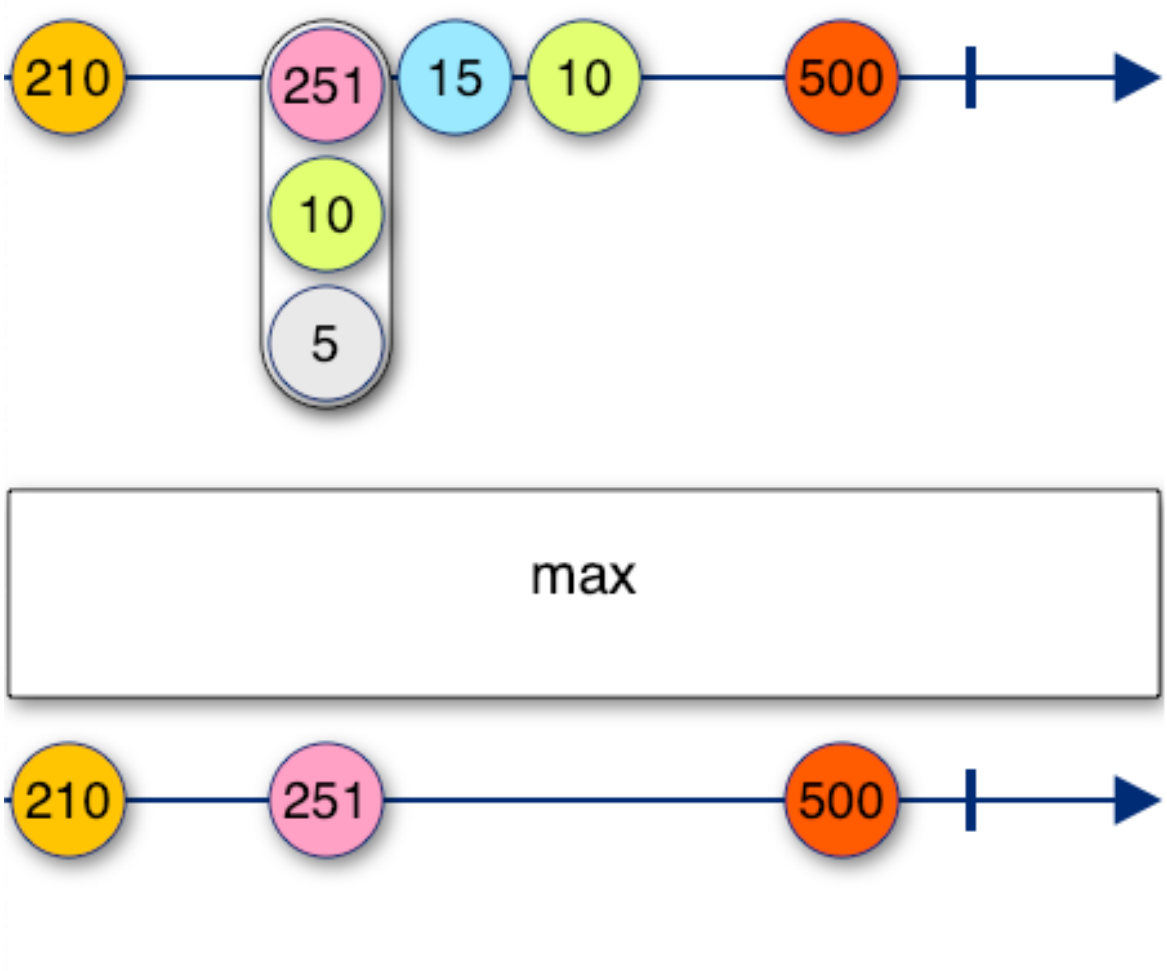

Figura 4.8 Diagrama cenário *max*

4.1.4 Operador *min*

Similarmente ao operador *max* o operador *min* tem como responsabilidade detectar eventos que se caracterizem como mínimos globais, isto é, contém o menor valor detectado até então. Teremos então um cenário de validação muito similar ao utilizado pelo operador *max*, porém dessa vez estamos buscando os valores mínimos emitidos pelo *EventStream*. Tal cenário está descrito na figura a seguir 4.9 e sua implementação está disponivel no apêndice B.4.

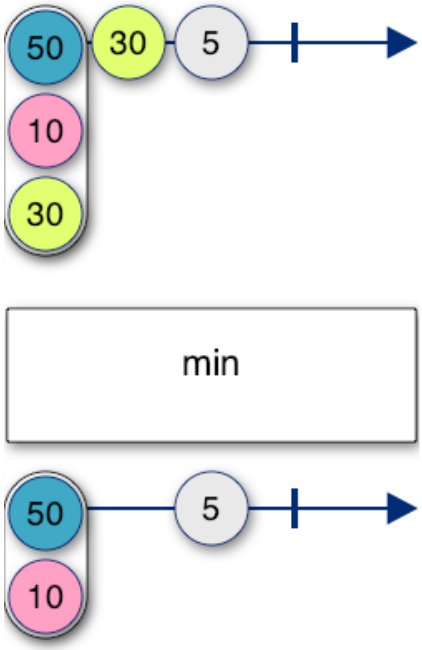

Figura 4.9 Diagrama cenário *min*

4.1.5 Operador *union*

A partir desse momento trataremos de operadores mais complexos e portanto exigindo mais cuidado na sua avaliação. O operador *union* é responsavel por realizar a união de dois *EventStreams* de modo que os eventos emitidos sejam únicos. Tal operação é equivalente a a união de dois conjuntos de eventos, ou seja, teremos um novo conjunto contendo apenas uma instância de cada evento emitido pelos *streams*. O cenário utilizado para o operador foi desenvolvido a partir da simples união de dois *EventStreams*. Entretanto, como tal operador é aplicado sobre mais de um *EventStream* é necessária a verificação de problemas na detecção de eventos ocorrendo simultaneamente em ambos os *EventStreams*, como por exemplo, *glitches* [14]. Para isso, o operador será aplicado sobre dois fluxos que contenham colisões, e teremos como resultado esperado um conjunto de eventos contendo todos os eventos inclusive os relativos as colisões. O cenário pode ser melhor observado no diagrama a seguir 4.10 e sua implementação pode ser encontrada no apêndice B.5.

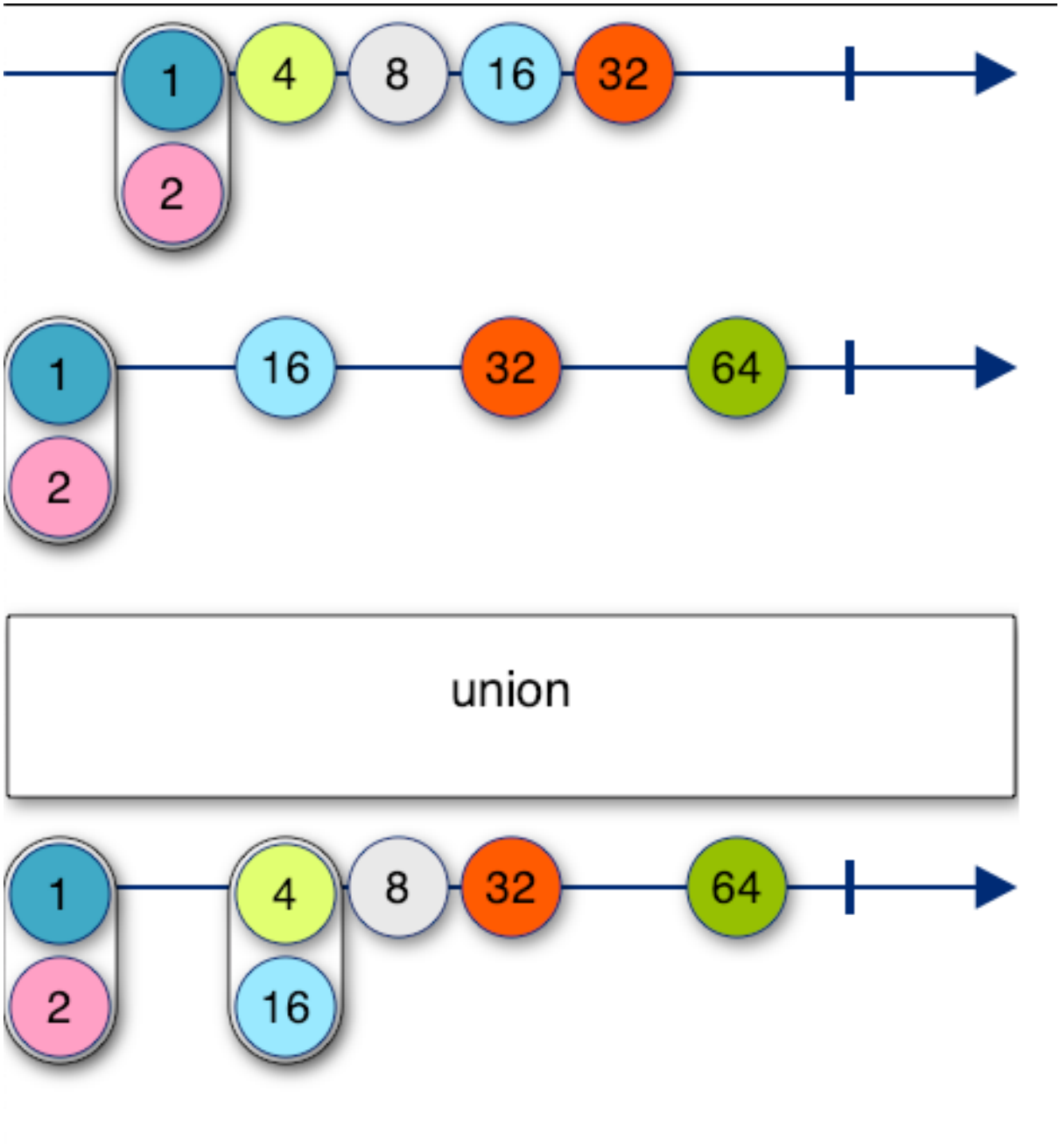

Figura 4.10 Diagrama cenário *union*

4.1.6 Operador *intersect*

O operador *intersect* é responsável por realizar a junção de dois *EventStreams* de modo que os valores emitidos pelo novo fluxo sejam valores comuns aos dois fluxos de entrada, isto é, um evento só pode ser emitido se ambos os *streams* de entrada o emitirem. Assim como o operador *union*, o operador*intersect* tem equivalência com o operador de mesmo nome relativo

a conjuntos.

Dado as características similares entre os operadores *intersect* e *union*, o cenário foi desenvolvido com os mesmos problemas em mente que o cenário do operador *union*, apresentando colisões temporais com o intuito de detectar problemas relacionados com *glitches*. O cenário utiliza dois fluxos distintos de entrada contendo alguns valores em comum, tais valores serão os resultados emitidos pelo fluxo de saída. O mesmo pode ser melhor observado no diagrama a seguir 4.11 e sua implementação pode ser encontrada no apêndice B.6.

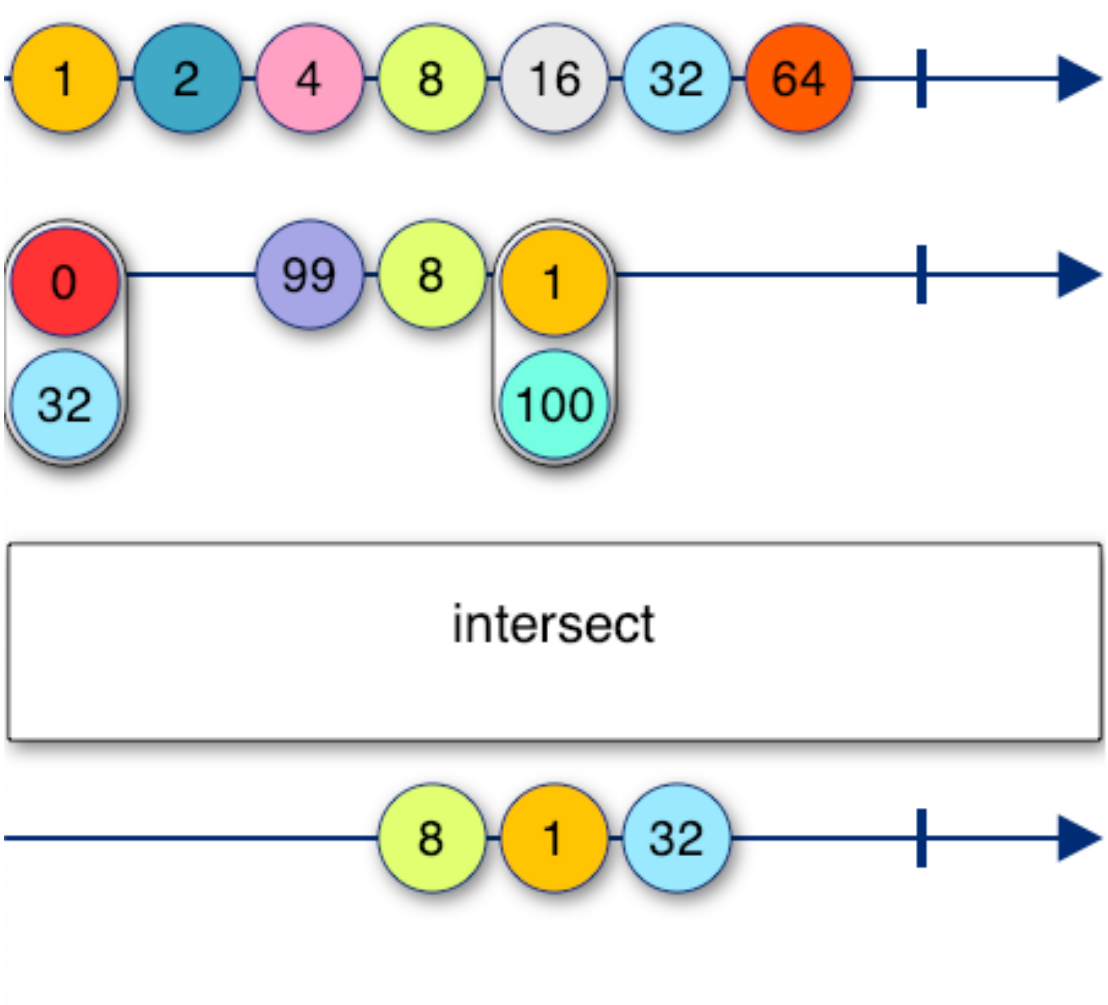

Figura 4.11 Diagrama cenário *intersect*

4.1.7 Operador *not(in:)*

O operador *not(in:)* é responsável por realizar a filtragem de elementos emitidos por um *EventStream* a partir dos elementos emitidos por um outro *EventStream* de modo que cada novo evento emitido pelo primeiro só seja adicionado ao fluxo de saída caso ainda não tenha sido emitido pelo segundo. O caso de teste desenvolvido recebe como entrada dois fluxos

de eventos e deve resultar nos eventos presentes no primeiro *stream* que no momento da sua emissão não foram emitidos pelo segundo *stream*. O cenário pode ser melhor observado no diagrama a seguir 4.12 e sua implementação pode ser encontrada no apêndice B.7.

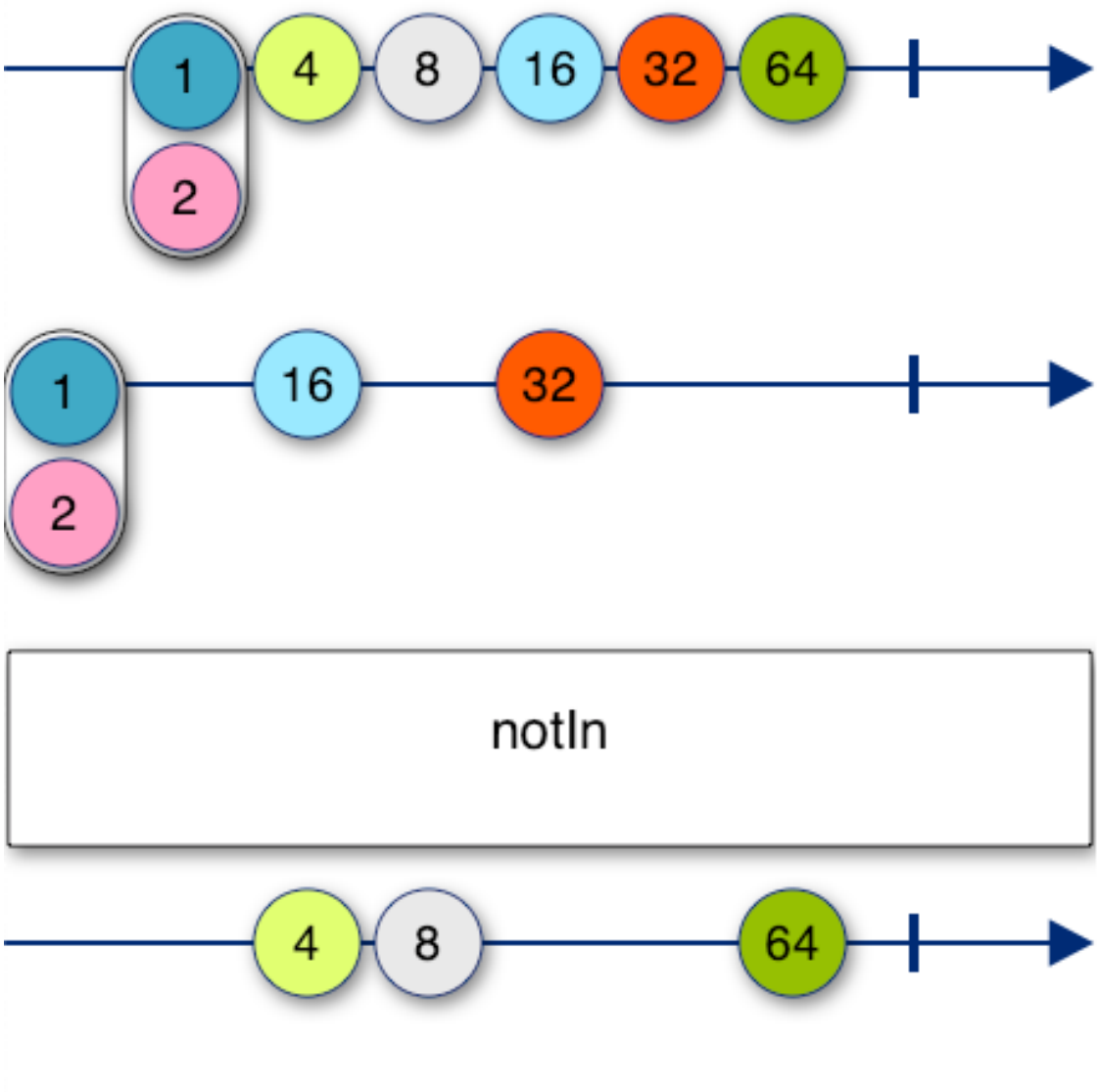

Figura 4.12 Diagrama cenário *not(in:)*

4.1.8 Operador followedBy

O operador followedBy é responsável por realizar uma filtragem de duplas de eventos em sequência a partir de um dada regra. Tal operador pode ser utilizado para detectar relações entre evento subsequentes, como por exemplo o crescimento ou decrescimento dos valores em *IntEvents*. O cenário de teste utilizado para a avaliação do operador consiste justamente na detecção de padrões de crescimento em sequências de *IntEvents*, para isso teremos como entrada

um conjunto de eventos contendo trechos crescentes e trechos decrescentes. O cenário pode ser melhor visualizado no diagrama a seguir 4.13 e sua implementação pode ser encontrada no apêndice B.8.

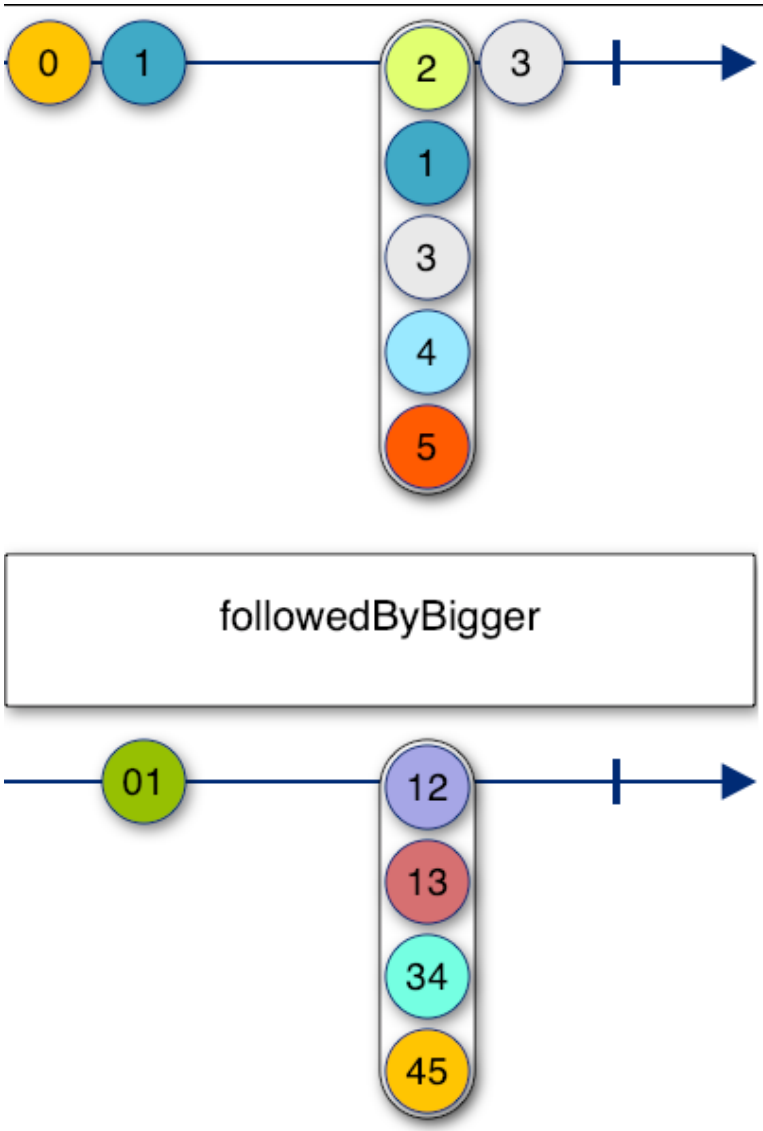

Figura 4.13 Diagrama cenário *followedBy*

4.1.9 Operador *dropDuplicates*

O operador *dropDuplicates* é responsável por realizar a remoção de eventos duplicados em um *EventStream*, ou seja para cada evento emitido pelo fluxo de entrada é feita uma verificação se o mesmo já foi emitido anteriormente, caso positivo ele não será emitido no fluxo resultante.

O cenário utilizado para a avaliação do operador *dropDuplicates* consiste numa sequência de eventos intercalados ou não temporalmente com a ocorrência de duplicatas, ao aplicar o operador o resultado deve conter apenas os eventos únicos removendo assim reincidências de um evento. O cenário pode ser melhor observado no diagrama a seguir 4.14 e sua implementação está disponível no apêndice B.9

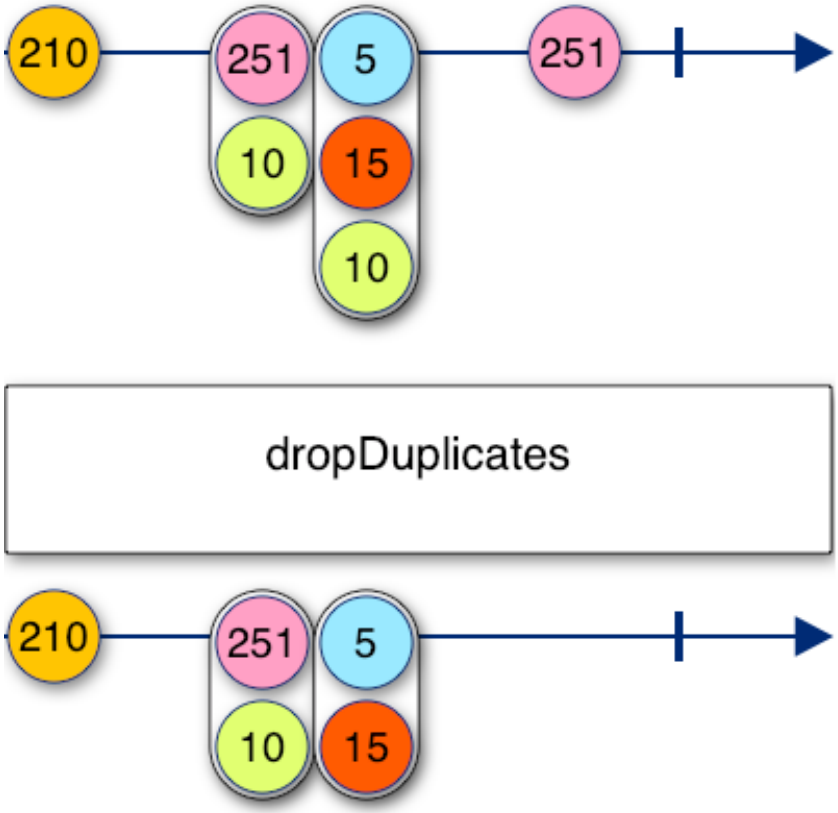

Figura 4.14 Diagrama cenário *dropDuplicates*

4.1.10 Operador *window*

A partir de agora descreveremos cenários de teste que não puderam ser representados através de um *Marble Diagram*, isso se deu devido as limitações de representação da ferramenta (*MarbleDiagram*). O operador *window* é responsável por realizar o agrupamento de eventos em um *EventStream* dado um intervalo de tempo e capacidade máxima de eventos. Por exemplo, poderíamos utilizar tal operador para detectar os cinco primeiros eventos que ocorreram em uma janela de três segundos. O cenário utilizado para validar tal operador busca verificar se o mesmo consegue agrupar janelas de eventos pela duração e capacidade máximas, dado a isso teremos como entrada eventos ocorrendo de maneira espaçada temporalmente e também ocor-

rendo simultaneamente e causando um estouro de capacidade. Sua implementação pode ser encontrada no apêndice B.10.

4.1.11 Operador *group(by:)*

O operador *group(by:)* é responsável por realizar o agrupamento de eventos segundo uma dada regra, tal operador funciona similarmente ao operador *filter*, entretanto a função que dita a filtragem, entretanto ao invés de remover elementos que não se enquadrem na regra utilizada iremos agrupar os eventos de acordo com o retorno da aplicação da regra sobre cada evento, dessa forma o resultado da aplicação deste operador é será um dicionário cujas chaves são os resultados obtidos pela aplicação da regra e os valores de cada chave são os eventos utilizados no processo. O cenário para a avaliação do operador *group(by:)* consiste no agrupamento de eventos pares e impares, os dados de entrada também contém colisões. Sua implementação pode ser encontrada no apêndice B.11.

4.1.12 Operador *ordered(by:)*

O operador *ordered(by:)* é responsável por ordenar todos os eventos emitidos por um *EventStream* seguindo a regra passada como entrada. O resultado obtido da aplicação de tal operador será a emissão do conjunto de todos os eventos ordenados sempre que um novo evento for emitido. O cenário de teste utilizado para o operador *ordered(by:)* consiste em um fluxo de eventos mesclando valores menores e maiores que seus anteriores, de maneira temporalmente espaçada e simultânea. Sua implementação pode ser encontrada no apêndice B.12.

4.1.13 Operador *merge*

O operador merge é o responsável por realizar a união entre dois *EventStreams* de tipos diferentes em um *ComplexEvent*, isso é feito a partir da detecção de eventos em ambos fluxos de entrada formando assim um evento complexo caracterizado pela emissão de ambos os eventos. Esse operador é particularmente importante dado que ele é o principal componente na detecção de tais eventos. O cenário utilizado para o operador *merge* consiste na composição de dois fluxos de eventos de tipos diferentes, como o valor dos eventos emitidos não importa para esse operador, uma vez que todo o processamento, transformação e filtragem é esperado que tenha sido feito anteriormente, utilizaremos fluxos de eventos do tipo *IntEvent* e *StringEvent* com valores arbitrários. O cenário deve ser capaz de detectar a ocorrência de um evento complexo após a emissão de um elemento de cada *EventStream*. Sua implementação pode ser encontrada no apêndice B.13.

4.2 Resultados

Ao executar a *suite* desenvolvida podemos verificar alguns resultados. Em primeiro lugar, podemos dizer que a *suite* de fato realiza uma checagem dos operadores oferecidos pela biblioteca, uma vez que foi obtido uma taxa de *code coverage* bastante elevada, sendo esta ainda maior ao analisarmos as classes *EventStream*, *EventStream+Comparable*, classes nas quais se concentram os operadores, onde obtivemos uma cobertura acima de 90%.

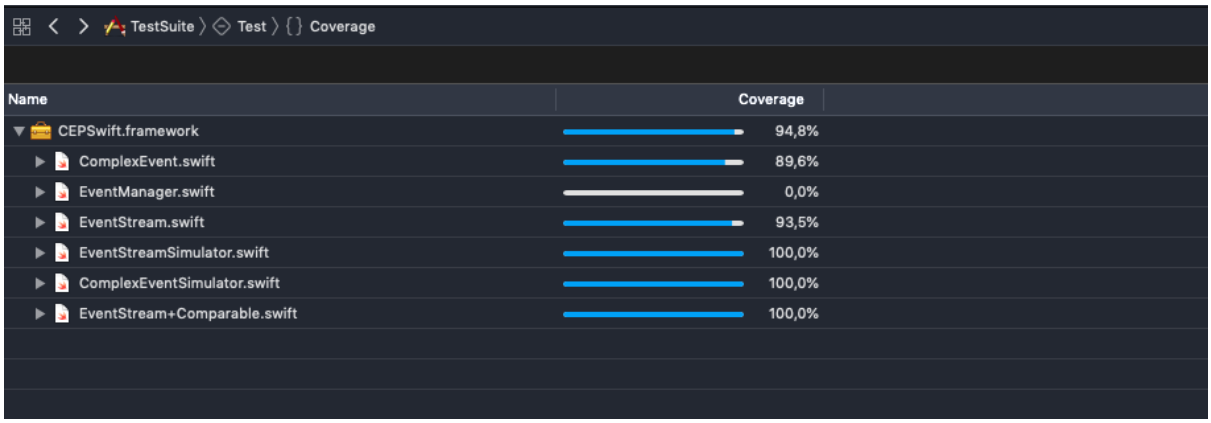

Figura 4.15 Code Coverage

Entretanto, nem todos os resultados foram positivos. Ao executar a *suite* obtivemos resultados de falha em alguns cenários, demonstrando que alguns operadores não estão funcionando de maneira correta sempre ou em alguns casos 4.16.

As falhas apresentadas ocorreram durante a aplicação dos operadores *window* e *merge*, sendo a falha ocorrida no operador *merge* extremamente preocupante. O operador *merge* é o responsável pela criação e composição do componente *ComplexEvent*, se tal operador apresenta inconsistências a utilização da biblioteca é comprometida.

Após uma analise realizada sob a implementação do operador *merge*, foi possível detectar a causa por trás da do problema. O operador *merge* é implementado pela utilização do operador *buffer*, presente na biblioteca *RxSwift*, de modo que os eventos são agrupados de acordo com a quantidade de fluxos que compõem o evento complexo, isto é, se criamos um *ComplexEvent* a partir da união de três fluxos distintos, os eventos desses fluxos serão agrupados em janelas de três elementos com duração máxima de cinco segundos. Um evento emitido é inserido na janela ativa no momento da emissão de modo que o mesmo não pode estar presente em mais de uma janela. Após isso, para cada janela detectada é verificado se a mesma contém um evento de cada fluxo de entrada, isso e o fato que um evento só pode ser inserido em uma janela é a origem da falha no operador, uma vez que dentro de uma janela de cinco segundos pode ocorrer a emissão de dois eventos por um dos fluxos e um por cada um dos três fluxos restantes de modo que caracterizaria um evento complexo, entretanto visto que as janelas tem como capacidade máxima a quantidade de fluxos que compõem o evento complexo podemos formar uma janela da seguinte forma F1,F1,F2 deixando de fora o evento F3 e dessa forma não

detectando o evento complexo ocorrido.

A falha detectada ocorrida no operador *window* se deve ao fato o mesmo na sua implementação definir uma *Scheduler* para a sua execução, dessa forma o operador se força a executar fora do ambiente de teste, utilizando o relógio real do sistema, gerando assim discrepâncias nos seus resultados.

Figura 4.16 Resultados dos testes

CAPÍTULO 5

Conclusão

O objetivo desse trabalho foi realizar uma expansão das capacidades funcionais da biblioteca *CEPSwift* através da implementação de um subconjunto de operadores comuns a bibliotecas e sistemas similares. Isso foi atingido através da adição de operadores categorizados como *flow management operators* [4] e realizando melhorias nos componentes já existentes. Outro objetivo deste trabalho foi a validação das operações até então oferecidas pela biblioteca como também a validação dos novos operadores adicionados. Isso foi atingido através do desenvolvimento de um conjunto de simuladores que permitiram o desenvolvimento de uma *suite* de testes. Tal desenvolvimento permitiu que usuários da biblioteca possam validar operadores específicos de suas áreas de atuação com uma maior facilidade, se tornando não só uma importante ferramenta para o desenvolvimento da biblioteca como também para os usuários da mesma.

Os resultados apresentados pela *suite* de testes mostrou que a biblioteca apresenta algumas falhas em operadores importantes como o operador *merge*, entretanto, o restante dos casos de testes apresentaram sucesso demonstrando que as falhas são isoladas não descreditando a implementação atual por completo.

5.1 Trabalhos futuros

5.1.1 Correção das falhas detectadas

Os problemas detectados pela *suite* de testes são críticos e devem ser resolvidos para que o desenvolvimento da biblioteca continue. Dessa maneira, uma analise deve ser realizada com o intuito de propor novas implementações para tais operadores de modo que sejam solucionadas as falhas discutidas neste trabalho.

5.1.2 Expansão dos testes

Uma *suite* de testes não garante que um sistema está livre de falhas, apenas garante que os casos cobertos pelos teste estão funcionais, dessa maneira a adição de novos testes e a utilização de dados extraídos de sistemas reais apresentaria grande valor na validação da mesma.

Referências Bibliográficas

[1] The reactive manifesto. https://www.reactivemanifesto.org/. Accessed: 2018-06-09. 2.2.1

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

[3] George Belo. Cepswift: Complex event processing framework for swift, 2017. 1.1, 1.2, 2.3.2, 2.5, 3.3

[4] Gianpaolo Cugola and Alessandro Margara. Processing flows of information: From data stream to complex event processing. *ACM Comput. Surv.*, 44(3):15:1–15:62, June 2012. 1, 2.1.2, 2.1.2.1, 2.3, 2.3.2, 3.2, 3.2.1, 3.2.2, 5

[5] Opher Etzion and Peter Niblett. *Event Processing in Action*. Manning Publications Co., Greenwich, CT, USA, 1st edition, 2010. 2.1.1

[6] Erich Gamma. *Design patterns: elements of reusable object-oriented software*. Pearson Education India, 1995. 1

[7] R. Gu, Y. Zhou, Z. Wang, C. Yuan, and Y. Huang. Penguin: Efficient query-based framework for replaying large scale historical data. *IEEE Transactions on Parallel and Distributed Systems*, 29(10):2333–2345, Oct 2018. 2.4.3, 3.3

[8] Alessandro Margara and Guido Salvaneschi. Ways to react: Comparing reactive languages and complex event processing. 2013. 1, 2.1.3, 2.2.3, 2.3

[9] Roy Osherove. *The Art of Unit Testing: With Examples in .Net*. Manning Publications Co., Greenwich, CT, USA, 1st edition, 2009. 2.4.1, 2.4.2, 2.4.3

[10] Adrian Paschke, Alexander Kozlenkov, and Harold Boley. A homogeneous reaction rule language for complex event processing. *CoRR*, abs/1008.0823, 2007. 2.1.1

[11] Quick. Quick project. https://github.com/Quick/Quick. Accessed: 2018-11-28. 4

[12] ReactiveX. Rxswift project. https://github.com/ReactiveX/RxSwift. Accessed: 2018-06-09. 1, 1.1, 2.2.1, 2.2.2, 3.2.3, 3.3

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

[14] Andre Staltz. Rx glitches aren't actually a problem. https://staltz.com/rx-glitches-arent-actually-a-problem.html. Accessed: 2018-10-28. 4.1.5

[15] R.C. Team, F. Pillet, and J. Bontognali. *Rxswift: Reactive Programming with Swift, Second Edition*. Razeware LLC, 2017.

APÊNDICE A

Simuladores

```
public class EventStreamSimulator<T> {
    private var scheduler: TestScheduler

    public init() {
        self.scheduler = TestScheduler(initialClock: 0)
    }

    public func simulate<K>(with events: [EventEntry<T>],
                            handler: @escaping
                             ↪ StreamToStream<T,K>) ->
                             ↪ [EventEntry<K>] {

        let input = events.map(self.recorded)
        let xs = self.scheduler.createHotObservable(input)

        let results = self.scheduler.start(created: 0,
                                           subscribed: 0,
                                           disposed: 1000) {
                                            ↪ () ->
                                            ↪ Observable<K>
                                            ↪ in

            handler(EventStream<T>(withObservable:
             ↪ xs.asObservable())).observable
        }

        return results.events.map (self.entry)
    }

    public func simulate<K>(with events1: [EventEntry<T>],
                          with events2: [EventEntry<T>],
                          handler: @escaping
                           ↪ StreamsToStream<T,K>) ->
                           ↪ [EventEntry<K>] {
```

```
        let input1 = events1.map(self.recorded)
        let input2 = events2.map(self.recorded)

        let xs1 = self.scheduler.createHotObservable(input1)
        let xs2 = self.scheduler.createHotObservable(input2)

        let results = self.scheduler.start(created: 0,
                                           subscribed: 0,
                                           disposed: 1000) {
                                            ↪ () ->
                                            ↪ Observable<K>
                                            ↪ in

            handler(EventStream<T>(withObservable:
             ↪ xs1.asObservable()),
                    EventStream<T>(withObservable:
                     ↪ xs2.asObservable())
            ).observable
        }

        return results.events.map (self.entry)
    }

    private func recorded<K>(from entry: EventEntry<K>) ->
     ↪ Recorded<RxSwift.Event<K>> {
        return next(entry.time, entry.event)
    }

    private func entry<K>(from recorded:
     ↪ Recorded<RxSwift.Event<K>>) -> EventEntry<K> {
        return (time: recorded.time, event:
         ↪ recorded.value.element!)
    }
}

class ComplexEventSimulator {
    typealias StreamsToComplex<T,K> = (EventStream<T>,
     ↪ EventStream<K>) -> ComplexEvent
    typealias StreamToComplex<T> = (EventStream<T>) ->
     ↪ ComplexEvent
    private var scheduler: TestScheduler

    public init() {
        self.scheduler = TestScheduler(initialClock: 0)
```

```
}

public func simulate<T,K>(with events1: [EventEntry<T>],
                          with events2: [EventEntry<K>],
                          handler: @escaping
                            ↪ StreamsToComplex<T,K>
                         ) -> [Int] {

    let input1 = events1.map(recorded)
    let input2 = events2.map(recorded)

    let xs1 = self.scheduler.createHotObservable(input1)
    let xs2 = self.scheduler.createHotObservable(input2)

    let results = self.scheduler.start(created: 0,
                                       subscribed: 0,
                                       disposed: 1000) {
        handler(EventStream<T>(withObservable:
          ↪ xs1.asObservable()),
                EventStream<K>(withObservable:
                  ↪ xs2.asObservable())
            ).startObserving()
    }

    return results.events.map (self.entry)
}

public func simulate<T>(with events: [EventEntry<T>],
                        handler: @escaping
                          ↪ StreamToComplex<T>) -> [Int] {

    let input = events.map(recorded)

    let xs = self.scheduler.createHotObservable(input)

    let results = self.scheduler.start(created: 0,
                                       subscribed: 0,
                                       disposed: 1000) {
        handler(EventStream<T>(withObservable:
          ↪ xs.asObservable())
            ).startObserving()
    }
```

```
        return results.events.map (self.entry)
    }

    private func recorded<K>(from entry: EventEntry<K>) ->
      ↪ Recorded<RxSwift.Event<K>> {

        return next(entry.time, entry.event)
    }

    private func entry<K>(from recorded:
      ↪ Recorded<RxSwift.Event<K>>) -> Int {

        return recorded.time
    }
}
```

APÊNDICE B

Casos de teste

B.1 Caso de teste *map times 10*

```
context("When mapping a EventStream") {
    let input = [
        (time: 1, event: IntEvent(value: 0)),
        (time: 3, event: IntEvent(value: 1)),
        (time: 5, event: IntEvent(value: 2)),
        (time: 5, event: IntEvent(value: 3)),
        ]

    let expectedOutput = [
        (time: 1, event: IntEvent(value: 0)),
        (time: 3, event: IntEvent(value: 10)),
        (time: 5, event: IntEvent(value: 20)),
        (time: 5, event: IntEvent(value: 30)),
        ]

    func mapTimesTen(_ stream: EventStream<IntEvent>) ->
     ↪ EventStream<IntEvent> {
        let resultStream = stream.map { IntEvent(value:
         ↪ $0.value * 10) }

        return resultStream
    }

    let simulator = EventStreamSimulator<IntEvent>()
    let output = simulator.simulate(with: input,
                                    handler: mapTimesTen)

    it("Should output \(expectedOutput.count) events") {
        expect(output.count).to(equal(expectedOutput.count))
    }

    it("Should output the expected events") {
        for (idx, elem) in expectedOutput.enumerated() {
```

```
            let outputElem = output[idx]

            expect(outputElem.time).to(equal(elem.time))
            expect(outputElem.event).to(equal(elem.event))
        }
    }
}
```

B.2 Caso de teste *filter even*

```
context("When filtering a EventStream") {
    let input = [
        (time: 1, event: IntEvent(value: 210)),
        (time: 2, event: IntEvent(value: 251)),
        (time: 4, event: IntEvent(value: 10)),
        (time: 5, event: IntEvent(value: 5)),
        (time: 5, event: IntEvent(value: 15)),
        (time: 8, event: IntEvent(value: 10)),
        (time: 8, event: IntEvent(value: 500))
    ]

    let expectedOutput = [
        (time: 1, event: IntEvent(value: 210)),
        (time: 4, event: IntEvent(value: 10)),
        (time: 8, event: IntEvent(value: 10)),
        (time: 8, event: IntEvent(value: 500))
    ]

    func filterEven(_ stream: EventStream<IntEvent>) ->
     ↪ EventStream<IntEvent> {
        let resultStream = stream.filter { $0.value % 2 == 0 }

        return resultStream
    }

    let simulator = EventStreamSimulator<IntEvent>()
    let output = simulator.simulate(with: input,
                                    handler: filterEven)

    it("Should output \(expectedOutput.count) elements") {
        expect(output.count).to(equal(expectedOutput.count))
    }
```

```
    it("Should output the expected events") {
        for (idx, elem) in expectedOutput.enumerated() {
            let outputElem = output[idx]

            expect(outputElem.time).to(equal(elem.time))
            expect(outputElem.event).to(equal(elem.event))
        }
    }
}
```

B.3 Caso de teste *max*

```
context("When fetching the biggest events of a stream") {
    let input = [
        (time: 1, event: IntEvent(value: 210)),
        (time: 3, event: IntEvent(value: 251)),
        (time: 3, event: IntEvent(value: 10)),
        (time: 3, event: IntEvent(value: 5)),
        (time: 4, event: IntEvent(value: 15)),
        (time: 5, event: IntEvent(value: 10)),
        (time: 7, event: IntEvent(value: 500))
    ]

    let expectedOutput = [
        (time: 1, event: IntEvent(value: 210)),
        (time: 3, event: IntEvent(value: 251)),
        (time: 7, event: IntEvent(value: 500))
    ]

    func max(_ stream: EventStream<IntEvent>) ->
      ↪ EventStream<IntEvent> {
        return stream.max()
    }

    let output = EventStreamSimulator().simulate(with: input,
                                                 handler: max)

    it("Should output \(expectedOutput.count) events") {
        expect(output.count).to(equal(expectedOutput.count))
    }

    it("Should output the expected events") {
        for (idx, elem) in expectedOutput.enumerated() {
```

```
            let outputElem = output[idx]

            expect(outputElem.time).to(equal(elem.time))
            expect(outputElem.event).to(equal(elem.event))
        }
    }
}
```

B.4 Caso de teste *min*

```
context("When fetching the smallest events of a stream") {
    let input = [
        (time: 1, event: IntEvent(value: 50)),
        (time: 1, event: IntEvent(value: 10)),
        (time: 1, event: IntEvent(value: 30)),
        (time: 2, event: IntEvent(value: 30)),
        (time: 3, event: IntEvent(value: 5)),
        ]

    let expectedOutput = [
        (time: 1, event: IntEvent(value: 50)),
        (time: 1, event: IntEvent(value: 10)),
        (time: 3, event: IntEvent(value: 5)),
        ]

    func min(_ stream: EventStream<IntEvent>) ->
      ↪ EventStream<IntEvent> {
        return stream.min()
    }

    let simulator = EventStreamSimulator<IntEvent>()
    let output = simulator.simulate(with: input,
                                    handler: min)

    it("Should output \(expectedOutput.count) events") {
        expect(output.count).to(equal(expectedOutput.count))
    }

    it("Should output the expected events") {
        for (idx, elem) in expectedOutput.enumerated() {
            let outputElem = output[idx]

            expect(outputElem.time).to(equal(elem.time))
```

```
            expect(outputElem.event).to(equal(elem.event))
        }
    }
}
```

B.5 Caso de teste *union*

```
context("When filtering common elements of two EventStreams")
  ↪ {
    let input1 = [
        (time: 2, event: IntEvent(value: 1)),
        (time: 2, event: IntEvent(value: 2)),
        (time: 3, event: IntEvent(value: 4)),
        (time: 4, event: IntEvent(value: 8)),
        (time: 5, event: IntEvent(value: 16)),
        (time: 6, event: IntEvent(value: 32)),
        ]

    let input2 = [
        (time: 1, event: IntEvent(value: 1)),
        (time: 1, event: IntEvent(value: 2)),
        (time: 3, event: IntEvent(value: 16)),
        (time: 5, event: IntEvent(value: 32)),
        (time: 7, event: IntEvent(value: 64))
    ]

    let expectedOutput = [
        (time: 1, event: IntEvent(value: 1)),
        (time: 1, event: IntEvent(value: 2)),
        (time: 3, event: IntEvent(value: 4)),
        (time: 3, event: IntEvent(value: 16)),
        (time: 4, event: IntEvent(value: 8)),
        (time: 5, event: IntEvent(value: 32)),
        (time: 7, event: IntEvent(value: 64))
    ]

    func union(_ stream1: EventStream<IntEvent>,
               _ stream2: EventStream<IntEvent>) ->
                 ↪ EventStream<IntEvent> {
        return stream1.union(with: stream2)
    }

    let simulator = EventStreamSimulator<IntEvent>()
```

```
    let output = simulator.simulate(with: input1,
                                    with: input2,
                                    handler: union)

    it("Should output \(expectedOutput.count) elements") {
        expect(output.count).to(equal(expectedOutput.count))
    }

    it("Should output the expected elements)") {
        for (index, elem) in expectedOutput.enumerated() {
            let outputElem = output[index]

            expect(outputElem.time).to(equal(elem.time))
            expect(outputElem.event).to(equal(elem.event))
        }
    }
}
```

B.6 Caso de teste *find common*

```
context ("When intersecting two EventStreams") {
    let input1 = [
        (time: 1, event: IntEvent(value: 1)),
        (time: 2, event: IntEvent(value: 2)),
        (time: 3, event: IntEvent(value: 4)),
        (time: 4, event: IntEvent(value: 8)),
        (time: 5, event: IntEvent(value: 16)),
        (time: 6, event: IntEvent(value: 32)),
        (time: 7, event: IntEvent(value: 64))
    ]

    let input2 = [
        (time: 1, event: IntEvent(value: 0)),
        (time: 1, event: IntEvent(value: 32)),
        (time: 3, event: IntEvent(value: 99)),
        (time: 4, event: IntEvent(value: 8)),
        (time: 5, event: IntEvent(value: 1)),
        (time: 5, event: IntEvent(value: 100))
    ]

    let expectedOutput = [
        (time: 4, event: IntEvent(value: 8)),
        (time: 5, event: IntEvent(value: 1)),
```

```
            (time: 6, event: IntEvent(value: 32))
        ]

        func intersect(_ stream1: EventStream<IntEvent>,
                       _ stream2: EventStream<IntEvent>) ->
                         ↪ EventStream<IntEvent> {
            return stream1.intersect(with: stream2)
        }

        let simulator = EventStreamSimulator<IntEvent>()
        let output = simulator.simulate(with: input1,
                                        with: input2,
                                        handler: intersect)

        it("Should output \(expectedOutput.count) elements") {
            expect(output.count).to(equal(expectedOutput.count))
        }

        it("Should output the expected elements)") {
            for (index, elem) in expectedOutput.enumerated() {
                let outputElem = output[index]

                expect(outputElem.time).to(equal(elem.time))
                expect(outputElem.event).to(equal(elem.event))
            }
        }
    }
```

B.7 Caso de teste *unique elements*

```
    context("When subtracting two EventStreams") {
        let input1 = [
            (time: 2, event: IntEvent(value: 1)),
            (time: 2, event: IntEvent(value: 2)),
            (time: 3, event: IntEvent(value: 4)),
            (time: 4, event: IntEvent(value: 8)),
            (time: 5, event: IntEvent(value: 16)),
            (time: 6, event: IntEvent(value: 32)),
            (time: 7, event: IntEvent(value: 64))
        ]
        let input2 = [
            (time: 1, event: IntEvent(value: 1)),
            (time: 1, event: IntEvent(value: 2)),
```

```
            (time: 3, event: IntEvent(value: 16)),
            (time: 5, event: IntEvent(value: 32)),
            ]

        let expectedOutput = [
            (time: 3, event: IntEvent(value: 4)),
            (time: 4, event: IntEvent(value: 8)),
            (time: 7, event: IntEvent(value: 64))
        ]

        func notIn(_ stream1: EventStream<IntEvent>,
                   _ stream2: EventStream<IntEvent>) ->
                     ↪ EventStream<IntEvent> {
            return stream1.not(in: stream2)
        }

        let simulator = EventStreamSimulator<IntEvent>()
        let output = simulator.simulate(with: input1,
                                        with: input2,
                                        handler: notIn)

        it("Should output \(expectedOutput.count) elements") {
            expect(output.count).to(equal(expectedOutput.count))
        }

        it("Should output the expected elements)") {
            for (index, elem) in expectedOutput.enumerated() {
                let outputElem = output[index]

                expect(outputElem.time).to(equal(elem.time))
                expect(outputElem.event).to(equal(elem.event))
            }
        }
}
```

B.8 Caso de teste *filter increasing events*

```
context("When filtering tuples of increasing value") {
    let input = [
        (time: 1, event: IntEvent(value: 0)),
        (time: 2, event: IntEvent(value: 1)),
        (time: 5, event: IntEvent(value: 2)),
        (time: 5, event: IntEvent(value: 1)),
```

```
            (time: 5, event: IntEvent(value: 3)),
            (time: 5, event: IntEvent(value: 4)),
            (time: 5, event: IntEvent(value: 5)),
            (time: 6, event: IntEvent(value: 3)),
            ]
        let expectedOutput = [
            (time: 2, event: (IntEvent(value: 0), IntEvent(value:
             ↪ 1))),
            (time: 5, event: (IntEvent(value: 1), IntEvent(value:
             ↪ 2))),
            (time: 5, event: (IntEvent(value: 1), IntEvent(value:
             ↪ 3))),
            (time: 5, event: (IntEvent(value: 3), IntEvent(value:
             ↪ 4))),
            (time: 5, event: (IntEvent(value: 4),IntEvent(value:
             ↪ 5))),
            ]

        func followedBy(_ stream: EventStream<IntEvent>) ->
         ↪ EventStream<(IntEvent, IntEvent)> {
            return stream.followedBy { $0.value < $1.value }
        }

        let simulator = EventStreamSimulator<IntEvent>()
        let output = simulator.simulate(with: input, handler:
         ↪ followedBy)

        it("Should output \(expectedOutput.count) elements") {
            expect(output.count).to(equal(expectedOutput.count))
        }

        it("Should output the expected events") {
            zip(output, expectedOutput).forEach {
                expect($0.0.time).to(equal($0.1.time))
                expect($0.0.event.0).to(equal($0.1.event.0))
                expect($0.0.event.1).to(equal($0.1.event.1))
            }
        }
}
```

B.9 Caso de teste *drop repeated events*

```
context("When filtering unique elements of a stream") {
    let input = [
        (time: 1, event: IntEvent(value: 210)),
        (time: 3, event: IntEvent(value: 251)),
        (time: 3, event: IntEvent(value: 10)),
        (time: 4, event: IntEvent(value: 5)),
        (time: 4, event: IntEvent(value: 15)),
        (time: 4, event: IntEvent(value: 10)),
        (time: 6, event: IntEvent(value: 251))
    ]

    let expectedOutput = [
        (time: 1, event: IntEvent(value: 210)),
        (time: 3, event: IntEvent(value: 251)),
        (time: 3, event: IntEvent(value: 10)),
        (time: 4, event: IntEvent(value: 5)),
        (time: 4, event: IntEvent(value: 15)),
        ]

    let output = EventStreamSimulator().simulate(with: input)
     ↪ { (stream: EventStream<IntEvent>) in
        return stream.dropDuplicates()
    }

    it("Should output \(expectedOutput.count) events") {
        expect(output.count).to(equal(expectedOutput.count))
    }

    it("Should output the expected events") {
        for (idx, elem) in expectedOutput.enumerated() {
            let outputElem = output[idx]

            expect(outputElem.time).to(equal(elem.time))
            expect(outputElem.event).to(equal(elem.event))
        }
    }
}
```

B.10 Caso de teste *grouping events every two seconds*

```
context("When applying a window over a EventStream") {
    let input = [
        (time: 1, event: IntEvent(value: 0)),
        (time: 2, event: IntEvent(value: 1)),
        (time: 5, event: IntEvent(value: 2)),
        (time: 5, event: IntEvent(value: 3)),
        (time: 5, event: IntEvent(value: 4)),
        (time: 5, event: IntEvent(value: 5)),
        (time: 6, event: IntEvent(value: 3)),
        ]

    let expectedOutput = [(time: 2, event: [IntEvent(value:
     ↪ 3),
                                            IntEvent(value:
                                             ↪ 1)]),
                          (time: 5, event: [IntEvent(value:
                           ↪ 2),
                                            IntEvent(value:
                                             ↪ 3),
                                            IntEvent(value:
                                             ↪ 4)]),
                          (time: 6, event: [IntEvent(value:
                           ↪ 5),
                                            IntEvent(value:
                                             ↪ 3)])]

    func window(_ stream: EventStream<IntEvent>) ->
     ↪ EventStream<[IntEvent]> {
        let resultStream = stream.window(ofTime: 2, max: 3)

        return resultStream
    }

    let simulator = EventStreamSimulator<IntEvent>()
    let output = simulator.simulate(with: input,
                                    handler: window)

    it("Should output \(expectedOutput.count) elements") {
        expect(output.count).to(equal(expectedOutput.count))
    }
```

```
    it("Should output the expected events") {
        zip(output, expectedOutput).forEach {
            expect($0.0.time).to(equal($0.1.time))
            expect($0.0.event).to(equal($0.1.event))
        }
    }
}
```

B.11 Caso de teste *group even and odd events*

```
context("When grouping even and odd events") {
    let input = [
        (time: 1, event: IntEvent(value: 0)),
        (time: 3, event: IntEvent(value: 1)),
        (time: 5, event: IntEvent(value: 2)),
        (time: 5, event: IntEvent(value: 3)),
        ]

    let expectedOutput = [
        (time: 1, event: [true: [IntEvent(value: 0)]]),
        (time: 3, event: [true: [IntEvent(value: 0)],
                          false: [IntEvent(value: 1)]]),
        (time: 5, event: [true: [IntEvent(value: 0),
                                 IntEvent(value: 2)],
                          false: [IntEvent(value: 1)]]),
        (time: 5, event: [true: [IntEvent(value: 0),
                                 IntEvent(value: 2)],
                          false: [IntEvent(value: 1),
                                  IntEvent(value: 3)]]),
        ]

    func groupedByEvenOrNot(stream: EventStream<IntEvent>) ->
      ↪ EventStream<[Bool: [IntEvent]]> {
        return stream.group { $0.value % 2 == 0}
    }

    let simulator = EventStreamSimulator<IntEvent>()
    let output = simulator.simulate(with: input,
                                    handler:
                                      ↪ groupedByEvenOrNot)

    it("Should output \(expectedOutput.count) elements") {
        expect(output.count).to(equal(expectedOutput.count))
```

```
    }

    it("Should output the expected events") {
        for (idx, elem) in expectedOutput.enumerated() {
            let outputElem = output[idx]

            expect(outputElem.time).to(equal(elem.time))
            expect(outputElem.event).to(equal(elem.event))
        }
    }
}
```

B.12 Caso de teste *sort events in increasing order*

```
context("When ordering a EventStream") {
    let input = [
        (time: 1, event: IntEvent(value: 210)),
        (time: 3, event: IntEvent(value: 251)),
        (time: 3, event: IntEvent(value: 10)),
        (time: 3, event: IntEvent(value: 5)),
        (time: 4, event: IntEvent(value: 15)),
        (time: 5, event: IntEvent(value: 10)),
        (time: 7, event: IntEvent(value: 500))
    ]

    let expectedOutput = [
        (time: 1, event: [IntEvent(value: 210)]),
        (time: 3, event: [IntEvent(value: 210),
                          IntEvent(value: 251)]),
        (time: 3, event: [IntEvent(value: 10),
                          IntEvent(value: 210),
                          IntEvent(value: 251)]),
        (time: 3, event: [IntEvent(value: 5),
                          IntEvent(value: 10),
                          IntEvent(value: 210),
                          IntEvent(value: 251)]),
        (time: 4, event: [IntEvent(value: 5),
                          IntEvent(value: 10),
                          IntEvent(value: 15),
                          IntEvent(value: 210),
                          IntEvent(value: 251)]),
        (time: 5, event: [IntEvent(value: 5),
                          IntEvent(value: 10),
```

```
                            IntEvent(value: 10),
                            IntEvent(value: 15),
                            IntEvent(value: 210),
                            IntEvent(value: 251)]),
        (time: 7, event: [IntEvent(value: 5),
                            IntEvent(value: 10),
                            IntEvent(value: 15),
                            IntEvent(value: 210),
                            IntEvent(value: 251),
                            IntEvent(value: 500)])
    ]


    func ordered(_ stream: EventStream<IntEvent>) ->
     ↪ EventStream<[IntEvent]> {
        return stream.ordered { $0.value <= $1.value}
    }

    let output = EventStreamSimulator().simulate(with: input,
                                                 handler:
                                                  ↪ ordered)

    it("Should output \(expectedOutput.count) events") {
        expect(output.count).to(equal(expectedOutput.count))
    }

    it("Should output the expected events") {
        for (idx, elem) in expectedOutput.enumerated() {
            let outputElem = output[idx]

            expect(outputElem.time).to(equal(elem.time))
            expect(outputElem.event).to(equal(elem.event))
        }
    }
}
```

B.13 Caso de teste *ComplexEvents from Int and String events*

```
context("When merging two EventStreams") {
    let input1 = [(time: 205, event: IntEvent(value: 10)),
                  (time: 210, event: IntEvent(value: 20)),
                  (time: 225, event: IntEvent(value: 30)),
                  (time: 227, event: IntEvent(value: 30)),
```

```
                (time: 228, event: IntEvent(value: 30))]

    let input2 = [(time: 206, event: StringEvent(value:
     ↪ "10")),
                (time: 213, event: StringEvent(value:
                 ↪ "20")),
                (time: 220, event: StringEvent(value:
                 ↪ "30")),
                (time: 230, event: StringEvent(value:
                 ↪ "40"))]

    let expectedOutput = [206, 213, 225, 230]

    func merge<T, K>(_ stream1: EventStream<T>,
                     _ stream2: EventStream<K>) ->
                      ↪ ComplexEvent {
        return stream1.merge(with: stream2)
    }


    let simulator = ComplexEventSimulator()
    let output = simulator.simulate(with: input1,
                                    with: input2,
                                    handler: merge)

    it("Should output \(expectedOutput.count) elements") {
        expect(output.count).to(equal(expectedOutput.count))
    }

    if output.count == expectedOutput.count {
        it("Should output the expected events") {
            zip(output, expectedOutput).forEach {
                expect($0.0).to(equal($0.1))
            }
        }
    }
}
```

Este volume foi tipografado em LaTeX na classe UFPEThesis (www.cin.ufpe.br/~paguso/ufpethesis).
Para detalhes sobre este documento, clique aqui.