Random testing basado en lógica temporal para Apache Flink

Random testing based on temporal logic for Apache Flink

Cristina Valentina Espinosa Victoria

MÁSTER EN INGENIERÍA INFORMÁTICA

FACULTAD DE INFORMÁTICA

DEPARTAMENTO DE SISTEMAS INFORMÁTICOS Y COMPUTACIÓN

UNIVERSIDAD COMPLUTENSE DE MADRID

TRABAJO FIN DE MÁSTER EN INGENIERÍA INFORMÁTICA

Madrid, 14 de enero de 2019

Directores:

Adrián Riesco Rodríguez

Enrique Martín Martín

Convocatoria enero-febrero 2019

Calificación: Sobresaliente - 10

Índice

Resumen

Random testing basado en lógica temporal para Apache Flink

Actualmente, existen muy pocas alternativas para probar los sistemas de *stream processing*, consistiendo la mayoría de ellas en tests de unidad, los cuales no son viables en casos en los que se requiera disponer de una gran cantidad de *streams* o *streams* de gran longitud, ya que es necesario definir cada *stream* de entrada y el esperado de salida. Es por esto que surgió la idea de implementar un programa que pueda aplicar *random testing*, una técnica de *testing* basada en propiedades en lugar de en coberturas, a uno de estos sistemas de *stream processing*, incorporando además propiedades de lógica temporal. Además, ha sido empíricamente demostrado que el *random testing* funciona igual o incluso mejor que las técnicas de cobertura, lo que inclina la balanza a favor de esta técnica de *testing*.

En este trabajo presentamos una herramienta para realizar *testing* basado en propiedades para Apache Flink, un sistema de *stream processing* capaz de procesar datos a tiempo real. Esto significa que Flink trata los datos a medida que van siendo generados y en el momento en que son recibidos. Para desarrollar esta herramienta se ha utilizado el lenguaje Scala y la técnica de *random testing* combinada con lógica temporal.

Ya existe un entorno con la misma filosofía que el aquí presentado dirigido a Spark Streaming, Sscheck. Sin embargo, Spark maneja lotes y no tiene, por tanto, tiempo real. Por ello, el objetivo de este proyecto es implementar ese mismo programa pero adaptado para trabajar con Apache Flink, aunque presenta diferentes problemas, como menor flexibilidad en el tratamiento de los datos.

Palabras clave

Apache Flink, lógica temporal, entornos distribuidos, Big Data, random testing, Scala, stream processing, tiempo real.

Abstract

Random testing based on temporal logic for Apache Flink

Nowadays, there are very few alternatives for stream processing systems testing, most of them consisting of unit tests, which are not viable when it is required a large number of streams or very long streams, because it is necessary to define every stream received as input and every stream expected as corresponding output. Because of this situation, the idea of implementing a program that can use random testing, a testing technique based in properties instead of coverage, with one of these processing systems, came up, also adding temporal logic properties. Furthermore, it has been empirically proved that random testing works as well as, or even better than, coverage techniques, something that makes the choice of using this technique even more attractive.

In this Master's Thesis we present a property-based testing tool for Apache Flink, a stream processing system capable of processing data in real time. This means that Flink treats data as it is generated and as it is received. To develop this tool we have used the Scala language and random testing techniques combined with temporal logic.

There exists an environment with the same philosophy as the one presented here applied to Spark Streaming, Sscheck. However, Spark handles batches and therefore has no real time. Therefore, the aim of this project is to implement the same program but adapted to Apache Flink. It presents different problems, such as less flexibility in data processing.

Key words

Apache Flink, temporal logic, distributed environments, Big Data, random testing, Scala, stream processing, real time.

Capítulo 1: Introducción

En este primer capítulo se procede a explicar y dar a conocer los puntos más importantes en los que se basa el proyecto, así como su motivación, con el fin de ofrecer una visión general del mismo.

1.1 Descripción del proyecto

Hoy en día, debido a la expansión de las nuevas tecnologías, se generan enormes cantidades de datos a cada momento. Datos de todo tipo, de los cuales se puede obtener muchísima información, a la que se le pueden dar numerosas utilidades, encontrando entre ellas aplicaciones tan diferentes como calcular qué productos se venderán mejor en un supermercado a partir de los gustos y hábitos de los clientes, o predecir una infección gracias a los datos obtenidos de la monitorización constante de un paciente [1]. Son muchos los conocedores del gran provecho que pueden obtener manejando estos datos de la forma correcta y que, además, ya hacen uso de ellos. Por ejemplo, Amazon hace uso de la información de la que dispone sobre sus clientes tanto para mejorar sus algoritmos de publicidad como para mejorar el servicio al cliente, y Netflix aprovecha los datos sobre la popularidad de cada película y serie para decidir qué nuevo contenido añadir [2].

Sin embargo, los sistemas de almacenamiento masivo no siempre son capaces de enfrentarse a esta gran cantidad de datos. Además, hay situaciones en las que incluso es necesario ser capaz de manejar los datos de forma constante y a medida que estos se generan, como en el caso de la monitorización de pacientes para prevenir infecciones. Es por eso por lo que cada vez son más necesarios sistemas que permitan y faciliten el procesamiento de flujos de datos (en esta memoria referidos como *streams*). Estos sistemas son llamados sistemas de *stream processing*, capaces de procesar datos en movimiento, es decir, manejar los datos en el momento en que estos son generados o recibidos, y evitando así tener que almacenarlos para poder hacer uso de ellos [3].

Flink es uno de estos sistemas de *stream processing*; recibe un *stream* de datos, ofrece herramientas al usuario para facilitar el manejo y transformación de la información recibida a través de dicho *stream*, los datos se van procesando a medida que entran en el sistema, y el resultado se devuelve en otro *stream* de salida, el cual puede ser procesado nuevamente si así se desea. El funcionamiento de Flink se explicará más a fondo en el capítulo 3 de esta memoria.

En el ámbito de la informática, el *testing* de *software* se refiere a toda actividad llevada a cabo con el fin de verificar que los resultados obtenidos encajan con los resultados esperados, y por tanto el *software* funciona correctamente y sin fallos. Además, el *testing* también ayuda a encontrar errores en la implementación, posibles situaciones que no se tuvieron en cuenta, o incluso la falta de algún requisito [4].

Ningún *testing* es capaz de garantizar al 100% que un *software* esté totalmente libre de

fallos, ya que el *testing* en realidad ayuda a confirmar la existencia de errores, no la ausencia de ellos, pero sin *testing*, estos errores en la implementación no serían detectados. Por lo tanto, disponer de métodos de *testing* sólidos para los sistemas de *stream processing* y así ganar confianza en que el procesado de los datos es correcto y, en consecuencia, la información obtenida del mismo lo más fiable posible, es de gran importancia

Sin embargo, en general existen muy pocas alternativas de *testing* para los sistemas de *stream processing*, que tan útiles y necesarios son hoy en día. Básicamente solo se dispone de tests de unidad, lo cual resulta inviable en los casos para los que más interesa hacer uso de este tipo de sistemas, ya que en dichas situaciones se tiene una enorme cantidad de datos y los tests de unidad requieren definir un *stream* realista de entrada y su correspondiente *stream* de salida.

Por este motivo surge la idea principal de este proyecto, que es adaptar la herramienta Sscheck, una aplicación que permite hacer *random testing* en el sistema de *stream processing* Spark Streaming, con el fin de poder hacer *random testing* en el entorno distribuido de Flink, el cual maneja un tiempo más real que Spark.

1.2 Estado del arte

A continuación se presentan los desarrollos predecesores a este, destacando aquellas características que comparten con el proyecto actual.

1.2.1 QuickCheck

QuickCheck es una herramienta de *testing* para Haskell desarrollada por Koen Claessen y John Hughes. La idea de esta herramienta es, en lugar de definir una serie de datos de entrada para comprobar que la salida es la esperada, definir una lista de propiedades que la función debe cumplir. En otras palabras, sigue una técnica de *testing* basado en propiedades en vez de utilizar tests de unidad [5].

Para probar estas propiedades, la herramienta genera una serie de datos aleatorios, permitiendo así tener un gran número de casos de prueba para probar el *software.* Estos datos aleatorios se obtienen de los generadores que proporciona QuickCheck, a partir de los cuales pueden hacerse otros más complejos, adaptados a las necesidades de cada programa. Un generador es, simplemente, una función que recibe varios o ningún parámetro y produce datos aleatorios dependientes de estos. Cuando en esta memoria se utilice el término ‘generador’ se estará refiriendo a esta misma definición.

Para realizar el *testing*, QuickCheck utiliza la ténica de *random testing;* genera de forma automática tantos datos como se deseen a partir de los generadores, las funciones a probar reciben estos datos como entrada, y se comprueba que las salidas cumplan las propiedades definidas.

Como resultado, tenemos un gran número de pruebas realizadas con datos diferentes cada vez, lo que facilita el encontrar errores imprevistos con gran rapidez.

Son muchos los lenguajes de programación a los que se ha extendido esta herramienta, Python (Hypothesis), Java (QuickTheories) y Scala (ScalaCheck), siendo este último el escogido para este proyecto.

1.2.2 Sscheck

Sscheck [6] es una herramienta de *testing* desarrollada por Adrián Riesco y Juan Rodríguez-Hortalá que, haciendo uso de lógica lineal temporal y la biblioteca ScalaCheck, permite probar programas de Spark Streaming con técnicas de *random testing*. La motivación de Sscheck es la ya explicada en el punto anterior de esta memoria; ofrecer una alternativa viable para poder hacer *testing* en sistemas de *stream processing*.

Además, para el desarrollo de Sscheck Adrián Riesco y Juan Rodríguez-Hortalá idearon una lógica que encaje con la aplicación de la técnica de *random testing* a sistemas de *stream processing*, debido a los problemas encontrados al utilizar Lógica Lineal Temporal. Esta lógica, llamada Lógica Lineal Temporal para Sistemas de Streaming (LTLss) y también utilizada en el presente proyecto, extiende LTL3 añadiendo temporizadores (representados como números naturales) a los operadores temporales (excepto al operador siguiente) para definir el número de instantes (que en Spark Streaming no se corresponde a tiempo, sino a lotes) dentro del cual debe cumplirse una propiedad. En el capítulo 3 de esta memoria se desarrolla más a fondo el funcionamiento y motivación de la lógica LTLss.

Por otro lado, también existe otro proyecto anterior, realizado por Max Arnulfo Tello Ortiz, el cual extiende la funcionalidad de Sscheck con el fin de permitir cargar y guardar casos de test y así poder reproducir los errores encontrados gracias al testeo realizado con Sscheck, tarea difícil en caso de no poder guardar los casos de test debido a que los datos generados durante el *testing* son aleatorios [7].

El proyecto tratado en la presente memoria se basa la herramienta Sscheck, sin la extensión anterior presente, adaptando su código [8] para que este funcione sobre Flink .

El resto de la memoria se estructura como sigue:

- El capítulo 2 expone todos los objetivos del proyecto y el plan de trabajo seguido para alcanzarlos.
- El capítulo 3 trata los preliminares, donde se desarrollan en profundidad los conceptos principales manejados en el proyecto.
- En el capítulo 4 se explican todos los detalles sobre la implementación de la herramienta de *testing* desarrollada.
- El capítulo 5 incluye información detallada sobre implementaciones que utilizan técnicas de *testing* basado en lógica temporal pero no hacen uso de *random testing*.
- En el capítulo 6 pueden encontrarse los resultados obtenidos, acompañados de un razonamiento para su entendimiento.
- Finalmente, el capítulo 7 contiene las conclusiones derivadas de la finalización del proyecto, así como posibles ideas y mejoras que podrían incluirse en el futuro.

El código de la aplicación está disponible en:

https://github.com/Valcev/TFM

Capítulo 2: Objetivos y plan de trabajo

En este capítulo se presentan los objetivos del proyecto, así como la organización de sus diferentes fases y las actividades realizadas para completarlos.

2.1 Objetivos generales del proyecto

El objetivo principal de este proyecto consiste en la adaptación del proyecto Sscheck al entorno distribuido de Flink, para así poder aplicar la técnica de *random testing* con Lógica Temporal a Flink. En este objetivo se incluye el diseño, implementación y *testing* de la adaptación a realizar.

Este objetivo principal puede dividirse en los siguientes objetivos generales:

- Profundización en la Lógica Lineal Temporal y en sus aplicaciones prácticas en *testing* de *software* y otros usos que se le puede dar en sistemas de *streaming*, así como en su implementación mediante lenguajes de programación.
- Estudio de Flink y su funcionamiento. Aprender el uso de su *framework* y API, de su tratamiento de los datos en *stream* y del manejo las ventanas. También es necesario profundizar en el funcionamiento interno de Flink. Con esto se consigue llegar a un nivel de desarrollo en la plataforma más avanzado en el que, como se verá más adelante, incluye entender cómo funciona realmente la división del *stream* en ventanas. Llegar a este nivel de profundidad es necesario para poder implementar los generadores que generan los datos de prueba y aplicar las fórmulas correctamente.
- Aprendizaje del lenguaje Scala y de sus características funcionales. También se aprenderá sobre bibliotecas de Scala usadas en este proyecto y en Sscheck, como ScalaCheck y Specs2, para facilitar el *testing*, y Scalaz, para ampliar la sintaxis de Scala y hacer más fácil la escritura de fórmulas de Lógica Temporal.
- Estudio y comprensión de la funcionalidad y del código del proyecto ya desarrollado, Sscheck. En este apartado se incluye aprender la estructura principal del proyecto, dividirlo en distintas partes que se tendrán que replicar en Flink, identificar aquellos elementos que puedan reutilizarse, y aprender también de la aplicación de fórmulas lógicas temporales en Spark.
- Diseño y desarrollo de la solución para Flink. Este objetivo cuenta con los siguientes subobjetivos:
    - Desarrollo de generadores de datos análogos a los desarrollados en el proyecto Sscheck. Los generadores deben usar ScalaCheck para producir datos en forma de ventanas de Flink que cumplan ciertas condiciones indicadas mediante operadores de Lógica Temporal.
    - Desarrollo de una función `test` que evalúe una fórmula de Lógica Temporal aplicada a una colección de ventanas generada por los generadores mencionados en el punto inmediatamente anterior.

    - Diseño e implementación de casos de prueba en los que se apliquen generadores y la función `test` para probar su funcionamiento.
- Documentación del desarrollo y de lo aprendido, con el fin de crear una memoria del proyecto.

2.2 Plan de trabajo

El plan de trabajo ha sido diseñado con ayuda de los directores, ordenando los objetivos, estableciendo dependencias entre ellos, y fijando tareas que realizar para cada uno de ellos.

En general, se puede dividir el proyecto en seis fases principales, cada una dividida en subfases correspondiendo con distintos objetivos del proyecto. Las fases principales son:

1. Investigación y estudio de tecnologías. Esta fase comenzó con el aprendizaje del lenguaje Scala [9] y el estudio de Lógica Temporal, seguido del estudio del proyecto Sscheck y el entorno Flink. También fue necesario aprender a utilizar GitHub [10].
2. Desarrollo de una aplicación sencilla para Flink. Esta aplicación intentará comprobar que se cumplen ciertas condiciones en un flujo de datos, pero sin usar las técnicas de Lógica Temporal para *random testing*. El objetivo es que este desarrollo sirva tanto como una toma de contacto con Flink y puesta en práctica de lo aprendido en la fase anterior, como para una comparativa cuando se desarrolle el mismo caso de prueba, esta vez ya utilizando técnicas de *random testing* mediante Lógica Temporal.
3. Implementación de los generadores de pruebas de manera análoga a los desarrollados en el proyecto Sscheck, llevando a cabo las adaptaciones necesarias para su uso en Flink.
4. Implementación de la función `test` de manera similar a la existente en el proyecto Ssheck, haciendo uso de los generadores desarrollados.
5. Diseño e implementación de pruebas que pongan en práctica y muestren las capacidades de la funcionalidad desarrollada.
6. Documentación y escritura de la memoria del proyecto.

Aunque se hizo esta división en fases, al final algunas estuvieron solapadas entre sí. En particular ocurrieron los siguientes solapes:

- La fase 1 acabó solapándose con todas las demás, ya que había mucho que investigar y siempre surgían problemas que no se sabía cómo resolver, teniendo que consultar documentación de Flink y de Sscheck de forma recurrente.
- La fase 6 se solapó con las fases 4 y 5. En estas fases se alcanzaron ciertos bloqueos y se decidió ir escribiendo la memoria mientras se investigaba cómo desbloquear la situación.

Los solapes no supusieron un impacto notable en el desarrollo del proyecto y nunca se

produjeron de manera forzada. La necesidad de volver a investigar funcionalidad de Flink o buscar si la solución a un problema surgido ya se había resuelto en Sscheck surgió de manera natural a medida que se avanzaba en el proyecto. Ir redactando la memoria cuando se acercaba el final del proyecto no rompió ningún esquema, ya que se había ido creando documentación a lo largo del desarrollo del proyecto y tan solo había que redactarla formalmente.

A continuación se muestra en forma de tabla una división en subfases y tareas de las fases principales mencionadas:

FASE I:

	TAREAS
Repaso y profundización en Lógica Temporal y *random testing*	1. Repaso de los apuntes de Lógica Temporal de la asignatura ACFI. 2. Aprendizaje de la Lógica Lineal Temporal para sistemas de *streaming*. 3. Aprendizaje de la técnica *random testing* y sus ventajas y desventajas con respecto a otras técnicas de *testing*.
Aprendizaje de Scala y ScalaCheck	1. Realización de un curso de Coursera de Scala [9]. 2. Estudio de Scalacheck y su uso en el proyecto Sscheck.
Apache Flink	1. Conocer qué es Flink y su modelo de tratamiento de datos. 2. Buscar las diferencias entre Flink y Spark. 3. Conocer más a fondo la implementación de Flink, en particular la generación de datos para flujos, la implementación de funciones personalizadas para ventanas y la creación de disparadores (*triggers*) de ventanas globales propios.
Proyecto Sscheck	1. Conocer la finalidad del proyecto. 2. Averiguar la arquitectura y dividir el proyecto en conceptos principales. 3. Estudiar el flujo de conexión entre los conceptos principales.

	4. Para cada concepto, investigar las clases relacionadas con ellos y su implementación.

FASE II:

	TAREAS
Definición de la aplicación de toma de contacto con Flink	1. Tratar con los directores del proyecto distintas ideas a implementar y elegir una de ellas. 2. Elaborar la idea elegida para que en la solución de esta se usen una variedad de las principales funcionalidades para el tratamiento de flujos ofrecidas por Flink.
Proceso de generación de datos	1. Desarrollo de un generador de datos mediante colecciones en Flink. 2. Desarrollo de un generador más complejo en Python, que genera y envía los datos al *socket* en el que se encuentre escuchando Flink, haciendo uso de hilos y números aleatorios. 3. Proceso de escucha en Flink. 4. Desarrollo de un tercer generador, imitando el implementado en Python pero utilizando el lenguaje Scala.
Proceso de tratamiento de datos	1. Diseño del proceso de separación en *streams* y en ventanas. 2. Diseño de las funciones a aplicar en las ventanas. 3. Implementación de la solución.

FASE III:

	TAREAS
Investigación y análisis	1. Establecer las diferencias entre los generadores ya implementados para Spark y los que se deben implementar en Flink. 2. Establecer paralelismos y similitudes entre lo desarrollado para Spark y lo que se debe desarrollar para Flink. 3. Buscar en la implementación de los generadores de Sscheck ideas y código que puedan ser reutilizadas para el proyecto.
Diseño	1. Establecer las entradas y las salidas de los métodos generadores. 2. Aplicar el diseño orientado a objetos para representar las entradas, salidas y datos intermedios de dichos métodos.
Implementación	1. Implementación de los métodos auxiliares de generación de colecciones. 2. Implementación de la clase **`ListStream`**. 3. Implementación de los métodos propiamente generadores de datos que usan sintaxis de Lógica Temporal. 4. Implementación del método que convierte el resultado en una colección de ventanas.
Pruebas	1. Realización de pruebas globales de cada uno de los métodos generadores. 2. Realización de pruebas del paso de los datos generados a ventanas.

FASE IV:

	TAREAS
Investigación y análisis	1. Establecer las diferencias entre el tratamiento de **`RDDs`** de Spark y de ventanas en Flink, y las implicaciones que eso tiene. 2. Estudio de cómo se ha realizado la función **`test`** en el proyecto Sscheck. 3. Estudio de la representación de las fórmulas de Lógica Temporal en Sscheck. 4. Búsqueda de elementos reutilizables, ya sea mediante adaptaciones o sin necesidad de cambios.
Diseño	1. Diseño iterativo (añadir funcionalidad poco a poco) de las funciones y clases necesarias para la implementación de la función **`test`**. 2. Estudio de posibles cambios necesarios de la clase Formula reutilizada del proyecto Sscheck. 3. Diseño del formato de salida de la función.
Implementación	1. Implementación iterativa de las distintas ideas para conseguir la función ***`test`***. Esto incluye: a. Implementación de las funciones a aplicar a las ventanas. b. Implementación de la función **`test`** usando las funciones anteriores. c. Implementación de una función que procese los resultados obtenidos para mostrarlos de forma entendible.
Pruebas	1. Realización de pruebas de las distintas funciones **`test`** realizadas en cada iteración.

FASE V:

	TAREAS
Definición de enunciados y pruebas de concepto	1. Búsqueda, junto con los directores del proyecto, de problemas a los que poder aplicar técnicas de *random testing*. 2. Escritura de la narrativa de los problemas.
Análisis y Diseño	1. Análisis del problema y establecimiento del formato de los datos para representar el escenario. 2. Diseño de los generadores de datos sobre los que realizar el *testing*. 3. Diseño de las fórmulas de Lógica Temporal que se van a utilizar.
Implementación	1. Implementación de los generadores. 2. Implementación de las fórmulas.
Pruebas	1. Realización de pruebas atómicas de los generadores. 2. Pruebas atómicas de las fórmulas. 3. Ejecución global de las pruebas sobre las pruebas previamente diseñadas e implementadas.

FASE VI:

	TAREAS
Redacción de la memoria	1. Establecer la estructura de la memoria. 2. Redacción de los distintos apartados. 3. Revisión con los directores y corrección de errores. 4. Maquetación y revisión final.

Capítulo 3: Preliminares

En este capítulo se explican, de forma detallada, los conceptos con los que se ha trabajado durante el proyecto. Esta información es necesaria para poder comprender el contenido de esta memoria y el propio proyecto.

3.1 Testing

El concepto de *testing* ya ha sido introducido en el capítulo 1 de esta memoria, por lo que en este apartado se limitará a explicar el tipo de *testing* utilizado en este proyecto, el basado en propiedades, y se comparará con el basado en coberturas, otro tipo de *testing* muy extendido.

3.1.1 Testing basado en coberturas

El *testing* basado en coberturas consiste en diseñar pruebas de caja blanca orientadas a abarcar tanto código como sea posible, midiendo la calidad en base a esto.

En cuanto a *testing* basado en coberturas*,* se pueden encontrar los siguientes criterios para medir la calidad [11]:

- Según el número de instrucciones ejecutadas.
- Según el número de ramas elegidas: si en un programa se cuenta con sentencias condicionales, se cuenta el número de caminos tomados en relación al total.
- Según el número de condiciones: a diferencia del anterior, en este se comprueba que en sentencias condicionales complejas se cumplan todas las posibles condiciones de estas y se cuentan las posibilidades y los caminos tomados por cada una.
- Según el camino: al contar un programa con varios caminos a elegir debido a varias sentencias condicionales, se comprueba que se pase por todas las posibles combinaciones de todos los posibles caminos.

Para entender mejor cada criterio supongamos que tenemos el siguiente código, y aplicamos las pruebas anteriores para el caso de prueba en el que *a=1* y *b=-2*. Los resultados serían:

- Según el número de instrucciones. Como se puede ver en la figura 3.1 (izquierda), el código cuenta con 10 instrucciones y, con los valores de *a* y *b* definidos, se ejecutan 4 de esas instrucciones. La medida sería 4/10.

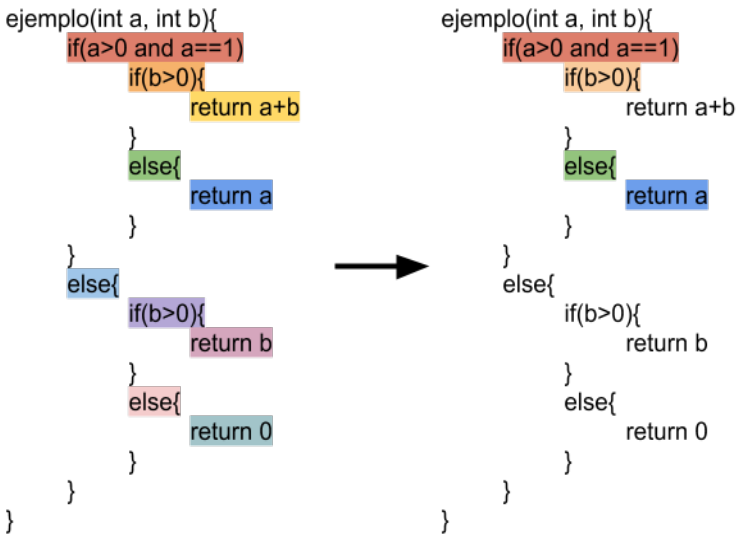

Figura 3.1: Ejemplo de *testing* basado en coberturas según el número de instrucciones.

- Según el número de ramas. El código cuenta con 4 ramas, y para los valores de *a* y *b* escogidos pasa por una de esas ramas. La medida en este caso es 1/4 (figura 3.2).

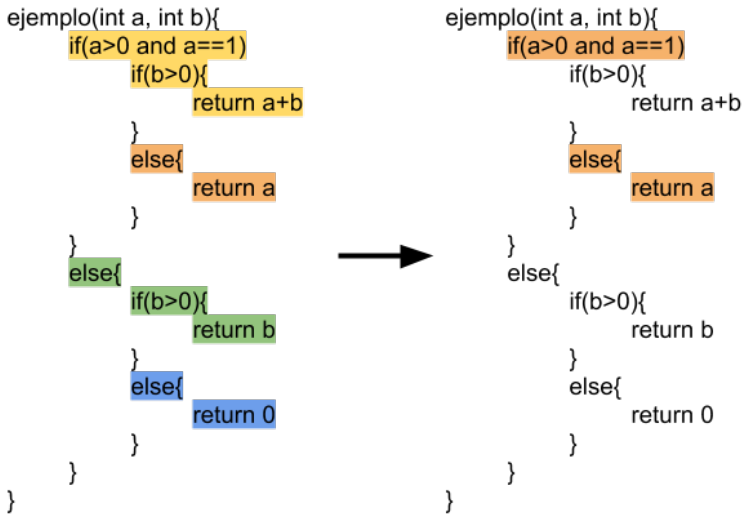

Figura 3.2: Ejemplo de *testing* basado en coberturas según el número de ramas.

- Según el número de condiciones. El código presenta un total de 8 condiciones posibles, ya que para las 4 condiciones que pueden verse cuenta los valores que pueden tomar (**`True`** o **`False`**). Como puede verse en la figura 3.3, para *a=1* y *b=-2*, pasa por tres condiciones, tomando un solo valor cada una. Por eso, la medida resultante es 3/8.

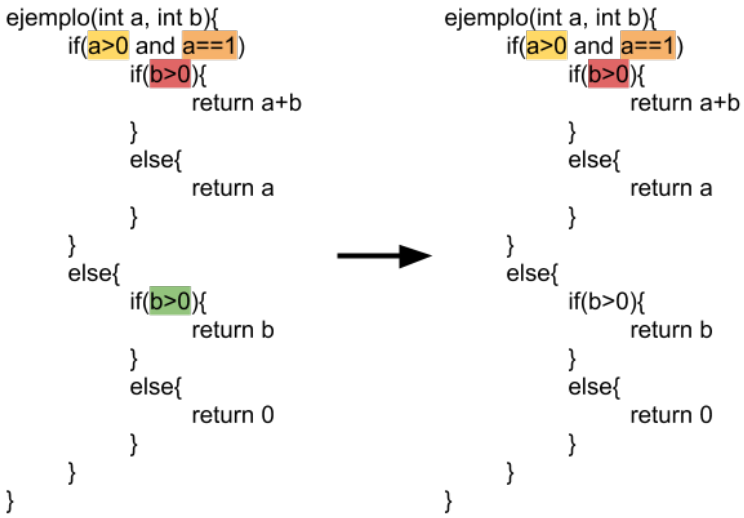

Figura 3.3: Ejemplo de *testing* basado en coberturas según el número de condiciones.

- Según el camino. En este caso, la medición coincide con el número de ramas (figura 3.4). Sería distinto si, por ejemplo, hubiera otro **`if`** al final del código, pues habría que contar las combinaciones de los caminos del primer **`if/else`** con el segundo.

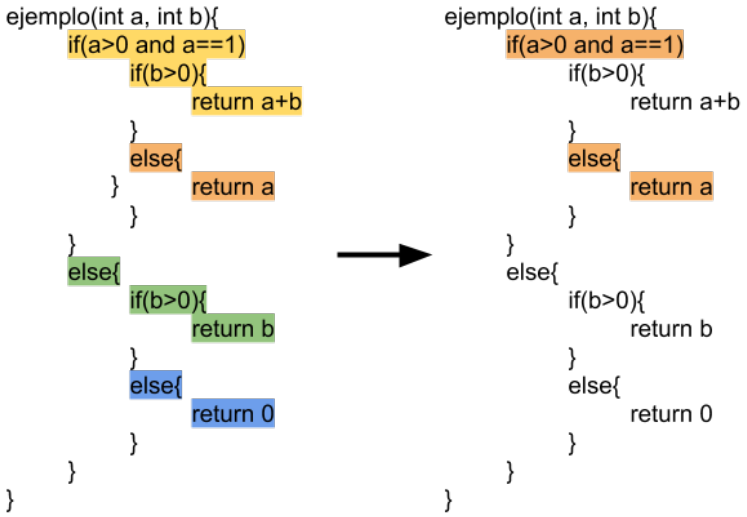

Figura 3.4: Ejemplo de *testing* basado en coberturas según el camino.

3.1.2 Random testing

Por otro lado, el *testing* basado en propiedades consiste en definir propiedades que el programa deba cumplir. Por ejemplo, si se tiene una lista ordenada y una función para insertar elementos en ella, puede definirse una propiedad que indique que la lista debe seguir estando ordenada tras pasar por dicha función.

El *random testing* es una técnica de *testing* basada en propiedades que utiliza generación aleatoria de datos para comprobar que estas propiedades se cumplen. De este modo, los programas se prueban generando entradas aleatorias e independientes entre sí, y los

resultados generados por el programa a partir de esos datos aleatorios se comparan con las especificaciones del programa a fin de comprobar si la salida es correcta o incorrecta. Para medir la calidad se cuentan el número de propiedades que se cumplen y que fallan.

Respecto a las técnicas de cobertura, el *random testing* tiene la ventaja de que escala bien cuando tenemos ramas **`if/for/while`**, ya que simplemente produce datos aleatorios y ejecuta el programa con estos datos, aunque tiene la desventaja de que es necesario dar con las propiedades adecuadas y formarlas correctamente para que comprueben lo que de verdad queremos que comprueben.

3.2 Lógica Lineal Temporal

La Lógica Lineal Temporal (LTL) es una lógica modal que interpreta los operadores modales como temporales. Estos operadores son [12]:

- Next φ: φ debe cumplirse en el siguiente estado, en todos los caminos.
- Always φ: φ debe cumplirse siempre, en todos los caminos.
- Eventually φ: φ se cumple en algún momento, en todos los caminos.
- φ_1 Until φ_2: φ_1 debe cumplirse hasta que se cumple φ_2 en el siguiente estado, y φ_2 debe cumplirse alguna vez, en todos los caminos.
- φ_1 Release φ_2: φ_1 debe cumplirse hasta que se cumplen φ_1 y φ_2. También son válidos los casos en los que solo se cumpla φ_1. Al igual que en los casos anteriores, esto debe suceder en todos los caminos.

LTL necesita estados o tiempo finito, pero los sistemas de *stream processing* están diseñados para un número indefinido de tiempo y estados, ya que los datos que llegan al *stream* están generándose constantemente y no tiene por qué haber modo de saber cuándo o si llegará un último dato. Por este motivo no es posible utilizar esta lógica directamente.

La solución a este problema es utilizar LTL3 [13], una lógica lineal temporal que maneja cierto, falso y un valor indefinido ‘?’. Las normas que sigue son las mismas que la Lógica Lineal Temporal, por lo tanto, ambas devuelven los valores cierto y falso en los mismos casos, pero además, LTL3 puede devolver el valor indefinido cuando no se tienen suficientes datos para saber si una fórmula es cierta o falsa.

Y es a partir de esta lógica, LTL3, donde surge la Lógica Lineal Temporal para Sistemas de *Streaming* (LTLss [6]) que se utiliza en este proyecto. Esta lógica, creada por Adrián Riesco y Juan Rodríguez-Hortalá, surge debido a los problemas encontrados a la hora de utilizar Lógica Lineal Temporal con la técnica de *random testing* aplicada a sistemas de *stream processing*.

Esta lógica extiende la lógica LTL3 añadiendo números naturales a los operadores y permitiendo comportamientos finitos y por tanto, más realistas. De este modo es posible definir propiedades que se cumplan en un intervalo finito de tiempo y evitar así el uso del valor ‘indefinido’, ya que la propiedad sabe que debe cumplirse en un tiempo dado y, si acabado ese tiempo no se ha cumplido la propiedad, puede concluirse que es falsa,

mientras que en la LTL3 la formula quedaría en un estado inconcluso. Además, debido a que en LTLss las propiedades son finitas, es posible expresar cualquier fórmula utilizando el operador *next*, y esto se aprovecha para evaluar dichas fórmulas. Cuando una fórmula está construida de este modo, se dice que está en 'forma *next*'. Por ejemplo, un $always_3\,\alpha$ en 'forma *next*' tendría el siguiente aspecto:

$$\alpha \text{ and } next(\alpha) \text{ and } next(next(\alpha))$$

3.3 Sscheck

En el apartado 1.2 ya se ha presentado la herramienta Sscheck [6]. En este punto se centrará en los conceptos y tecnologías que utiliza Sscheck y que son relevantes para este proyecto.

3.3.1 Apache Spark

El proyecto Sscheck aplica *random testing* basado en Lógica Temporal a Spark, así que se comienza este apartado con una introducción a Spark, en la que se explican sus características principales.

Estas características serán especialmente útiles en los siguientes apartados de este capítulo, cuando se hable de las diferencias entre Spark y Flink a la hora de diseñar los generadores de datos de prueba y la función de test.

Spark [14] es un sistema de computación de código abierto, que permite la posibilidad de ejecutar el cómputo de manera distribuida en distintos *clusters*. Para ello ofrece un *framework* y una API que abstrae al desarrollador de parte de la complejidad de desarrollar este tipo de algoritmos, encargándose Spark de la distribución del cómputo.

Usando estas características, han surgido numerosas bibliotecas adaptadas a Spark para tratamiento de datos en *streaming* [15] de distintas fuentes, que abarcan desde el más bajo nivel, como pueden ser datos recogidos directamente desde un fichero, a fuentes más avanzadas como Twitter o Kafka.

El tratamiento de datos en *streaming* mediante Spark Streaming [16] se realiza mediante *streams* discretizados (**`DStreams`**) que representan el flujo continuo de datos mediante una composición de colecciones inmutables de datos (**`RDD`**) generadas a lo largo del tiempo. Es decir, un **`DStream`** estará compuesto por un **`RDD`** generado entre t=0 y t=1, otro **`RDD`** entre t=1 y t=2, etc.

Esta división dependiente del tiempo permite que la aplicación de fórmulas de Lógica Temporal se adapte de un modo totalmente natural al tratamiento de datos en *streaming* mediante Spark.

3.3.2 Generadores

Aunque ya se dispone de bibliotecas que proporcionan generadores para *random testing*, en el proyecto Sscheck se han utilizado estos generadores para crear unos nuevos que

cubran las siguientes necesidades:

- Los elementos generados deben estar contenidos dentro de un **`PDStream`**. De este modo los datos generados tendrán el mismo formato que los que recibe Spark desde otras fuentes de *streaming*.
- Se requiere de datos que cumplan con las condiciones establecidas por los operadores de LTLss.

Estos nuevos generadores son:

- $\underline{Always}_n$: todos los datos que genera cumplen una misma propiedad durante *n* instantes.
- $\underline{Eventually}_n$: dentro de *n* instantes, crea un dato que cumple una propiedad.
- $\underline{Until}_n$: genera elementos que cumplen una primera propiedad hasta generar un elemento que cumple una segunda propiedad antes de superar *n* instantes.
- $\underline{Release}_n$: produce elementos que cumplen una primera propiedad hasta producir un elemento que cumple tanto la primera como una segunda propiedad antes de superar *n* instantes. También puede ocurrir que todos los elementos siempre cumplan la primera propiedad y nunca la segunda.

3.3.3 Test

Para la aplicación de *random testing* a Spark, la herramienta Sscheck dispone de una función **`test`** que, dada una fórmula de LTLss, un generador de **`PDStream`**, y un número de ejecuciones, evalúa la fórmula con los datos generados por el generador tantas veces como indique el número de ejecuciones.

3.3.4 Formula

Sscheck necesita un modo de representar las fórmulas de lógica lineal temporal. Aprovecha características diferenciales de Scala respecto a otros lenguajes usados en entornos de computación distribuida como Java y Python, tales como su manejo exclusivamente de objetos, el hecho de que fue creado como un lenguaje funcional y dispone de funciones de primer orden y demás características funcionales, la definición de funciones implícitas, y disponer de objetos inmutables, perfectos en el procesado concurrente para evitar que varios procesos intenten modificar el mismo objeto y se produzcan problemas de concurrencia. También hace uso de la biblioteca Scalaz [17] para facilitar la creación de fórmulas como si estas formaran parte del propio lenguaje Scala. Estas fórmulas siguen las normas establecidas por la Lógica Lineal Temporal ya explicadas en el apartado 3.2.

3.4 Apache Flink

Flink, al igual que Spark, es un entorno de computación distribuida diseñado para el tratamiento de datos en *streaming,* y ofrece un *framework* y una API para hacer más sencillo el tratamiento de los datos [18].

En el caso de Flink, este no solo ofrece la API para enmascarar la complejidad de la paralelización, sino que busca ofrecer un paradigma distinto a la hora de tratar los datos. Aunque Flink cuenta con un modo parecido a Spark de representar los datos, un **`DataStream`** (similar a una lista con los datos que se van generando), Flink es mucho más restrictivo con respecto a las operaciones que se pueden realizar sobre este flujo.

El paradigma que ofrece Flink es el de separar el cómputo de este *stream* en partes más pequeñas. Para esto se permite separar el *stream* original en varios *streams* según una clave, y separar el *stream* original o estos nuevos *streams* en distintos subconjuntos de datos generados según una cierta condición; estos subconjuntos se denominan ventanas.

Existen múltiples formas de agrupar los datos en ventanas; según el intervalo de tiempo en el que se ha generado cada dato, según el orden en el que los datos llegan al sistema, o incluso según las sesiones de actividad de los usuarios [19].

En las ventanas es donde se puede realizar el grueso del cómputo. Se permiten operaciones de transformación de los datos, agregación y reducción, mediante las típicas funciones como **`map`** o **`aggregate`** y el uso de funciones anónimas, o bien usando herencia a clases específicas de su API para que una implementación compleja de estas funciones resulte más sencilla.

Por tanto, el esquema que sigue Flink para el tratamiento del *stream* de datos es cíclico, ya que al aplicar una función sobre las ventanas se devuelve otra vez un *stream* de datos, el cual puede volverse a dividir según una clave y volverse a separar en ventanas.

Sin embargo, esta división del *stream* de datos en ventanas es solo lógica, no física, y en realidad los datos se guardan de una manera distinta. Esto tiene implicaciones importantes en el desarrollo del proyecto, y por eso a continuación se explica más en profundidad los distintos datos y operaciones que se pueden realizar en Flink.

3.4.1 Streams

El **`DataStream`** es la representación básica e inicial del *stream* de datos en Flink. Un **`DataStream`** se genera a través de distintas fuentes y permite aplicar varias transformaciones a sus datos, que podrán generar otro **`DataStream`** u otros tipos de *streams*. Finalmente Flink permite mandar el *stream* resultante a una variedad de diferentes destinos.

Flink permite crear **`DataStreams`** a partir de distintos tipos de orígenes, denominados **`DataSource`** [20]:

- Ficheros (de texto línea a línea, o de ficheros sin especificar tipo).
- *Sockets* para la comunicación entre procesos.
- A través de colecciones. Flink cuenta con distintos métodos para crear **`DataStreams`** a través de secuencias, listas de elementos o elementos que sean iterables. Esta funcionalidad es la que más se usará en este proyecto.

- Origen personalizable: es posible crear un **`DataSource`** propio mediante el uso de conectores. Existen también varios conectores ya incluidos que permiten usar, entre otros, Twitter o Kafka como generadores de **`DataStreams`**.

Las transformaciones que se le pueden aplicar a los **`DataStreams`** pueden diferenciarse por el tipo de *stream* resultante. En esta memoria solo hablaremos de cuatro [21]:

- División del **`DataStream`** en varios *streams* según un criterio configurable. Se obtiene un **`SplitStream`** que representa el **`DataStream`** dividido, estos **`SplitStream`** aceptan operaciones para poder seleccionar el *stream* deseado tras la separación (figura 3.5).

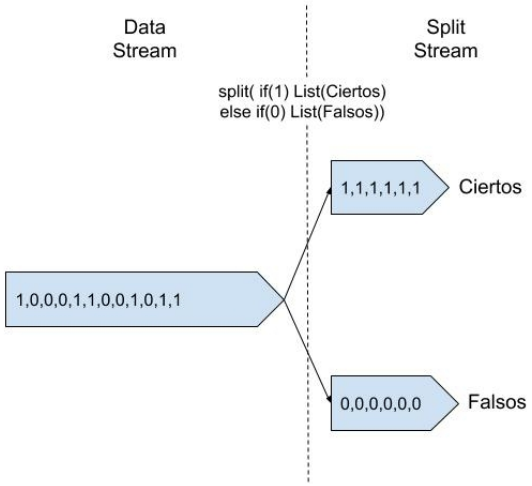

Figura 3.5: Ejemplo de **`SplitStream`**

- Conectar varios **`DataStream`** con distintos tipos en uno solo, denominado **`ConnectedStream`** (figura 3.6). Esto permite que se apliquen operaciones que guarden un estado en común a ambos **`DataStream`**.

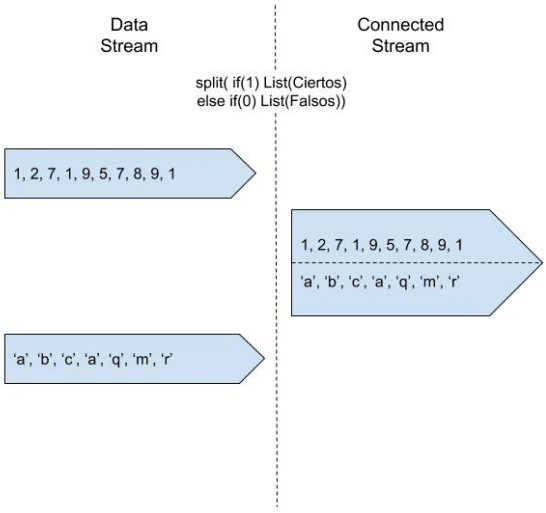

Figura 3.6: ejemplo de **`ConnectedStream`**

- Agrupar el *stream* según una clave presente en los datos del *stream*. Se obtiene un **`KeyedStream`**. Nótese que en la implementación de Flink se sigue usando un único *stream* para representar un *stream* al que se le ha aplicado esta agrupación, es decir, la diferencia en la implementación de Flink no está tanto en el modo en el que se guardan los datos como en el modo en el que luego se aplican las funciones de transformación en el **`KeyedStream`**. Estos **`KeyedStream`** pueden transformarse otra vez en **`DataStream`*`s`*** o en **`WindowedStreams`**.
- Dividir el *stream* en una o varias ventanas. Como ya se ha explicado antes esto divide de manera lógica el *stream* en varios grupos de datos que se tratan de manera independiente. Al aplicar transformaciones sobre estas ventanas se devuelve un **`DataStream`**. Al igual que ocurre con los **`KeyedStreams`**, la representación de un **`DataStream`** dividido en ventanas es un único **`WindowedStream`**. Las implicaciones que esto acarrea se tratan en el siguiente subapartado.

En la figura 3.7 se puede ver un ejemplo de aplicar transformaciones entre distintos *streams* para ilustrar mejor la explicación anterior. En este ejemplo se comienza con un **`DataStream`** de parejas que se separan en un **`KeyedStream`** según su segundo elemento. Tras esto, se separan en ventanas de dos elementos cada una, y finalmente se agregan los elementos de las ventanas sumando el primer elemento de la pareja.

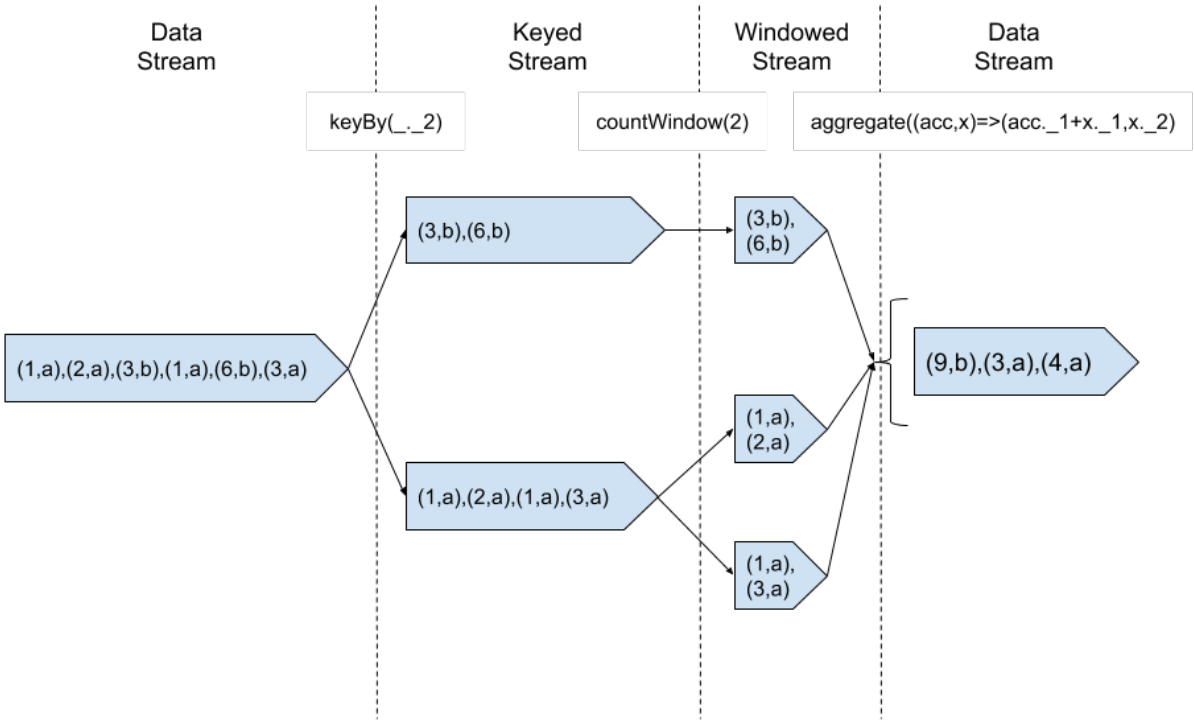

Figura 3.7: ejemplo de transformaciones en **`DataStream`**

Todas estas transformaciones serían inútiles si no se pudiera guardar su resultado en algún lugar. Flink representa los destinos como **`Sinks`** [20], y al igual que ocurre con los **`DataSources`**, hay una serie de destinos ya incluidos en Flink por su común uso, como volcar los datos en ficheros de texto o CSV, mostrarlos por pantalla, o canalizarlos a un

socket para que otra aplicación los reciba. También se puede implementar un s*ink* propio a través de conectores como pasaba con los **`DataSource`**, pudiendo enviar los datos a Kafka, a un sistema de ficheros Hadoop o a Elasticsearch; nótese que los conectores son distintos según son generadores o receptores, por ejemplo Twitter no puede ser usado como s*ink* y Elasticsearch no puede ser un **`DataSource`**.

3.4.2 Windows y Triggers

En el apartado anterior ya se ha explicado el funcionamiento lógico de las ventanas y también se ha introducido la idea de que su implementación no es una traducción directa de la descripción de alto nivel de las ventanas que se ha hecho.

Este apartado tiene como objetivo explicar más en detalle el funcionamiento de las ventanas, los distintos tipos de ventanas que hay, las funciones que se les pueden aplicar e introducir el concepto de *trigger* en Flink [19].

Las ventanas en Flink se pueden separar en varios tipos según distintos parámetros:

- Según si son ventanas estáticas (*tumbling windows)* o se desplazan (*sliding windows)*. Es decir, si hay varias ventanas cada una con sus elementos, o hay una ventana que se va desplazando a lo largo del *stream* según la opción indicada. En este grupo también se pueden incluir las ventanas de sesión (*session windows*), que no pertenecen a los grupos anteriores, sino que tratan los elementos por sesiones de actividad; no se va a dedicar más tiempo a este tipo de ventanas por que no han sido usadas en el proyecto.
- Según el parámetro que decide cómo separar el *stream* en ventanas:
    - *Time windows*: los datos se agrupan según el tiempo en el que han sido generados. Por ejemplo, una **`TimeWindow`**(3) contiene los elementos generados en intervalos de tres segundos.
    - *Count windows*: los datos se agrupan cuando llega un número determinado de elementos. Por ejemplo, una **`CountWindow`**(3) contiene los tres primeros elementos que llegan en orden, y se generará otra cuando lleguen los tres elementos siguientes, etc.

Estos dos parámetros por los que hemos separado las ventanas en tipos son compatibles entre sí, es decir, se pueden tener *time windows* y *count windows* que se desplacen. En la figura 3.8 se puede ver un ejemplo de ventana por contador de tres elementos con un desplazamiento de un elemento a las que se le aplica la función de concatenar sus elementos. Y en la figura 3.9 se puede ver la división de un **`DataStream`** en ventanas estáticas de cinco segundos.

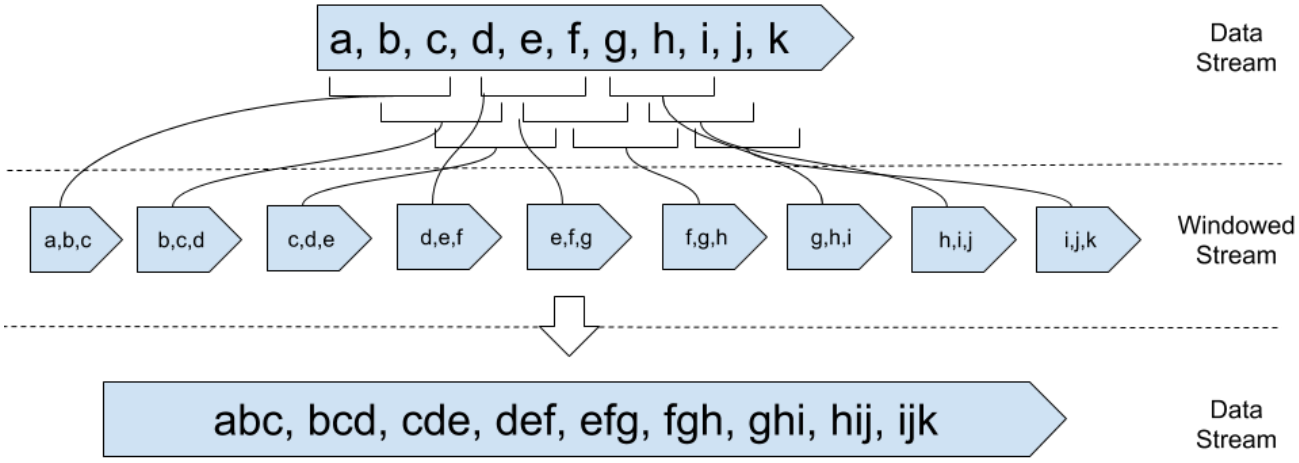

Figura 3.8: Ejemplo de **`CountWindows`**(3) con un desplazamiento de un elemento

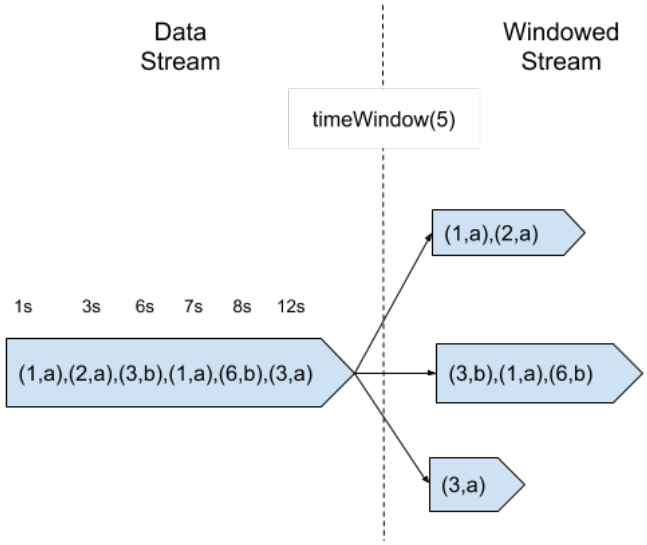

Figura 3.9: Ejemplo de **`TimeWindow`**(5) sin desplazamiento

Existen también las ventanas globales (**`GlobalWindow`**), que introducen todos los elementos del *stream* en la misma ventana. En principio se podría pensar que estas ventanas no tienen utilidad práctica, ya que no ejecutarían las funciones de transformación nunca a no ser que el *stream* fuera finito, y además generarían un **`DataStream`** de un solo elemento; sin embargo, esto no es así. Las **`GlobalWindow`** deben recibir un disparador que indica tras qué eventos se van a ejecutar las funciones de transformación aplicadas a la **`GlobalWindow`**. Estas funciones de transformación devolverán un único elemento por cada ejecución de la función que se introducirá en el **`DataStream`** resultante de la transformación. El **`DataStream`** final tendrá como máximo tantos elementos como veces salte el disparador.

Los *triggers* (disparadores) son objetos que indican el funcionamiento que debe tener una **`GlobalWindow`** al producirse ciertos eventos. El disparador debe implementar métodos que indiquen qué hacer cuando la **`GlobalWindow`** recibe un elemento, cuándo salta el temporizador de eventos y cuándo salta el temporizador de procesado. También se deben

implementar métodos que indiquen el comportamiento al unir ventanas y al eliminar la ventana. En estos métodos se debe indicar mediante un *flag* cuando el disparador debe solo saltar, saltar y eliminar lo que tiene almacenado o continuar.

Se puede observar que para cumplir con su función, los disparadores deben poder almacenar datos comunes a toda la **`GlobalWindow`**, digamos guardar información de estado. Esto no se puede hacer definiendo una variable dentro de la clase, ya que al realizar la computación de manera distribuida podría haber problemas. Para evitar esto Flink ofrece una serie de descriptores de estado de distinto tipo en los que guardar la información que se quiere indicando el nombre de la variable, su valor y su tipo.

Ahora que se ha introducido lo que es un disparador y una **`GlobalWindow`**, se puede entender el verdadero funcionamiento de las ventanas en Flink. Aunque de manera lógica y de cara al usuario parezca que el **`DataStream`** se divide en una colección de ventanas en la que cada una contiene sus respectivos datos, lo que en realidad hay es un **`WindowedStream`**, un único *stream* que contiene todos los datos, una **`GlobalWindow`** y, según el tipo de ventana que se le ha aplicado, cuenta con un disparador u otro adaptado. Si es una *count window*, el disparador cuenta el número de elementos que entran y se dispara cuando el número de elementos que contiene coincide con el parámetro indicado en el constructor de la ventana; de una manera equivalente funcionan las *time windows* y todas las demás ventanas.

A las ventanas también se les puede indicar un **`Evictor`**, un objeto en el que se indica los elementos de la ventana que se eliminan tras ejecutarse el disparador y antes de ejecutar las funciones. No se dedicará más tiempo en la memoria a los **`Evictors`** ya que no se han utilizado en este proyecto.

Como conclusión a este apartado de Flink, es importante destacar que el modelo de **`DataStream`** y ventanas que sigue Flink no es comparable al de **`PDStream`** y **`RDDs`** que sigue Spark, ya que en Spark sí que realmente un *stream* es una colección de **`RDDs`**, y ambos son iterables y se pueden manejar de manera similar. En Flink en el fondo solo disponemos de un *stream*, al que se le puede aplicar funciones para transformarlo en otros tipos de *stream*, que cuentan con distintas funcionalidades y un funcionamiento distinto de cuándo, cómo y a qué elementos se aplican; no existe en realidad una separación del *stream* en colecciones más pequeñas.

3.5 Scala

Scala es un lenguaje de alto nivel que combina características de programación funcional y programación orientada a objetos, aprovechando los puntos fuertes de ambos para formar un lenguaje altamente escalable que se adapta a las necesidades de sus usuarios [22]. Además corre en la máquina virtual de Java y es compatible con aplicaciones Java.

Flink está escrito en Scala, motivo por el cual este proyecto ha sido implementado con este lenguaje.

3.6 ScalaCheck

ScalaCheck [23] es una biblioteca de Scala orientada al *testing* automatizado y basado en propiedades de programas Java y Scala. Proporciona funcionalidad para generar colecciones de datos de manera determinista o aleatoria, y permite especificar distintos parámetros para acotar la generación de estos datos.

ScalaCheck nos proporciona también un tipo enumerado ideal para la representación de los estados de las fórmulas temporales. Esto es si la fórmula es cierta, falsa, o todavía no se ha decidido su resultado.

ScalaCheck se usa en el proyecto junto con Specs2 y JUnit para la creación de clases específicamente de *testing*. Specs2 [24] es un *framework* orientado a *testing* que especifica una serie de métodos que ejecutan las pruebas, y el orden y modo de ejecución de estas.

Estas bibliotecas se usan tanto en el proyecto Sscheck, como en este proyecto. Las bibliotecas se usan en ambos proyectos para:

- La implementación de los generadores de datos orientados a fórmulas de LTLss.
- Guardar el estado de decisión de las fórmulas.
- La implementación y ejecución de las clases de *testing*.

Capítulo 4: Implementación

Este capítulo contiene toda la información sobre la implementación del proyecto, incluyendo su estructura y funcionamiento.

4.1 Descripción general

En este proyecto se encuentran cuatro partes diferenciadas, tal y como hemos visto en los objetivos. Estas son la toma de contacto con Flink, los generadores, la función **`test`**, y las pruebas de concepto.

Podemos ver estas cuatro partes reflejadas en la distribución del código del proyecto, el cual se divide en los siguientes paquetes:

- Examples: Aquí se puede encontrar el paquete que contiene la primera aplicación desarrollada para tomar contacto con el funcionamiento de Flink.
- Gen: En este paquete se encuentra el generador de ventanas y todo lo necesario para la implementación de este generador.
- Test: Contiene todo lo necesario para poder realizar el *testing*, principalmente la función **`test`** y la clase **`Formula`** de Sscheck, además de las clases necesarias para su funcionamiento y una clase con varias pruebas sencillas para probar la función **`test`**. Además, contiene otros tres paquetes con implementaciones que comenzaron siendo parte de soluciones de la aplicación principal y, debido al potencial que tienen por sí solos, se continuó con su desarrollo.
  - FormulaTrigger: Contiene la clase que implementa un disparador que se ejecuta según el resultado obtenido de aplicar una formula a los datos recibidos. También contiene un test con varias pruebas.
  - KeyedStreamTest: Contiene la clase que implementa un modo de aplicar una fórmula a un **`KeyedStream`**, es decir, realiza *testing* sobre un **`KeyedStream`**. También contiene un test con varias pruebas.
  - StreamTest: El funcionamiento de las clases es similar al anterior, solo que el *testing* se realiza sobre un **`DataStream`**.
- Demo: Finalmente, este paquete contiene los dos casos de prueba que muestran el funcionamiento la función **`test`** final. Cada prueba está compuesta por una clase con los generadores de datos y funciones necesarias para las condiciones de las fórmulas, y otro con diferentes pruebas que ejecutan la función **`test`**, por lo que cada prueba se encuentra en su propio paquete dentro de este.

4.2 Toma de contacto con Flink: detector de polución

En los siguientes puntos se desarrolla en profundidad la aplicación desarrollada como toma de contacto con Flink, explicando el problema planteado, la generación de los datos

y los pasos obtenidos para su tratamiento y la obtención de los resultados deseados.

4.2.1 Descripción del problema

Como toma de contacto con Flink, con el objetivo de aprender el funcionamiento del flujo de un programa Flink, y también con el objetivo de tener un problema al que luego aplicarle *random testing* basado en LTLss, se propuso el siguiente problema al que buscarle solución.

Disponemos de varios sensores de polución repartidos por la ciudad, y estos sensores devuelven el nivel de polución en forma de números enteros cada cierto tiempo, acompañado del identificador del sensor que lo detecta, también como un entero.

Por otro lado, cada sensor envía los datos a una alarma encargada de activarse o desactivarse dependiendo de si estos datos indican si hay polución. Para saber si un dato indica polución, se dispone de una variable que establece un límite que, de alcanzarse o superarse, significa que el valor es de polución alta. Sin embargo, que un solo valor lo supere no es suficiente.

Para decidir cuándo se activa la alarma, la información enviada por los sensores se divide en *time windows* donde entran todos los datos detectados en un intervalo de cinco segundos. Hecha esta división, se comprueba en cuántos intervalos seguidos se ha detectado algún valor superior al límite establecido, y si esto ocurre durante más de un número fijado de intervalos, la alarma se activa, informando del sensor al que está conectada. En caso de que la alarma ya estuviera activada, no se modifica su estado.

Por el contrario, una alarma se desactiva cuando no se encuentra ningún dato que supere el límite durante *n* intervalos consecutivos de cinco segundos. Es decir, se sigue el mismo proceso anterior pero, en lugar de buscar algún dato que supere el límite, se busca que no exista ninguno que lo supere. Igual que en el caso anterior, la alarma informa del sensor para el cual se ha desactivado y, en caso de que ya estuviera desactivada, no se modifica su estado.

4.2.2 Generadores de datos

La primera parte del desarrollo de la solución es una simulación de los sensores. Para esta parte no se ha utilizado un generador de valores fijos, sino que se han propuesto una solución que los genere de manera semi-aleatoria pero garantizando ciertas condiciones.

Para buscar que el programa de verdad funcione y la comprobación de esto sea sencillo buscamos el siguiente comportamiento:

1. Los 100 primeros datos que detecta cada sensor serán números aleatorios entre 0 y una variable **`maxPollution`** que indica el límite a partir del cual se considera que hay polución.
2. Los 100 siguientes datos serán inferiores a **`maxPollution`** para garantizar que las alarmas se desactiven.

3. Los siguientes datos volverán a superar **`maxPollution`** y las alarmas volverán a activarse, quedándose activadas hasta que el programa termine.

Esta implementación se hizo utilizando hilos y concurrencia, siendo cada hilo un sensor. De este modo, simula el comportamiento que se obtendría al tener distintos sensores distribuidos en diferentes zonas. Cada sensor tiene un identificador y un número de datos a ‘detectar’ (en la práctica, es cada sensor el encargado de producir los datos que detecta). Para simplificar, la aplicación crea tres sensores y los tres generan el mismo número de datos siguiendo el comportamiento explicado anteriormente.

La primera prueba se hizo guardando estos datos en una lista y usando la lista como origen del **`DataStream`**. Sin embargo, al obtener los datos de una lista, los tiempos en los que se generaban eran muy cercanos entre ellos y no se conseguía ver que la alarma se desactivase, ya que la mayoría de los datos entraban en la primera ventana de cinco segundos y no se distinguían los tramos en los que se generaba polución de los que no y siempre se tenía polución en las ventanas.

Como solución a esto, y también para dar una mayor complejidad y realismo a la solución, se decidió desarrollar un programa que generara y enviara los datos a *sockets* de Flink uno a uno, esperando medio segundo entre el envío de cada dato. De este modo los datos se distribuyen de un modo más lógico entre ventanas, consiguiendo ventanas que no tienen nada de polución y ventanas que sí la tienen y, en consecuencia, logrando que la alarma se comporte de acuerdo a los datos recibidos.

Aunque el proyecto está desarrollado casi plenamente en Scala, la primera versión de este generador de datos de polución se desarrolló en Python, debido a que en ese momento del desarrollo facilitaba un prototipado más rápido. Más adelante, para mantener la coherencia y uniformidad del proyecto y facilitar su mantenimiento, se decidió migrar el programa que envía y genera datos escrito en Python a Scala.

4.2.3 Controlador de la alarma mediante Flink

La siguiente parte consiste en el tratamiento de los datos generados para activar o desactivar alarmas de alta contaminación en los distintos sensores. Toda esta parte se ha realizado usando el tratamiento de datos en *streaming* de Flink.

El programa sigue el siguiente flujo:

1. Creación del *stream* mediante los datos recibidos en el *socket* enviados desde la alarma.
2. El *stream* de datos se transforma para que cada elemento sea un par de la forma: (*polucion*, *id_sensor*), con *polucion* e *id_sensor* números naturales que representan el nivel de polución detectado y el identificador del sensor que lo detecta, respectivamente.
3. Se convierte el *stream* a un **`KeyedStream`** para que se traten los datos de cada sensor por separado.

4. Se agrupan los datos en ventanas con un intervalo de cinco segundos.
5. Por cada ventana, el *stream* se transforma de tal manera que si hay un elemento superior al límite establecido se devuelva un 1 y en caso contrario un 0.
6. El nuevo *stream* vuelve a dividirse en ventanas que se desplazan con un tamaño de 3 elementos, y un desplazamiento de 1 elemento. De este modo, será posible ver cuantos unos y ceros seguidos se han obtenido, es decir, durante cuantas ventanas (intervalos de cinco segundos) seguidas se ha tenido o no polución.
7. Contamos el número de elementos en cada ventana que son unos o ceros, respectivamente. Si una ventana tiene tres unos, significa que ha habido polución durante tres intervalos seguidos y por tanto la alarma debe activase. Si, por el contrario, tiene tres ceros, es porque no se ha detectado polución durante tres intervalos seguidos y la alarma debe desactivarse. Y en caso de que la ventana contenga unos y ceros, la alarma mantendrá su estado ya que significa que el nivel de polución fluctúa con demasiada frecuencia para establecer si hay o no polución. Con esta condición se crea un nuevo *stream* en el que cada elemento podrá tomar tres valores: un 2 cuando haya que activar la alarma, un 0 para desactivarla, y un 1 si no hay motivos para cambiar su estado.
8. Finalmente, este s*tream* se transforma en mensajes de activación o desactivación de la alarma manteniendo un estado que guarde el estado actual de la alarma para cada sensor. Si se muestra este *stream* por pantalla o a un archivo se muestran estos mensajes, que van indicando el estado de la alarma (activada, desactivada o sin cambios) e identificador del sensor para el que se indica ese estado. La alarma se encuentra inicialmente desactivada.

Este pequeño proyecto de detección de alarmas ha servido como toma de contacto con muchas funcionalidades básicas de Flink:

- Crear un *stream* a partir de colecciones y de datos que llegan a un *socket*.
- Operar con **`DataStreams`** y **`KeyedStreams`**.
- Utilizar distintos tipos de ventanas: *time windows* estáticas y *count windows* con desplazamiento.
- Aplicación de funciones personalizadas a las ventanas usando el modelo de Flink, mediante definición de clases que hereden de **`AggregateFunction`** (se usa para contar los elementos que superan el límite de polución en cada intervalo de cinco segundos y para contar las ventanas seguidas que tienen polución).
- Uso de funciones que guarden un estado (se usa para mantener el estado de la alarma y saber si es necesario cambiarlo o no).

A continuación se muestra un ejemplo para un límite de polución de 20 (figura 4.1)

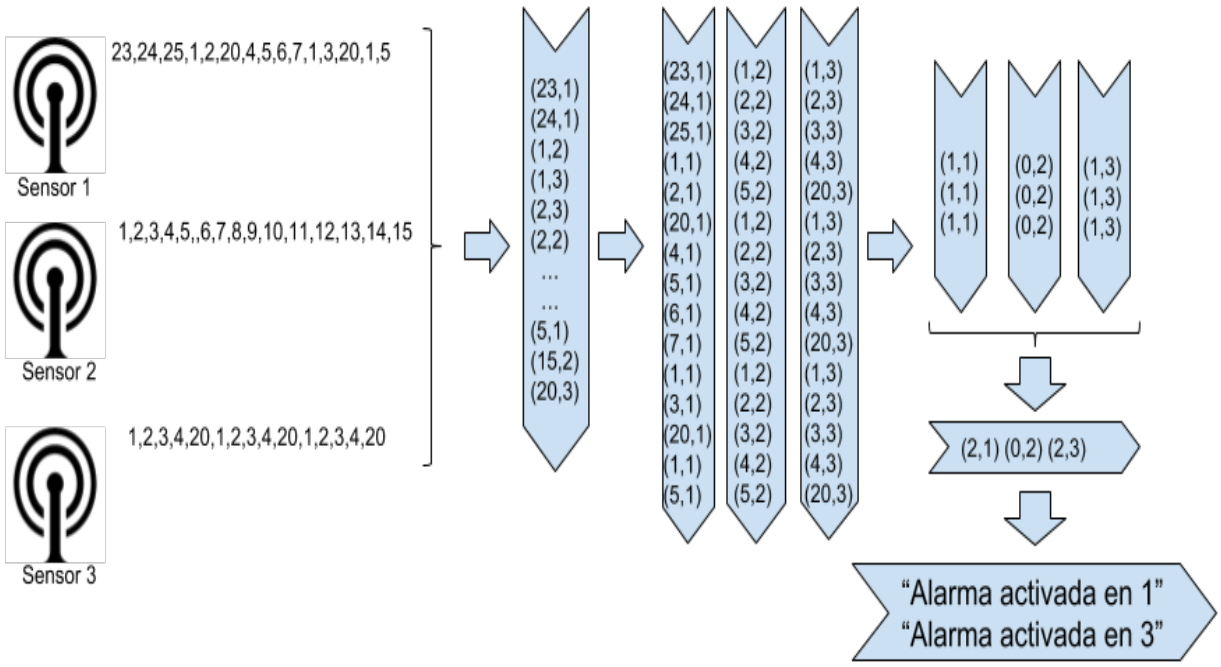

Figura 4.1: Ejemplo de detección de alarmas

4.3 Generadores

Este capítulo trata sobre todos los generadores de datos aleatorios implementados para hacer posible el *testing*, incluyendo los generadores auxiliares, los principales que cumplen las especificaciones de LTLss, y el generador final de ventanas.

4.3.1 Generadores (WinGen) y ListStream

Puesto que la lógica LTLss maneja instantes de tiempo, es necesario establecer qué es un instante en nuestro marco. En el caso de Sscheck, cada lote representa un instante, y para este proyecto se decidió que fueran ventanas. Por este motivo, la idea inicial era implementar generadores de ventanas de Flink cuyo contenido fuera generado aleatoriamente, al igual que en Sscheck se generan lotes y **`PDStreams`** para Spark .

Pero como ya se ha explicado anteriormente, aunque en teoría tanto Flink como Spark realizan una división del *stream* (Spark en lotes y Flink en ventanas), en la práctica son modelos muy distintos. Esto se debe a que, mientras que el *stream* de Spark sí que está formado por una serie de lotes, en Flink no existen las ventanas como tal, y la división del stream es solo lógica y su manejo bastante complejo.

Tras ver que no es posible crear ventanas directamente y pensar posibles soluciones, se optó por implementar generadores de colecciones y posteriormente pasarlas a ventanas como se hizo inicialmente en la aplicación de toma de contacto con Flink, mostrada en la sección 4.2.

Más concretamente, estos generadores devuelven **`ListStreams`** (listas de listas). De este modo, al pasarlo a un **`DataStream`** de Flink se obtiene un *stream* de listas que puede dividirse de forma que cada lista quede en una ventana. Por tanto, en el momento de producir los datos, cada lista es un instante. Esto se hizo así para dar opción a que cada ventana tenga un número diferente de datos, ya que las listas de cada ventana pueden tener longitudes diferentes.

El procedimiento del paso a ventanas se explicará en el siguiente apartado, pero era necesario hablar un poco sobre él para entender el diseño de los generadores.

Estos generadores aleatorios pueden encontrarse en la clase **`WinGen`** y, en su mayoría, adaptan los generadores de lotes de Sscheck para que los datos generados estén en forma de **`ListStream`**. Estos generadores son:

- Generadores auxiliares. Estos son de utilidad tanto para implementar los generadores que satisfagan los operadores de la LTLss, como para facilitar la implementación de generadores propios para realizar *testing*.
  - **`Of`**: dado un generador de tipo **`T`**, devuelve un generador de listas con elementos de este tipo. La longitud de las listas es aleatoria entre 0 y 100.
  - **`OfN`**: hace lo mismo que el anterior, pero las listas generadas tendrán tamaño *N*.
  - **`OfNtoM`**: de nuevo, hace lo mismo que los anteriores, pero en este caso las listas generadas tendrán tamaños comprendidos entre *N* y *M*.
  - **`OfNList`**: convierte un generador de listas en un generador de **`ListStream`**. Esto es útil porque, para poder pasar los datos a ventanas, necesitamos que sea en **`ListStreams`**, y este generador permite que esta acción sea transparente para el usuario.
  - **`LaterN`**: dado un generador de listas, devuelve un **`ListStream`** formado por *N* listas vacías y al final una lista obtenida del generador dado.
  - **`ConcatToListStream`**: concatena dos generadores de listas en uno y lo devuelve como un **`ListStream`**. El generador resultante contendrá las listas generadas por el segundo generador, seguidas de las listas generadas por el primero. Esto es de utilidad a la hora de hacer pruebas. Por ejemplo, si se tiene un generador de datos de polución alta y otro de datos de polución baja, podemos utilizar uno de los tres primeros generadores para obtener listas con y sin datos de polución, y concatenar estos nuevos generadores de tal forma que obtengamos un generador que primero produce datos de polución alta y a continuación datos de polución baja, logrando así algo similar a la primera aplicación de toma de contacto con Flink.
  - **`Concat`**: su función es similar al anterior. Dados dos generadores de listas, los concatena en una sola lista, en lugar de concatenar las listas que generan. En este caso, el resultado será una lista con los elementos de las listas generadas por el segundo generador seguidos de los elementos de las listas generadas por el primero.

- **`Union`**: dados dos generadores de **`ListStream`**, devuelve un **`ListStream`** en el que cada elemento está formado por un elemento del primer generador y otro elemento del segundo.

- Generadores de datos. Generan datos que siguen las especificaciones de LTLss:
  - **`Next`** **α**: dado un generador α, crea una lista vacía seguida de una lista con los datos generados por α, de tal forma que la lista vacía sería el instante actual, y la lista con los valores aleatorios el instante siguiente.
  - **`Always`**$_n$ **α**: genera *n* datos con α.
  - **`Eventually`**$_n$ **α**: genera un máximo de *n-1* listas vacías seguidas de una lista generada por α.
  - **α** **`Until`**$_n$ **β**: genera datos con α un máximo de *n-1* veces y termina con un dato generado por β. Por tanto, genera un máximo de *n* datos. También puede ocurrir que genere un solo dato de β.
  - **α** **`Release`**$_n$ **β**: genera datos con β un máximo de *n-1* veces y termina con un dato generado por la unión de α y β. Genera un máximo de *n* datos, y puede ocurrir que solo genere datos de β.

4.3.2 Paso a ventanas

Igual que en Sscheck se tiene un **`PDStream`** formado por lotes, en este proyecto se tiene un **`ListStream`** formado por listas, aunque posteriormente este **`ListStream`** se transforma en ventanas, que son lo que Flink realmente utiliza.

El procedimiento seguido es:

1. Ejecuta el generador para obtener la lista con los datos divididos en listas.
2. Se pasa esta lista a Flink, el cual la transforma en un *stream* de listas. Al pasar la lista a Flink, se añade un **`List(null)`** al final para saber cuando termina el *stream*. El motivo por el que esto es necesario se explica en el siguiente apartado.
3. El *stream* se ‘divide’ en *count windows* de un solo elemento, quedando cada lista en una ventana.
4. Se devuelve el *stream* final dividido en ventanas.

Tras realizar esta implementación y comprobar su funcionamiento, se intentó encontrar un modo de introducir los datos en ventanas sin necesidad de agruparlos previamente en listas, con el fin de que el *stream* de Flink y su tratamiento fuera más natural.

Para esto, se pensó en implementar un disparador propio que funcionara como el disparador de una *count window*, pero que permitiera tener ventanas de diferentes tamaños. Sin embargo, al poco de comenzar la implementación se vio que esto no es posible, por lo que la solución se descartó y se continuó con el procedimiento explicado al principio de este mismo apartado.

El problema es que no se encontró el modo de que el disparador se ejecutara para un

número diferente de datos cada vez. Sería necesario que el usuario incluyera una lista con los diferentes tamaños de las ventanas deseadas, lo cual no es viable para *streams* de gran tamaño. Además, al utilizar generadores aleatorios, en muchos casos se desconoce el número de datos generados en cada instante de tiempo, lo que anula la posibilidad de indicarle al disparador el número de datos que debe ir en cada ventana.

4.4 Test

La función **`test`** es análoga a la definida en el proyecto Sscheck, con la diferencia de que los tipos que maneja son los utilizados por Flink y los generadores explicados en el apartado anterior.

Como ya se ha explicado, no ha sido posible implementar generadores de ventanas. Por este motivo, la función **`test`** recibe un generador de **`ListStream`** que posteriormente utiliza para crear el *stream* de ventanas al que aplicarle la fórmula, la cual también recibe como parámetro junto con el número de tests a realizar.

Para la creación y evaluación de las fórmulas de LTLss se reutiliza la clase ***`Formula`*** de Sscheck. Para el desarrollo de la función **`test`**, se realizaron varias iteraciones en las que se fue añadiendo complejidad y funcionalidad hasta obtener la función final. En los siguientes apartados se explican cada una de las versiones en orden cronológico de desarrollo.

4.4.1 Primera iteración

La funcionalidad básica que se pretendió conseguir en esta primera iteración era evaluar la fórmula en las ventanas generadas por el generador.

La primera solución pasaba por usar una función **`aggregate`** propia que hereda de la clase **`AggregateFunction`** de Flink. Las funciones **`aggregate`** reciben los datos de una ventana y los utilizan para obtener un resultado que devuelven en un nuevo *stream,* teniendo así un nuevo dato por ventana.

La función **`aggregate`** desarrollada, llamada ***`ForEachWindow`***, tiene el siguiente comportamiento:

1. Recibe la fórmula a evaluar en ‘forma *next*’.
2. Para cada elemento de la ventana (en este caso, la lista análoga al **`RDD`** de Spark), evalúa la fórmula.
3. Cuando acaba de procesar la ventana, devuelve el resultado obtenido de la evaluación de la fórmula.

Puesto que cada ventana es un instante de tiempo, el resultado debería ser el obtenido de evaluar la fórmula a todas las ventanas generadas en conjunto. Sin embargo, en esta primera versión, la fórmula se evaluaba a cada ventana de forma individual. En la siguiente iteración, explicada a continuación, se corrigió este problema.

4.4.2 Segunda iteración

En esta segunda iteración, aprovechando que la función **`aggregate`** es una clase y se van a crear objetos de ella, se añadió un atributo a la clase en el que se guarda la fórmula según se va evaluando al consumir ventanas. De este modo se soluciona el problema encontrado en la primera iteración y se logra que la fórmula se evalúe usando todas las ventanas.

Sin embargo, debido a cómo está hecha la función **`aggregate`** de Flink, obliga a añadir al *stream* resultante un dato por ventana procesada. Debido a esto, el *stream* resultante contiene todas evaluaciones por las que ha ido pasando la fórmula y no hay modo de saber cuál es el resultado final.

4.4.3 Tercera iteración

El problema anterior, tras pensar y probar diferentes opciones, se resolvió acompañando el resultado de cada evaluación de la fórmula con un indicador de si es el resultado final o no, obteniendo así un *stream* de parejas con estos dos valores.

No obstante, no es tan sencillo saber cuándo se obtiene el resultado final. Los *streams* de Flink son infinitos y no hay modo de distinguir cuál es el último elemento del *stream*. Para solucionar esto se decidió modificar la función encargada de pasar el **`ListStream`** a ventanas para que añada una lista con un **`null`** como único elemento al final. De este modo, es posible identificar cuándo la fórmula obtiene su evaluación final comprobando que la ventana contiene esta lista.

Finalmente, se filtran los datos del *stream* resultante de aplicar la función **`aggregate`** desechando aquellos elementos en los que el indicador sea falso, y a continuación se transforma el *stream* para que solo contenga el resultado de la fórmula en lugar de la pareja entera.

4.4.4 Función test completa: iteraciones y mostrar resultados

En la última iteración se añadió la posibilidad de realizar varios tests consecutivos con el generador y la fórmula dados. Esto se consiguió ejecutando la función **`test`** tantas veces como se indica, y uniendo los *streams* de manera recursiva.

La última funcionalidad añadida fue contar los resultados obtenidos. Para ello, se utiliza una segunda función **`aggregate`** que cuenta el número de valores **`True`**, **`False`** y **`Undecided`**, guardando estos valores en un atributo de la clase.

Como vuelve a repetirse el problema de identificar el resultado final, se ha utilizado la misma técnica que con la anterior función **`aggregate`**, devolviendo los resultados en parejas con un valor que indica si es el resultado final y un *string* indicando el número de tests pasados, fallados o indecisos.

Finalmente, se filtra el *stream* resultante para eliminar todos los resultados no finales y transformar el *stream* de parejas para que solo contenga el mensaje final.

4.6 Casos de prueba

En esta sección se explica en qué consisten los casos de prueba desarrollados para probar la función **`test`**, explicando en qué consiste cada uno, los generadores de datos implementados, y la fórmulas utilizadas en cada prueba. Estas pruebas pueden encontrarse en:

https://github.com/Valcev/TFM/tree/master/tfm/src/main/scala/org/demo

4.6.1 Polución

En este apartado se habla de los generadores y las pruebas realizadas para aplicar *random testing* al problema de detectar polución. Este problema fue ya introducido en la sección 4.2.

Al igual que en la aplicación de detección de polución inicial, se tienen varios sensores que detectan los niveles de polución, y una alarma que se activa y desactiva dependiendo de los datos que envían los sensores.

Sin embargo, en este caso el funcionamiento no es exactamente el mismo. En la primera aplicación todos los sensores generaban el mismo número de datos, mientras que en esta prueba tanto el sensor como el dato generado son escogidos de forma aleatoria. Otra diferencia se encuentra en el criterio seguido para activar y desactivar la alarma, que en este caso debe encontrar, en una ventana, un número determinado de datos que superen la polución máxima para activarse, o que no la superen para desactivarse. Además, se cuentan de manera independiente los datos de cada sensor, aunque posteriormente, para simplificar, solo se comprueba que la alarma se haya activado o desactivado en alguno de los sensores.

Para esta prueba se han desarrollado generadores de datos especiales que faciliten crear datos para la prueba que sigan las especificaciones del problema. Estos generadores son usados en conjunto con los generadores genéricos desarrollados que siguen las especificaciones de LTLss y los generadores ofrecidos por la biblioteca Scalacheck.

Los generadores desarrollados son:

- Generador de polución en *N* sensores: crea parejas de la forma (*valor*, *id_sensor*) en las cuales el *valor* es un entero mayor o igual que el límite a partir del cual se considera que hay polución, así se garantiza que se generan siempre valores con polución. El id del sensor (*id_sensor)* siempre será un entero entre 1 y *N.*
- Generador de no polución en *N* sensor*es*: es igual al anterior, solo que los valores generadores son estrictamente menores que el límite a partir del cual se considera que hay polución.
- Generador de valores acotados entre un mínimo y un máximo en *N* sensores: crea parejas (*valor*, *id_sensor*) donde *valor* está acotado entre el mínimo y el máximo

indicado, sin preocupar si sobrepasan o no el límite de polución.

- Generador de sensores: devuelve identificadores de sensor acotados entre 1 y un argumento indicado. Esto se utiliza en los generadores anteriores para que los sensores que detectan polución sean aleatorios.

Para cada prueba de las enumeradas a continuación se ha usado un generador de datos compuesto por varios generadores de los arriba mencionados y una fórmula de LTLss que evaluar mediante la función **`test`**.

La primera comprobación consiste en verificar que la generación de los identificadores de los sensores es correcta. Para esto se han usado los siguientes generadores:

- Generador de sensores: se pueden dar identificadores de 1 a *N*.
- Generador de datos de polución acotados entre 1 y 100.

Se busca probar que los identificadores generados en cualquier ventana van a ser siempre menores al número total de sensores. Para esto se usa la función auxiliar **`sensorIdLessThan`***(ventana_actual, id_sensor),* que confirma si en la ventana actual hay algún identificador mayor al total de sensores. Para confirmar que se cumple en todas las ventanas, tenemos la formula:

formula = always(sensorIdLessThan(ventana, sensor_generado+1))

La siguiente prueba busca probar que nunca hay polución. Se usan los siguientes generadores:

- Generador de polución para *N* sensores.
- Generador **`always`** aplicándolo al anterior generador durante *M* ventanas. Así garantizamos que durante *M* ventanas nunca se genere polución.

Se ha creado la función **`checkNoPol`***(ventana,umbral)* que comprueba que los valores de la ventana son menores al umbral para comprobar que no hay polución. Con esto, la fórmula resultante es sencilla:

formula = always(checkNoPol(ventana, umbral))

En contraposición a la prueba anterior, a continuación se busca constatar que siempre hay polución. La casuística es similar al anterior caso. Se cuentan con los siguientes generadores:

- Generador de no polución para *N* sensores.
- Generador **`always`** aplicándolo al anterior durante *m* ventanas.

De manera análoga, se ha creado una función **`checkPol`***(ventana,umbral)* para comprobar que siempre hay polución en la ventana. De esta manera:

formula = always(checkPol(ventana, umbral))

Una vez se han desarrollado estos primeros tests, se pasa a algunos con mayor complejidad. El siguiente tiene como objetivo comprobar que en algún momento se encuentra polución.

Los generadores usados son:

- Generador de valores acotados de polución.
- Generador **`always`** usando el generador anterior para producir datos durante *m* ventanas.

La fórmula utilizada es:

formula = eventually(checkPol(ventana,umbral))

Nótese que esta fórmula no tiene por qué ser siempre cierta, pero en el capítulo siguiente se verá que debido a los valores de cota elegidos y al número de ventanas es muy difícil que no se evalúe como cierta.

Otra prueba que se ha realizado es comprobar que, en los datos generados, o bien no hay polución hasta que hay polución, o bien hay polución hasta que deja de haberla. En las anteriores pruebas una ventana contenía una colección de parejas generadas, pero en esta prueba no interesa esto, así que se va a producir un dato por ventana. Los generadores usados son los siguientes:

- Generador de valores acotados de polución.
- Generador de ventanas de un elemento aplicándolo al generador anterior.
- Generador **`always`** del generador anterior en *N* ventanas.

Se definen las siguientes fórmulas auxiliares:

- *noPolUntilPol = checkNoPol(ventana,umbral) until checkPol(ventana,umbral)*
- *polUntilNoPol = checkPol(ventana,umbral) until checkNoPol(ventana,umbral)*

La fórmula final es: *formula = noPolUntilPol or polUntilNoPol*

Las pruebas hasta ahora solo han inspeccionado de manera estricta datos relativos a la polución, pero a diferencia de la aplicación original desarrollada para detectar polución de momento no ha habido ningún sistema de activar o desactivar alarmas. A partir de ahora las pruebas van a usar el sistema explicado al inicio de este apartado.

La primera prueba de este estilo va a consistir en comprobar que si siempre hay polución, entonces habrá al menos una alarma activa. Para esto se usan los siguientes generadores:

- Generador de polución.
- Generador **`always`** aplicado al anterior generador.

Se ha creado la función **`alarmController`***(ventana,sensor,umbral,trigger)* para informar de que una alarma está activa, aunque también se encarga de controlar cuando la alarmas deben activarse y desactivarse. También se han definido las siguiente fórmulas auxiliares:

- *alwaysPol = always(checkPol(ventana,umbral))*
- *eventuallyAlarm = eventually(alarmController(ventana,sensor,umbral,trigger))*

Usando estás fórmulas auxiliares, la fórmula final a aplicar es:

$$formula = alwaysPol \rightarrow eventuallyAlarm$$

Otra comprobación con el sistema de alarmas consiste en asegurarse de que, si no hay polución y más tarde la hay, implica que no había alarmas activas y más tarde se activarán. Aquí se usan los siguientes generadores:

- Generador de polución.
- Generador de no polución.
- Generadores que aseguren que se produce un número específico de datos en los anteriores generadores.
- Generador **`until`** aplicado a lo anterior, para que nos aseguremos de que se generen ventanas sin polución hasta que se generen con polución.

Se ha usado la fórmula auxiliar **`noPolUntilPol`** definida antes, y además se ha definido la siguiente fórmula auxiliar:

noAlarmUntilAlarm = not(alarmController(...)) until alarmController(…)

La fórmula final es: $formula = noPolUntilPol \rightarrow noAlarmUntilAlarm$

Finalmente, con el fin de demostrar toda la potencia de los generadores y las fórmulas, se ha propuesto una fórmula que no tiene sentido lógico en el problema que se está tratando, pero que sirve para ilustrar que hasta se podría comprobar este tipo de situaciones.

La prueba consiste en confirmar que en algún momento habrá polución y no la habrá a la vez. Para esta prueba se han usado los siguientes generadores:

- Generador de polución.
- Generador de no polucion.
- Generador **`release`**, para producir datos sin polución y en algún momento generar polución y no polución en el mismo instante; o bien nunca crear datos con polución.

La fórmula es análoga al generador:

formula = checkAnyPol(ventana, umbral) release checkAnyNoPol(ventana, umbral)

donde las funciones anteriores buscan que exista algún dato mayor que el umbral (no polución) o mayor que el umbral (polución).

Hay que recordar que en las fórmulas temporales anteriores es necesario indicar el número de instantes de acuerdo a las normas establecidas en LTLss. Tanto los instantes usados, como los valores concretos de las variables indicadas se indican en el capítulo 6.

4.6.2 Carreras

En este apartado se habla de los generadores y las pruebas realizadas para aplicar *random testing* a la siguiente situación:

Se celebran carreras de atletismo en las que participan *N* corredores. Se conoce un límite de distancia máxima razonable que un corredor no puede recorrer para un tiempo *t* de manera natural. Si se descubre que un corredor ha recorrido más de esta distancia para el tiempo definido, se considera que ha consumido estupefacientes y hay que descalificarlo de la carrera.

Para representar una carrera se ha decidido usar una lista de parejas, donde cada pareja representa a cada corredor. Las parejas siguen el formato *(corredor, velocidad mínima, velocidad máxima, distancia recorrida, autorizado, ganador),* siendo *corredor* un *string*, *velocidad mínima*,*velocidad máxima* y *distancia recorrida* números enteros, y donde *autorizado* indica si se han encontrado o no anomalías en el corredor, y *ganador* indica si es el ganador de la carrera.

Los generadores desarrollados para crear datos de acuerdo a esta representación son:

- Generador de la carrera inicial: este generador recibe una lista con los nombres de los participantes, y una velocidad mínima y máxima, y genera listas con los participantes y las velocidades mínima y máxima de cada uno. Por defecto, los participantes no se han dopado, no han recorrido ninguna distancia y no son ganadores.
- Generador de nueva posición: crea listas con nuevas posiciones de los corredores, eligiendo velocidades aleatorias acotadas entre la velocidad mínima y máxima indicadas en cada corredor. Dependiendo del valor de estas velocidades es posible que el corredor recorra una distancia mayor a la razonable, y se considere que se ha dopado.
- Generador de carrera: este generador permite originar distintas carreras. Controla si ha habido un ganador en el último dato generado para eliminarlo de los nuevos corredores generados, si hay algún dopado para eliminarlo y el momento en el que todos los participantes han acabado la carrera. Este generador usa los

generadores anteriores.

- Generador de *N* carreras: crea varias carreras. Como los controles del ganador y de cuándo se acaba una carrera se llevan de manera individual en el generador de carreras individuales, este generador no tiene excesiva complejidad.

Para la implementación de todos los controles anteriores, se manejan diferentes variables comunes a todos los generadores:

- Una variable con la velocidad a partir de la cual se considera que un corredor se ha dopado.
- Un indicador que inicialmente es falso y cambia a cierto cuando hay un ganador de la carrera que está generándose. De este modo, podemos controlar quien llega primero y no hacer ganador a los que superan la meta después, ya que para determinar un ganador se comprueba que haya cruzado la meta y aun no haya ganador.
- Un **`Map`** de Scala que guarda cada corredor junto a su posición actual, la cual se va incrementando a medida que el generador avanza en la carrera y sirve para medir la distancia recorrida entre un instante y el anterior, y su estado (si ha sido expulsado o sigue en la carrera).

Igual que en el apartado anterior, se han desarrollado diversas pruebas que usan los generadores desarrollados, fórmulas y la función **`test`**.

La primera prueba consiste en una comprobación de que las carreras se generan con el número de corredores debido. En esta prueba el único generador usado es el generador de una carrera. La fórmula busca confirmar que el número de corredores en cada ventana es menor o igual que el número de corredores indicado al generador (puede ser menor porque algún corredor se dope y se le descalifique o porque algún corredor acabe la carrera). La fórmula resulta:

formula = always(ventana.size <= numCorredores)

La segunda prueba consiste en ver si en algún momento algún corredor se acabará dopando. Se usa un generador de una única carrera. La fórmula verifica si en la pareja ya viene marcado el corredor como dopado, o por si acaso, si en este mismo instante la distancia recorrida ha superado la razonable para alguien no dopado. La fórmula usada es:

formula = eventually(ventana.findAny { runner is banned or wrongSpeed })

La tercera prueba busca comprobar que, si se detecta que un corredor en concreto ha recorrido una distancia no permitida, entonces el corredor estaba en un estado de “autorizado” y pasa a un estado de “no autorizado” y será eliminado de la carrera.

Solo es necesario usar un generador de una carrera. Lo que sí ha sido necesario es definir funciones y fórmulas auxiliares para hacer la fórmula final más legible, estas son:

- **`runnerSpeedWrong`**: busca al corredor en la ventana y si está comprueba si la distancia que ha recorrido hace que se considere que se ha dopado.
- **`runnerBanned`**: busca al corredor en la ventana y si está comprueba si ha sido descalificado.
- **`NotBannedUntilBanned`**: comprueba que un corredor estaba autorizado en la carrera antes de dejar de estarlo: *not(runnerBanned) until runnerBanned*
- **`eventuallyDeleted`**: busca que un corredor en particular sea eliminado de la carrera en algún momento (debido a que tras descalificar a un corredor, este se elimina de la carrera, pues ya no sigue participando en ella): *eventually(not(ventana.contains(corredor)))*

La fórmula final resultante es:

$$formula = runnerSpeedWrong \rightarrow (NotBannedUntilBanned \text{ and } eventuallyDeleted)$$

Como cuarta prueba, se busca confirmar que un corredor en particular, si se ha dopado y le han descalificado, nunca gane una carrera. Para aumentar un poco la complejidad de esta prueba se ha usado el generador de varias carreras, así por ejecución hay más ejemplos que comprobar.

Además de usar la fórmula **`runnerBanned`** definida antes, se usan las siguientes fórmulas auxiliares:

- **`runnerWins`**: busca un corredor en particular y comprueba si ha ganado la carrera.
- **`neverWins`**: busca que un corredor en particular nunca gane una carrera: *always(not(runnerWins))*

La fórmula final resultante es: $runnerBanned \rightarrow neverWins$

Esta última prueba ilustra lo que vería el usuario cuando un test falla, ya que estas pruebas no solo sirven para probar hechos ciertos. Para ello, se ha propuesto comprobar que un corredor en particular que nunca ha sido pillado dopado siempre gana alguna carrera. Esto se puede ver que es falso, puesto que es bastante probable que más de un corredor nunca se dope en una carrera y, sin embargo, solo puede ganar uno de ellos. También puede ocurrir, aunque con probabilidad bastante baja, que sea ese corredor en particular el que gane la carrera y no se haya dopado, ya sea porque es el más rápido o porque todos los demás se han dopado y han sido eliminados de la carrera.

Para esta prueba se usa el generador de varias carreras para confirmar si el corredor gane alguna de ellas, y también, igual que antes, para aumentar la posibilidad de que se generen distintas situaciones. Como fórmulas auxiliares se usan **`runnerBanned`** y

`neverBanned`, y se han definido las siguientes:

- **`eventuallyWins`**: busca un corredor en particular en la ventana y comprueba que eventualmente sea ganador: *eventually(isWinner(ventana.get(corredor)))*
- **`alwaysWins`**: comprueba que siempre el corredor en particular gana alguna carrera: *always(eventuallyWins)*

La fórmula final resultante es: *formula* = *neverBanned* → *alwaysWins*

Igual que en el apartado anterior, se han omitido los instantes aplicados a las fórmulas. En el capítulo de resultados se especifica el número de corredores para cada caso, el número de carreras generadas, y los tiempos de las fórmulas.

4.7 Estudio de licencias software

En este apartado se va a realizar un estudio de las licencias de *software* de las bibliotecas utilizadas en el proyecto, pretendiendo comprobar si esto afecta a la licencia *software* que tendría la aplicación desarrollada en este proyecto:

- Scalacheck: usa una licencia BSD de 3 cláusulas [25], un derivado de la licencia BSD. A diferencia de las licencias GPL, la licencia BSD es permisiva y autoriza el uso de código con licencia BSD en un código no libre. Las únicas restricciones que añadiría al proyecto el uso de esta biblioteca es incluir la cláusula de *copyrigth* original y no poder usar el nombre de la entidad que desarrolló Scalacheck ni de sus colaboradores sin su permiso para promocionar este proyecto.
- Specs2: usa una licencia MIT [26]. La licencia MIT es permisiva, como la licencia BSD, y no obliga a liberar el código de sus derivados. Solo obliga a incluir el aviso de copyright para referenciar a los autores originales.
- Flink: usa una licencia Apache 2.0 [27], La licencia Apache 2.0 es permisiva, como las anteriores. Las restricciones adicionales con las que cuenta es que las partes no modificadas de código deben preservar la licencia Apache 2.0, y si se cambia el texto del aviso de licencia, indicar los cambios que se han hecho respecto al original. Como Flink solo se usa como biblioteca, y no se ha modificado ni incluido código original, la licencia no afecta al proyecto.
- Scalaz: usa una licencia BSD de 2 cláusulas [28], a efectos de este proyecto, equivalente a la BSD de 3 cláusulas como la usada por Scalacheck. No afecta a la licencia de este proyecto.
- Sscheck: usa una licencia Apache 2.0 [29]. En este caso, sí que hay efectos en la licencia de partes del proyecto, puesto que hay código fuente tal cual del proyecto Sscheck que se ha usado en el proyecto. En particular, la clase Formula debe mantener la licencia Apache 2.0.
- SBT: licencia Apache 2.0 [30]. Mismos efectos en el proyecto que Flink, no se ha

usado código tal cual de SBT, solo se ha usado como herramienta para facilitar la resolución de dependencias del proyecto.

Como conclusión, se puede ver que no se ha usado ninguna biblioteca que influya en la licencia final del *software*, salvo Sscheck, en cuyo caso la clase Formula debe mantener la licencia Apache 2.0. De acuerdo a las demás licencias, se deben incluir las referencias a las bibliotecas usadas y sus autores.

Finalmente, y teniendo en cuenta todas las licencias anteriores, se ha decidido que este proyecto tenga la licencia MIT. Como ya se ha dicho, esta es una licencia de *software* libre, con muy pocas limitaciones de reutilización y por ello muy compatible con otras licencias, obligando solo a incluir el aviso de copyright para referenciar a los autores originales. Además, de este modo se mantiene la coherencia con el resto de licencias de las que se ha hablado.

Capítulo 5: Técnicas de testing basado en lógica temporal para Apache Flink

En este capítulo se han incluido diferentes implementaciones que no necesariamente tienen como aplicación el *random testing*, pero hacen uso de fórmulas temporales y pueden tener utilidades interesantes.

5.1 FormulaTrigger

Durante el proceso de pensar cómo aplicar una misma fórmula temporal a un conjunto de ventanas, se decidió utilizar un disparador, ya que los disparadores pueden mantener una referencia a un descriptor de estado común a todos los elementos del **`WindowedStream`**. Pero lo único que puede hacer un disparador es 'distribuir' los datos que recibe de acuerdo al criterio implementado en él, y no puede modificarlos ni devolver ningún valor, por lo que hubo que buscar una alternativa. Además, para aplicar el disparador era necesario que los generadores crearan los elementos en una única *global window* a la que aplicar el disparador.

Aunque el disparador desarrollado no sirvió para el fin inicial que se le había otorgado, se vio el potencial y la gran cantidad de aplicaciones útiles que tenía fuera del *random testing*, y se decidió desarrollar la idea por completo. Esto no quiere decir que no exista ninguna aplicación del disparador en el ámbito del *random testing*, se podría seguir madurando la idea y encontrar diversos usos dentro de este.

El disparador desarrollado tiene como objetivo ejecutarse cuando los elementos de la *global window* cumplan o no cumplan con una fórmula temporal incluida en el constructor del disparador, lo cual podría utilizarse, por ejemplo, para encontrar los ejemplos que hacen que una fórmula falle. Este disparador se ha denominado ***`FormulaTrigger`***.

El funcionamiento de ***`FormulaTrigger`*** es el siguiente:

1. Se construye el disparador y se guarda la fórmula de entrada dentro de un descriptor de estado.
2. Al recibir un elemento, se accede a la fórmula y se aplica al elemento de la ventana, consumiendo un instante de tiempo.
3. Si la fórmula deja de estar decidida, es decir, es cierta o falsa, se ejecuta el disparador devolviendo todos los elementos tratados hasta el momento, incluido el responsable de ejecutar el disparador, se purgan los elementos, y se reinicia la fórmula. Si no, se continúa.

Como los disparadores no manejan ventanas, hubo que cambiar la definición de instante que se manejaba hasta ahora. Ahora se considera un instante como la consumición de un elemento de la ventana. Por este motivo solo se ha implementado el método **`onElement`**

del disparador. Esta definición podría cambiar y se podría usar el **`onEventTime`** o el **`onProcessingTime`** del mismo modo que están implementadas las *time windows*, para actualizar el estado y consumir un instante en la fórmula.

Para que el descriptor de estado pueda guardar la fórmula se ha usado un descriptor de estado de agregación. De hecho, se ha podido reutilizar la primera versión que se desarrolló de la función de agregación utilizada en la función test, explicada en el apartado anterior.

En la figura 5.1 se ilustra el funcionamiento del disparador para producir ventanas, el cual se ejecuta cada vez que encuentra seis ‘1’ seguidos o un número distinto de ‘1’. En la figura se ha representado como si se generaran distintas ventanas, pero hay que recordar que en realidad solo hay un **`WindowedStream`** y el disparador solo sirve para decidir cada cuántos elementos se ejecutan las funciones indicadas del **`WindowedStream`**. **`FormulaTrigger`** por sí mismo no produce ningún tipo de cómputo.

En el apartado 5.4.1 se realizará una prueba más representativa de la utilidad de este disparador, y en el apartado 6.3 se analizarán los resultados de la prueba.

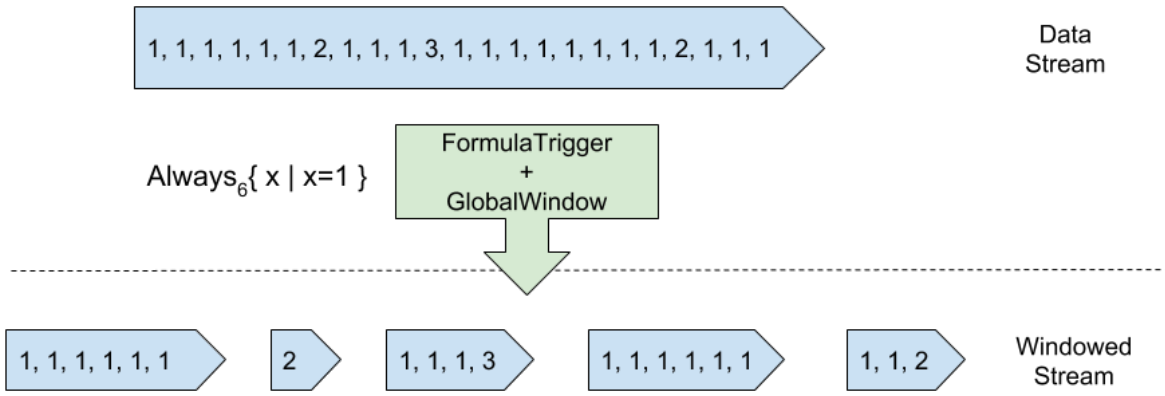

Figura 5.1: Ejemplo de aplicación de **`FormulaTrigger`**

Igual que se ha definido que salte este disparador cuando la fórmula deja de ser no decidida, podría pensarse en disparadores similares que saltasen cuando ocurran distintas condiciones, como que saltase solo cuando la fórmula fuese cierta o solo cuando fuese falso. Ya que estos disparadores compartirían el estado de agregación podría pensarse en una jerarquía de herencia para su implementación.

5.2 KeyedStreamTest

El desarrollo de **`KeyedStreamTest`** se produjo de una forma parecida al desarrollo de **`FormulaTrigger`**. Se decidió buscar los distintos métodos que permitieran un acceso a un estado común a todas las ventanas del *stream* y, entre estos métodos que ofrece Flink, hay uno que permite aplicar una función común a un **`KeyedStream`**.

De la misma forma que pasa con **FormulaTrigger**, al final este desarrollo no tuvo aplicaciones prácticas en el desarrollo de los generadores o de la función **test**. Sin embargo, sigue siendo útil para la aplicación de fórmulas de Lógica Temporal en *streams* de Flink. De hecho, **KeyedStreamTest** cubre un espacio que **FormulaTrigger** no cubría, y es la evaluación de una fórmula de Lógica Temporal en un **KeyedStream**, un **DataStream** divido por clave.

El modo de acceder ejecutar una función en un **KeyedStream** se realiza creando una clase que implemente una de las distintas interfaces de operaciones ofrecidas por la API de Flink. La clase que implemente la interfaz debe implementar un método **open** que contenga la inicialización de la clase, y otro método dependiente del constructor que debe contener el proceso en sí que se va a ejecutar.

Igual que pasa en los otros casos en los que hay que acceder a un estado en Flink, se ha de usar un descriptor de estado para ir guardando el estado. Y también, al igual que ha ocurrido en apartados anteriores, se va a guardar en el estado una fórmula en forma next, que es la que va consumiendo el tiempo restante y actualizando su evaluación.

Dado que en los **KeyedStreams** es normal que se trabaje con parejas de valores, ya que al crear el **KeyedStream** hay que indicar una clave, se ha decidido implementar una función **richMap**. Aunque la fórmula se evalúe a un solo resultado, esto permite a la clase ser fácilmente depurable al ir devolviendo los distintos estados por los que pasa la fórmula; y también permite que la clase sea lo suficientemente flexible como para que se cambie su funcionamiento si se quiere, ya que una función **richMap** no tiene que devolver siempre el mismo número de elementos que la colección sobre la que se ha aplicado.

El funcionamiento que tiene la clase es el siguiente:

- El constructor recibe una fórmula e inicializa el descriptor de estado.
- Al recibir un elemento se evalúa la fórmula consumiendo un instante de tiempo.
- Si la fórmula está resuelta se vuelcan todos los resultados conseguidos hasta el momento y se reinicia la fórmula.

Se puede observar que el modo de operar es muy parecido al de **TriggerWindow**, solo que en vez de aplicarse a un **WindowedStream**, esto puede aplicarse a un **KeyedStream**.

En la figura 5.2 se puede ver un ejemplo de aplicación que abarca desde el **DataStream** original hasta el **DataStream** resultante de aplicar **KeyedStreamTest**. En este ejemplo, la fórmula busca que el segundo valor de cada pareja sea menor que 10, y debe cumplirse en seis parejas seguidas. Aunque el *stream* resultante de la figura parece desordenado, siempre se puede (y de hecho será lo habitual) generar el resultado manteniendo el formato de las parejas del **KeyedStream**. Sin embargo, se ha decidido no hacerlo en este ejemplo para ilustrar que se puede devolver de manera pura el resultado de evaluar la fórmula.

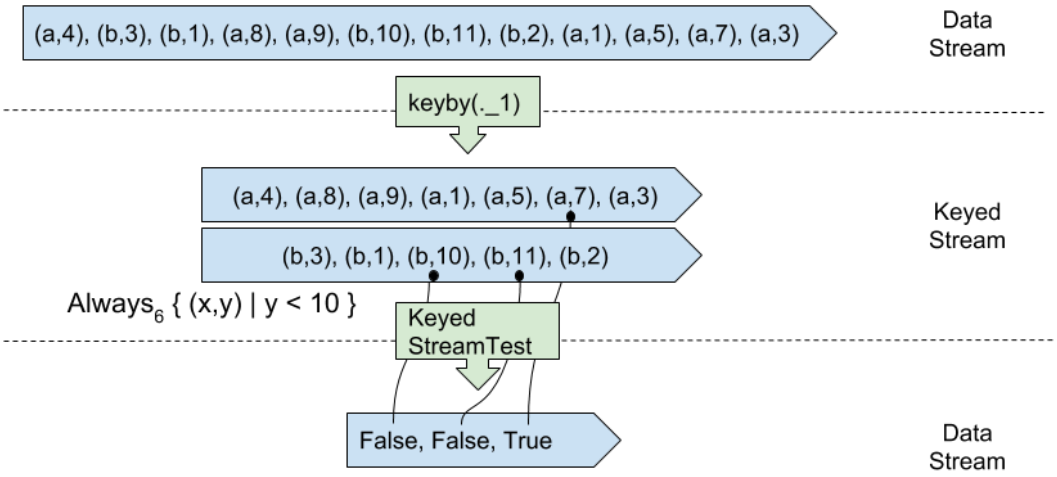

Figura 5.2: Ejemplo de aplicación de **`KeyedStreamTest`**

5.3 StreamTest

La clase **`StreamTest`** es una adaptación de la primera aplicación de toma de contacto con Flink para poder probarla con la función **`test`**. Para esto también fue necesario adaptar la función **`test`**, por lo que existe una versión que, en lugar de recibir un generador, recibe directamente un *stream* dividido en ventanas.

Lo primero fue modificar la función de Flink original que trata los datos de polución para que, en lugar de ir mostrando las alarmas por pantalla, devuelva el *stream* con los datos que indican cuando una alarma debe activarse, desactivarse o permanecer igual. El generador de datos es el mismo que en la aplicación original, sin modificaciones.

Por último, la función **`test`** utilizada para esta situación recibe un *stream* dividido en ventanas (la división en ventanas es necesaria ya que no es posible aplicar una función **`aggregate`** a un **`DataStream`**) y una fórmula de LTLss. Se consideró la opción de que recibiera un *stream* y se encargara de divirlo en ventanas la propia función **`test`**, pero la idea se acabó descartando para dar la opción al usuario de realizar la división a su gusto.

Al igual que en la función **`test`** original, el *stream* devuelto por la función **`aggregate`** se filtra para que solo devuelva el resultado final. Sin embargo, esta implementación se quedó en la primera fase, y por ello no tiene un mecanismo tan fiable como la función original para determinar el resultado final; solo constata si la fórmula ha sido evaluada a cierto o a falso para saber cuál es el resultado final, por lo que, en los casos en los que el resultado sea **`Undecided`**, no mostrará ningún resultado. Asimismo, también carece de la opción de realizar varios tests seguidos sobre un mismo *stream* con la fórmula dada, ya que está funcionalidad se añadió un poco más adelante y no hubo tiempo para incluirla en esta versión.

5.4 Casos de prueba

Este apartado contiene los casos de prueba desarrollados para las implementaciones explicadas en este capítulo, los cuales también sirven como ejemplo de las aplicaciones

que pueden tener estos desarrollos. Cada apartado comienza con una descripción del problema a resolver y continúa con la descripción de cada prueba. En el capítulo 6 se muestran los datos utilizados en concreto y los resultados obtenidos. Estas pruebas pueden encontrarse en:

https://github.com/Valcev/TFM/tree/master/tfm/src/main/scala/org/test

5.4.1 Prueba de FormulaTrigger

Twitter es una gran red social especialmente usada para dar a conocer tus gustos y opiniones sobre distintos temas, además de permitir a la gente reenviar tus comentarios a otros y darlos a conocer. Sin embargo, Twitter tiene un número máximo de caracteres que se pueden escribir por tuit, por lo que dar a conocer complejas opiniones o entablar tranquilas e informadas discusiones con otros resulta complicado.

Para solucionar esto, muchos tuiteros recurren a usar secuencias de caracteres especiales que indiquen que el mensaje que quieren transmitir no se acaba en su tuit o en su respuesta a otro tuit y que en el siguiente tuit el mensaje continúa. Un ejemplo de esto es la siguiente secuencia (figura 5.3) de tuits del presidente de los Estados Unidos, Donald Trump (@realDonaldTrump):

Figura 5.3: Ejemplo de único mensaje en varios tuits

Se puede ver que acaba los tuits con puntos suspensivos para indicar que su mensaje continúa en el siguiente tuit, y que comienza con puntos suspensivos para indicar que es necesario leer un tuit anterior para entender el mensaje. Se quiere encontrar una manera de obtener estos tuits relacionados de manera separada al resto de tuits, para que así puedan ser tratados en su justo contexto.

La solución planteada consiste en la aplicación de una fórmula de LTLss, junto con **`FormulaTrigger`**, para que se devuelvan los tuits no relacionados en ventanas independientes, y los tuits relacionados entre sí en la misma ventana.

Flink cuenta con conectores a Twitter para poder capturar los tuits en un **`DataStream`**, pero para simplificar las pruebas, se ha decidido copiar algunos de sus tuits dentro de una lista de Scala, y hacer que el origen de los datos sea esta colección.

Para resolver el problema se proponen las siguientes fórmulas:

- *comienzoEstricto(tuit) = tuit.endsWith("...") and not(tuit.startsWith("..."))*
- *finalEstricto(tuit) = tuit.startsWith("...") and not(tuit.endsWith("..."))*
- *mensajeIntermedio(tuit) = tuit.startsWith("...") and tuit.endsWith("...")*

Estas fórmulas se usan para reconocer si un tuit es el inicio de una cadena de tuits relacionados, si es el último tuit de una cadena de tuits relacionados, o si es un tuit intermedio que forma parte de una cadena de tuits relacionados.

Queremos que la secuencia de tuits relacionados vaya en la misma ventana, por lo cual buscamos una fórmula que devuelva cierto o falso cuando los tuits no están relacionados con ningún otro tuit; y en el caso de que encontremos un tuit que cumpla *comienzoEstricto(tuit)*, busque que todos los tuits siguientes cumplen *mensajeIntermedio(tuit)* hasta que se encuentre un tuit que cumpla *finalEstricto(tuit)*. La fórmula ideada es:

formula(tuit) = comienzoEstricto(tuit) →
always (mensajeIntermedio(tuit)) until (finalEstricto(tuit))

Como **`FormulaTrigger`** se dispara cuando la fórmula deja de ser **`Undecided`**, esta fórmula asegura que los tuits individuales vayan cada uno en una ventana al devolver la fórmula cierto por ser falsa la premisa. También asegura que si el tuit acaba con puntos suspensivos, la fórmula sea **`Undecided`** hasta que se encuentre con el último tuit de la cadena o alguno intermedio no cumpla la condición.

Para adaptar la fórmula a las especificaciones del proyecto, falta definir un *timeout*. El *timeout* necesita ser lo suficientemente grande como para que la fórmula no alcance un resultado final antes de lo esperado. En el capítulo de resultados se especifica el *timeout* aplicado de acuerdo a los datos introducidos.

Finalmente, para probar el correcto funcionamiento de este test, se aplica una función

reduce a las ventanas de tal manera que se sustituyen los puntos suspensivos finales por la cadena vacía y se concatenan los tuits separándolos por espacio.

5.4.2 Prueba de KeyedStreamTest

En este ejemplo se dispone de una aplicación médica conectada con un sensor que mide el nivel de actividad física de varios pacientes. El paciente debe tener el nivel de actividad entre ciertos valores considerados normales y, si se encuentra un valor más bajo que el normal, se deberá encontrar en las seis siguientes mediciones un valor más alto de lo normal para que se considere que la actividad está en equilibrio. Igual ocurre con las mediciones más bajas de lo normal, una vez encontrada una, deberá aparecer una más alta de lo normal para equilibrar.

Se quiere controlar para cada paciente que sus mediciones sean equilibradas según el apartado descrito anteriormente y, si no lo son, se indique lo ocurrido. Para ello, se usará una colección de pares de la forma (*nombre de paciente, valor*), siendo *nombre de paciente* un *string* y *valor* un entero, para simular la entrada de datos. El **`DataStream`** se genera a partir de esta colección, y se transforma en un **`KeyedStream`** usando el nombre del paciente como clave para separar el flujo y tratar los datos de cada paciente por separado.

Una vez se dispone del **`KeyedStream`**, se usa **`KeyedStreamTest`** para aplicar una fórmula de LTLss que compruebe los requisitos anteriormente mencionados. Se ha modificado la clase **`KeyedStreamTest`** para que no solo devuelva el resultado de evaluar la fórmula cuando deje de estar sin decidir, sino para que devuelva este resultado dentro de una pareja, donde el primer elemento es la clave que se ha usado para separar el **`KeyedStream`** (en este caso el nombre del paciente), y el segundo el resultado de la fórmula.

Se han definido las siguientes fórmulas lógicas:

- *normal(valor) = 3 <= valor <= 7*
- *alto(valor) = valor > 7*
- *bajo(valor) = valor < 3*

Y las siguientes fórmulas de LTLss:

- *eventualmenteBajo(valor) = eventually(bajo(valor)) during 6*
- *eventualmenteAlto(valor) = eventually(alto(valor)) during 6*
- *siempreNormal(valor) = always(normal(valor)) during 12*

Buscamos que los valores medidos sean siempre normales, o si el valor es alto eventualmente será bajo, o si es bajo eventualmente será alto, y si en el estado actual se ha cumplido que el valor no siempre es normal, entonces es porque eventualmente será alto o bajo. La fórmula final es:

formula(valor) = siempreNormal(valor) or (not(normal(valor)) → ((alto(valor) and eventualmenteBajo(valor)) or (bajo(valor) and eventualmenteAlto(valor)))) and (not(siempreNormal(valor) → (eventualmenteBajo(valor) or eventualmenteAlto(valor)))

Gracias a **`KeyedStreamTest`**, la fórmula saltará cuando todos los valores sean normales durante doce mediciones, o si se encuentra un valor fuera de lo normal, se encuentre su correspondiente valor que equilibre en seis mediciones. Si esto no ocurre saltará con un valor **`False`**. Si no hay suficientes valores no se mostrará ningún valor.

5.4.3 Prueba de StreamTest

También se han implementado dos pruebas sencillas para probar la función **`test`** que trabaja directamente con *streams* en lugar de con generadores.

La primera prueba consiste en un *stream* donde cada elemento es un *string*, y se comprueba que todos los *strings* del *stream* contienen una misma cadena durante un número dado de instantes. La fórmula es:

formula(string) = siempre(string contiene cadena)

La segunda prueba obtiene el *stream* generado de la ejecución de la aplicación adaptada (como se explica en el apartado 4.5.3). Este *stream* contiene parejas con un valor que indica si debe activar, desactivar o no modificar una alarma acompañado del id del sensor para el cual debe modificarse el estado de la alarma. Por ello, la fórmula comprueba que los primeros valores sean un 1 (indican que no hay que cambiar la alarma) y vayan seguidos del valor 2 (indican que hay que activar la alarma).

Para esto se tienen dos fórmulas auxiliares:

- *f1((valor, id_sensor)) = valor == 1*
- *f2((valor, id_sensor)) = valor == 2*

La fórmula final es:

formula((valor, id_sensor)) = f1 until f2

El comportamiento es muy similar a la función **`test`** original, solo que en este caso no hay ventanas y la comprobación de la fórmula se realiza directamente sobre cada elemento del *stream*.

Capítulo 6: Resultados

Este capítulo contiene los resultados de los casos de prueba explicados en el capítulo 4.6, junto a un estudio de análisis de rendimiento de los casos de prueba de polución y carreras usando diferentes parámetros para medir los tiempos.

El parámetro *timeout* indica el número de instantes utilizado por la fórmula. Por ejemplo, si es un **`always`**, indica el número de instantes durante el cual se debe cumplir la condición; o si es un **`until`**, de cuántos instantes se dispone para que la condición se cumpla. Cada vez que se consume una ventana representa un instante, por eso ocurre que coincide el número de ventanas con el *timeout.* Aunque esto no es necesario, sí que facilita el entendimiento de las pruebas.

6.1 Polución

A continuación se presentan los distintos parámetros utilizados para cada prueba, así como el número de aciertos, fallos y pruebas sin decidir resultantes.

	Nº de ventanas	Nº de sensores	Umbral de polución	Nº de datos por ventana	Timeout	Nº de tests	Aciertos	Fallos	Sin decidir
sensorIdsOk	10	3	-	1	10	100	100	0	0
neverPol	20	5	20	1	20	100	100	0	0
alwaysPol	20	5	20	1	20	100	100	0	0
eventuallyPol	20	5	20	1	20	100	100	0	0
checkAlarm	20	1	20	5	20	100	100	0	0
checkGenPol	10	4	20	1	10	100	100	0	0
alarmOn	20	1	20	5	20	100	100	0	0
noPolReleasePol	20	1	20	1	20	50	21	29	0

En las pruebas *sensorIdsOk*, *neverPol* y *alwaysPol,* se ha utilizado tanto el generador como la fórmula *always*. Por tanto, los datos que se generan siempre cumplen la misma condición que va a comprobar la fórmula. Es por eso que se obtiene un 100% de aciertos en estas tres primeras pruebas.

En la prueba *eventuallyPol* se generan datos aleatorios dentro de un rango entre 0 y 100, por lo que hay datos que superan el umbral de polución y datos que no. Como la fórmula utilizada comprueba que en algún momento se genera polución, y el umbral definido es 20, la mayoría de las veces esto se cumple, y se puede ver en los resultados, que son todos aciertos. Sin embargo, puede haber ocasiones en las que el generador solo genere valores menores que el umbral y la condición no se cumpla.

La prueba *checkGenPol* es similar a la anterior, pero en lugar de solo comprobar que en algún momento se genera polución, comprueba que si hay polución en algún momento deja de haberla o viceversa. Hay 100% de aciertos porque es muy improbable que solo se generen datos por debajo del umbral o por encima, aunque podría ocurrir.

En la prueba *checkAlarm* se obtiene un 100% de aciertos porque siempre hay polución, por lo que la alarma siempre termina activándose y la fórmula se cumple.

En la prueba *alarmOn,* se comprueba que al principio no haya polución y luego sí, y en consecuencia, la alarma, que inicialmente está desactivada, se termina activando. En este caso también se obtiene un 100% de aciertos porque se utiliza un generador *until* que devuelve valores inferiores al umbral hasta devolver valores que lo superen, y la alarma siempre empieza estando desactivada y se activa en cuanto detecta polución.

Por último, la prueba *noPolReleasePol* funciona de forma similar a las primeras, ya que utiliza el generador y la fórmula *release*, por lo que todos los datos generados siempre cumplen la fórmula. Sin embargo, solo se obtienen 21 aciertos de 50 iteraciones porque el generador y la fórmula *release* siguen definiciones distintas. La fórmula espera que siempre en algún estado se cumplan las dos condiciones, mientras que el generador permite que siempre se cumpla la segunda condición si la primera nunca se cumple.

6.2 Carreras

Al igual que en el apartado anterior, la siguiente tabla muestra los distintos parámetros utilizados para cada prueba y los resultados.

	Nº de corredores	Nº de carreras	Distancia a la meta	Velocidad mínima	Velocidad máxima	Timeout	Nº de tests	Aciertos	Fallos	Sin decidir
racesOk	5	1	50	1	10	10	50	45	0	5
checkSpeeds	5	1	50	1	10	50	50	50	0	0
checkBanned Runner	5	1	900	9	10	100	100	100	0	0
	5	1	900	1	2	100	100	100	0	0
bannedNever Win	5	5	50	10	10	25	50	50	0	0
	5	5	50	1	1	25	50	50	0	0
NotBanned AlwaysWin	5	3	50	5	5	30, 10	50	0	50	0

La prueba *racesOk* comprueba que el número de participantes siempre sea menor o igual que el total. Durante una carrera puede disminuir el número de corredores a medida que llegan a la meta o si alguno es descalificado, por eso esta propiedad siempre es cierta. Sin embargo, como estos eventos son aleatorios, no es posible saber en cuántos estados acabará una carrera, y si se da el caso de que termina antes del *timeout* establecido, no se puede decidir el resultado.

En la prueba *checkSpeeds* se comprueba que en algún momento se descalifica a alguien de la carrera. Debido a que la velocidad de cada corredor es aleatoria dentro del rango

fijado por las velocidades mínima y máxima, y si la velocidad obtenida es mayor que 8 se descalifica al corredor, puede ocurrir que el resultado no pueda decidirse si la carrera ha acabado antes del *timeout* sin que nadie haya sido descalificado, por lo que sabemos que nadie se ha dopado. Por esto mismo, también puede suceder que ningún corredor supere el límite de velocidad y el resultado sea un fallo, aunque es poco probable.

La prueba *checkBannedRunner* se centra en uno de los participantes de la carrera y comprueba que si su velocidad supera el límite, su estado pasa de autorizado a descalificado y se elimina de la carrera. Como se puede observar, se obtiene un 100% de aciertos tanto si se fuerza a que el corredor supere el límite como si se fuerza a que no lo supere nunca, pues en el primer caso, el corredor es descalificado y eliminado, y en el segundo simplemente no se le descalifica nunca porque su velocidad es correcta.

BannedNeverWin comprueba que un participante descalificado nunca gana una carrera. Al igual que el anterior, se centra en participante en concreto, y también se obtiene un 100% de aciertos independientemente de su velocidad porque, en caso de superar el límite, en efecto es descalificado y al ser eliminado ya no sigue avanzando y no puede llegar a la meta, y en caso de no superar el límite no se cumple la premisa de que el corredor es descalificado.

En último lugar, *notBannedAlwaysWin* comprueba si el hecho de que un participante nunca sea descalificado implica que ganará la carrera. Esto rara vez es así, ya que por participar en una carrera no significa que pueda llegar primero, y puede verse en los resultados, que son todos fallos, aunque en escasas ocasiones podría darse algún acierto si el corredor ganase alguna carrera. En esta prueba se manejan dos *timeouts* distintos: el *timeout* de 10 se aplica a la fórmula que comprueba si el corredor gana, ya que esta comprobación se hace en cada carrera; y el *timeout* de 30 se aplica a la fórmula que comprueba que siempre (en todas las carreras) se da la condición de la fórmula anterior.

6.3 Prueba de FormulaTrigger

En la prueba de **`FormulaTrigger`** se han usado textos correspondientes a tuits reales de @realDonaldTrump. En concreto, estos han sido un total de 17 tuits, entre los cuales se encuentran 5 cadenas distintas de tuits relacionados que agrupan 11 tuits distintos, 5 tuits que no forman parte de ninguna, y 1 tuit que probablemente esté escrito con intención de formar parte de una cadena, pero que no cumple las especificaciones descritas por no acabar el anterior tuit en puntos suspensivos. Los tuits utilizados son los siguientes:

"Looking forward to being with the wonderful Bush family at Blair House today. The former First Lady will be coming over to the White House this morning to be given a tour of the Christmas decorations by Melania. The elegance & precision of the last two days have been remarkable!",

"The negotiations with China have already started. Unless extended, they will end 90 days from the date of our wonderful and very warm dinner with President Xi in Argentina. Bob Lighthizer will be working closely with Steve Mnuchin, Larry Kudlow, Wilbur Ross and Peter Navarro.....",

".....on seeing whether or not a REAL deal with China is actually possible. If it is, we will get it done. China is supposed to start buying Agricultural product and more immediately. President Xi and I want this deal to happen, and it probably will. But if not remember,.....",

"....I am a Tariff Man. When people or countries come in to raid the great wealth of our Nation, I want them to pay for the privilege of doing so. It will always be the best way to max out our economic power. We are right now taking in $billions in Tariffs. MAKE AMERICA RICH AGAIN",

".....But if a fair deal is able to be made with China, one that does all of the many things we know must be finally done, I will happily sign. Let the negotiations begin. MAKE AMERICA GREAT AGAIN!",

"Could somebody please explain to the Democrats (we need their votes) that our Country losses 250 Billion Dollars a year on illegal immigration, not including the terrible drug flow. Top Border Security, including a Wall, is $25 Billion. Pays for itself in two months. Get it done!",

"I am glad that my friend @EmmanuelMacron and the protestors in Paris have agreed with the conclusion I reached two years ago. The Paris Agreement is fatally flawed because it raises the price of energy for responsible countries while whitewashing some of the worst polluters....",

"....in the world. I want clean air and clean water and have been making great strides in improving America's environment. But American taxpayers – and American workers – shouldn't pay to clean up others countries' pollution.",

"We are either going to have a REAL DEAL with China, or no deal at all - at which point we will be charging major Tariffs against Chinese product being shipped into the United States. Ultimately, I believe, we will be making a deal - either now or into the future....",

".....China does not want Tariffs!",

"“China officially echoed President Donald Trump's optimism over bilateral trade talks. Chinese officials have begun preparing to restart imports of U.S. Soybeans & Liquified Natural Gas, the first sign confirming the claims of President Donald Trump and the White House that.....",

".....China had agreed to start “immediately” buying U.S. products.” @business",

"Very strong signals being sent by China once they returned home from their long trip, including stops, from Argentina. Not to sound naive or anything, but I believe President Xi meant every word of what he said at our long and hopefully historic meeting. ALL subjects discussed!",

"One of the very exciting things to come out of my meeting with President Xi of China is his promise to me to criminalize the sale of deadly Fentanyl coming into the United States. It will now be considered a “controlled substance.” This could be a game changer on what is.....",

".....considered to be the worst and most dangerous, addictive and deadly substance of them all. Last year over 77,000 people died from Fentanyl. If China cracks down on this “horror drug,” using the Death Penalty for distributors and pushers, the results will be incredible!",

"Looking forward to being with the Bush family. This is not a funeral, this is a day of celebration for a great man who has led a long and distinguished life. He will be missed!",

"Hopefully OPEC will be keeping oil flows as is, not restricted. The World does not want to see, or need, higher oil prices!"

Para las pruebas se ha aplicado a la fórmula, tanto a la parte del *always* como a la parte del *until*, un *timeout* de 4. Tras la aplicación del programa de prueba, se han obtenido 11 mensajes distintos, efectivamente 5 correspondientes a los 11 tuits que forman parte de cadenas, y 6 correspondientes a tuits no relacionados:

"Looking forward to being with the wonderful Bush family at Blair House today. The former First Lady will be coming over to the White House this morning to be given a tour of the Christmas decorations by Melania. The elegance & precision of the last two days have been remarkable!",

".....But if a fair deal is able to be made with China, one that does all of the many things we know must be finally done, I will happily sign. Let the negotiations begin. MAKE AMERICA GREAT AGAIN!",

"The negotiations with China have already started. Unless extended, they will end 90 days from the date of our wonderful and very warm dinner with President Xi in Argentina. Bob Lighthizer will be working closely with Steve Mnuchin, Larry Kudlow, Wilbur Ross and Peter Navarro on seeing whether or not a REAL deal with China is actually possible. If it is, we will get it done. China is supposed to start buying Agricultural product and more immediately. President Xi and I want this deal to happen, and it probably will. But if not remember, I am a Tariff Man. When people or countries come in to raid the great wealth of our Nation, I want them to pay for the privilege of doing so. It will always be the best way to max out our economic power. We are right now taking in $billions in Tariffs. MAKE AMERICA RICH AGAIN",

"Very strong signals being sent by China once they returned home from their long trip, including stops, from Argentina. Not to sound naive or anything, but I believe President Xi meant every word of what he said at our long and hopefully historic meeting. ALL subjects discussed!",

"One of the very exciting things to come out of my meeting with President Xi of China is his promise to me to criminalize the sale of deadly Fentanyl coming into the United States. It will now be considered a “controlled substance.” This could be a game changer on what is considered to be the worst and most dangerous, addictive and deadly substance of them all. Last year over 77,000 people died from Fentanyl. If China cracks down on this “horror drug,” using the Death Penalty for distributors and pushers, the results will be incredible!",

"Could somebody please explain to the Democrats (we need their votes) that our Country losses 250 Billion Dollars a year on illegal immigration, not including the terrible drug flow. Top Border Security, including a Wall, is $25 Billion. Pays for itself in two months. Get it done!",

"“China officially echoed President Donald Trump’s optimism over bilateral trade talks. Chinese officials have begun preparing to restart imports of U.S. Soybeans & Liquified Natural Gas, the first sign confirming the claims of President Donald Trump and the White House that China had agreed to start “immediately” buying U.S. products.” @business",

"Looking forward to being with the Bush family. This is not a funeral, this is a day of celebration for a great man who has led a long and distinguished life. He will be missed!",

"I am glad that my friend @EmmanuelMacron and the protestors in Paris have agreed with the conclusion I reached two years ago. The Paris Agreement is fatally flawed because it raises the price of energy for responsible countries while whitewashing some of the worst polluters in the world. I want clean air and clean water and have been making great strides in improving America's environment. But American taxpayers – and American workers – shouldn't pay to clean up others countries' pollution.",

"Hopefully OPEC will be keeping oil flows as is, not restricted. The World does not want to see, or need, higher oil prices!",

"We are either going to have a REAL DEAL with China, or no deal at all - at which point we will be charging major Tariffs against Chinese product being shipped into the United States. Ultimately, I believe, we will be making a deal - either now or into the future China does not want Tariffs!"

La obtención de estos resultados ha tardado un tiempo de 0,57 segundos. Las pruebas que se explican en los siguientes apartados requieren un tiempo mayor que el obtenido en este test. Esto se debe a que aquí no se divide el **`DataStream`** en un **`KeyedStream`**, lo que supone un menor tiempo de ejecución en cuanto a la generación de ventanas mediante el disparador propio desarrollado, el cual es más sencillo que los disparadores de Flink para generar ventanas temporales o por número de elementos. Este tiempo demuestra un gran potencial en el uso de fórmulas para generar ventanas.

6.4 Prueba de KeyedStreamTest

Para las pruebas de **`KeyedStreamTest`** se han usado dos conjuntos de datos distintos para las pruebas, diseñados de tal manera que cada uno represente distintos escenarios:

- Escenario nº1: ("Tom", 4), ("Tom", 8), ("Ana", 2), ("Tom", 2), ("Ana", 2), ("Ana", 2), ("Ana", 2), ("Ana", 2), ("Ana", 2), ("Ana", 2), ("Ana", 9)
- Escenario nº2: ("Tom", 4), ("Tom", 4), ("Ana", 2), ("Tom", 4), ("Tom", 5), ("Tom", 6), ("Tom", 5), ("Tom", 4), ("Tom", 4), ("Tom", 4), ("Tom", 6), ("Tom", 6), ("Tom", 6), ("Tom", 6), ("Tom", 6), ("Tom", 6), ("Ana", 2), ("Ana", 9)

En el primer escenario, Tom tiene un valor alto que equilibra más tarde con un valor bajo; sin embargo Ana tiene un valor bajo que no consigue equilibrar a tiempo con un valor alto, pero si equilibra el valor bajo en los siguientes seis instantes.

En el segundo escenario, Tom tiene valores normales durante doce mediciones, y Ana tiene un valor bajo que consigue equilibrar con un valor alto a tiempo.

Los resultados obtenidos son:

- Escenario nº1: (Ana,False), (Tom,True), (Ana, True).
- Escenario nº2: (Ana,True), (Tom,True)

Como conclusión, se puede ver que los resultados obtenidos son los correctos. En el primer escenario Ana no equilibra el valor, Tom sí, y Ana equilibra el segundo. En el segundo escenario, todos los valores de Tom son normales y Ana equilibra el valor.

Los tiempos de cada prueba son 2,84 segundos (el primer escenario) y 2,58 segundos (el segundo escenario). Aunque parezca un tiempo relativamente alto para los pocos datos que hay, la mayoría del tiempo de procesado se lo lleva la creación del **`DataStream`** y su paso a **`KeyedStream`**, no la aplicación de la fórmula. Con esto se quiere decir que, al aumentar los datos, el tiempo no aumenta de manera lineal con ellos. El cambio del tiempo de ejecución en función del número de datos se aprecia mejor en el apartado 6.6.

6.5. Prueba de StreamTest

El test de **`StreamTest`** también consiste en dos pruebas. En la primera disponemos de los siguientes datos:

"hola", "hola", "hola", "hola", "hola", "hola", "hola", "hola"

Tras dividir el *stream* para que cada *string* esté en una ventana, se comprueba que todas las ventanas contengan la cadena “hola”. Como se puede observar, esto cierto, y el resultado obtenido es el esperado, pues es cierto.

En segundo lugar, se prueba la aplicación de polución de Flink, comprobando que los primeros datos sean el valor 1, y los siguientes el valor 2. Sabemos que el generador comienza enviando datos que superan el umbral de polución y que la alarma empieza estando desactivada y necesita recibir un cierto número de datos antes de poder decidir si debe activarse o desactivarse. Por tanto, primero se recibe el valor 1, indicando que la alarma no debe cambiar su estado debido a que aun no se disponen de datos suficientes, y a continuación, al seguir llegando datos que superan polución, empieza a recibirse el valor 2, que indica que la alarma debe activarse. Como vemos, esto encaja con la condición de la fórmula, y el resultado obtenido, como es de esperar, es también cierto.

Se ha medido el tiempo de ejecución de la primera prueba, siendo este de 2,67 segundos. Como se puede ver es similar a los tiempos obtenidos en el apartado anterior a pesar de que la fórmula es mucho más sencilla y los datos introducidos también.

No se ha realizado ninguna medición de la segunda prueba debido a que el *stream* al que se aplica la fórmula se va generando poco a poco, a medida que va recibiendo datos de los sensores generadores, por lo que el tiempo depende totalmente de la cantidad de datos producidos y el ritmo al que van llegando, y el tiempo medido no sería significativo.

6.6 Análisis de rendimiento

En este apartado se miden los tiempos de ejecución de las pruebas alterando el número de ejecuciones de la función **`test`** y el número de datos de entrada. En general en las pruebas no es posible aumentar directamente el número de datos de entrada por los generadores usados, pero sí que es posible aumentarlos indirectamente. Por ejemplo, en el caso de los tests de carreras, al aumentar el número de carreras generadas, la distancia que se ha de recorrer para acabar la carrera o el número de corredores es muy probable que se aumenten los datos generados. Estas medidas se han tomado en un equipo con procesador Intel Core i7 y 12 GB de RAM.

A continuación se presentan los tiempos de ejecución, en segundos, en función de los parámetros que se han cambiado para las pruebas. Para cada combinación de parámetros se han medido cinco tiempos de ejecución, y en las tablas se presenta su media.

6.6.1 Polución

		Número ejecuciones		
		50	100	200
neverPol	Elementos por ventana = 1, ventanas = 10	4,70605607540	7,92689302600	23,00016575820
	Elementos por ventana = 20, ventanas = 20	4,65146081900	7,83306482440	21,60973431620
	Elementos por ventana = 100, ventanas = 20	4,76003234800	8,25745197720	22,85802845260
	Elementos por ventana = 100, ventanas = 50	5,62591344500	8,80141364840	32,27455423620

		Número ejecuciones		
		50	100	200
checkGenPol	Elementos por ventana = 1, ventanas = 10	4,57393687660	7,85968586340	21,48578490980
	Elementos por ventana = 20, ventanas = 20	4,63867691080	8,03844486640	19,80810863720
	Elementos por ventana = 100, ventanas = 20	4,96583337120	8,40072162100	21,58727348300
	Elementos por ventana = 100, ventanas = 50	5,04769140240	8,46844282240	24,97824949760

		Número ejecuciones		
		50	100	200
checkAlarm	Elementos por ventana = 1, ventanas = 10	4,69892376920	7,70837660460	20,55809696240
	Elementos por ventana = 20, ventanas = 20	4,84375066840	7,89378141640	23,03795751380
	Elementos por ventana = 100, ventanas = 20	5,40253923060	8,75110834760	21,43426977880
	Elementos por ventana = 100, ventanas = 50	5,39319784160	8,54213841780	23,17762298680

		Número ejecuciones		
		50	100	200
alarmOn	Elementos por ventana = 1, ventanas = 10	4,79226726920	7,82911442540	20,42655902960
	Elementos por ventana = 20, ventanas = 20	4,80951835200	8,08871159360	23,61456063400
	Elementos por ventana = 100, ventanas = 20	5,00297381140	8,16504785840	20,27800188680
	Elementos por ventana = 100, ventanas = 50	5,44537121540	8,73575074020	20,88570323660

6.6.2 Carreras

		Número ejecuciones		
		50	100	200
racesOk	Distancia = 50, número corredores = 5, número carreras = 1	4,96651898420	6,73322610920	17,11439512360
	Distancia = 100, número corredores = 6, número carreras = 1	4,80652452620	8,02630551700	21,14006934820
	Distancia = 100, número corredores = 6, número carreras = 3	4,99755912960	8,22858430680	18,73317975180
	Distancia = 100, número corredores= 10, número carreras = 10	5,52226608020	8,96099747440	26,52746291620

		Número ejecuciones		
		50	100	200
Check Speeds	Distancia = 50, número corredores = 5, número carreras = 1	4,82744221760	7,97217682920	21,7703772680
	Distancia = 100, número corredores = 6, número carreras = 1	4,66696485460	7,94391899220	22,51231306240
	Distancia = 100, número corredores = 6, número carreras = 3	3,97975089200	8,10975065260	22,48076717460
	Distancia = 100, número corredores= 8, número carreras = 8	5,16915443640	8,63523156940	25,34447981580

		Número ejecuciones		
		50	100	200
Check Banned Runner	Distancia = 1200, número corredores = 5, número carreras = 1	4,70474412280	7,94777291300	21,29641840860
	Distancia = 1200, número corredores = 6, número carreras = 1	4,96339036220	7,86126689540	20,43176484580
	Distancia = 1200, número corredores = 6, número carreras = 3	4,74512420320	7,81083158800	21,27845092180
	Distancia = 1200, número corredores= 8, número carreras = 8	4,81638320380	8,06474824520	21,54857228140

		Número ejecuciones		
		50	100	200
banned Never Win	Distancia = 50, número corredores = 5, número carreras = 1	3,92272633740	7,60331251200	20,09054553940
	Distancia = 100, número corredores = 6, número carreras = 1	4,55179789500	7,71702870300	20,07396587160
	Distancia = 100, número corredores = 6, número carreras = 3	4,75102213280	7,69765381180	19,43879614080
	Distancia = 100, número corredores= 8, número carreras = 8	4,64086580840	7,80944248660	21,61228748860

6.6.3 Valoración de los resultados

De los resultados anteriores se pueden realizar las siguientes observaciones:

- Las distintas pruebas tardan tiempos parecidos para los mismos parámetros, esto quiere decir que el tiempo de evaluación de las distintas fórmulas de lógica temporal es similar. El uso de una fórmula u otra no parece tener mucho peso en el tiempo de evaluación.
- Tanto en las pruebas de Polución como las de Carreras, el tiempo de ejecución es parecido en función de las iteraciones de la función. Esto indica que el formato que siguen los datos tampoco tiene gran influencia en el tiempo de ejecución.

- Al aumentar la distancia que se ha de recorrer en una carrera a veces el tiempo baja. Esto parece un error, pero es lógico, ya que los corredores necesitan más tiempo para acabar la carrera, y la creación de un corredor dopado se toma al generar la distancia que ha recorrido por cada instante. Esto quiere decir que al haber más instantes es más probable que los corredores se dopen y se les descalifique, por lo que la carrera acaba antes.
- Aumentar el número de corredores y el número de carreras no aumenta drásticamente el tiempo de proceso. Este aumento solo se nota cuando se aumenta también el número de ejecuciones.
- Si sumamos los tiempos de procesado en los casos peores por cada conjunto de pruebas (Polución y Carreras por separado), se tarda en torno a 2 minutos en ejecutar todas las pruebas presentadas. Parece un tiempo razonable dado que en estos casos peores, por cada prueba, hay que generar los datos y evaluar la función 200 veces.

Para concluir este apartado, merece la pena notar lo poco que escalan los tiempos de procesado en función de los datos introducidos. En las pruebas de polución se puede ver claramente que para el mismo número de iteraciones, aumentando de 10 a 5000 elementos, apenas se aumenta el tiempo de ejecución en un segundo. Esto muestra que la herramienta desarrollada es una herramienta ideal para ejecutar pruebas de aplicaciones de sistemas de computación distribuida, caracterizados por la gran cantidad de datos con los que trabajan. Si además se tiene en cuenta que en las pruebas que ejecutan 200 iteraciones, cada una con 5000 elementos, se tarda de media aproximadamente medio minuto, se puede observar que la herramienta es eficaz realizando también un gran número de pruebas. Esto hace que sea perfecta para probar un gran sistema en el que hay que realizar un gran número de tests con muchos datos.

Capítulo 7: Conclusiones y Trabajo Futuro

En este capítulo se evalúa el trabajo realizado y los objetivos conseguidos, incluyendo una valoración personal. También se incluyen posibles mejoras y ampliaciones que podrían desarrollarse en el futuro.

7.1 Cumplimiento de objetivos

Para cerrar esta memoria, se va a hacer una retrospectiva de los objetivos fijados al inicio de esta, indicando para cada uno de ellos en qué grado se ha cumplido:

- Profundización en la Lógica Lineal Temporal y en sus aplicaciones prácticas en *testing* de *software.* Se estudió tanto la Lógica Lineal Temporal y su extensión desarrollada para sistemas de *stream processing* (LTLss), como su aplicación en Sscheck. También se consiguió desarrollar generadores de datos y casos de prueba que usan LTLss de manera correcta, por lo que el objetivo se considera cumplido.
- Aprendizaje del lenguaje Scala y de sus características funcionales. Se realizó un curso de Scala y el proyecto ha sido implementado en este lenguaje, aprovechando características diferenciales de Scala como funciones de orden superior y funciones implícitas. También se ha hecho uso de bibliotecas propias de este lenguaje, como ScalaCheck, y se han podido reutilizar clases de otro proyecto escrito en Scala, concretamente Sscheck, por lo que este objetivo también se considera cumplido.
- Estudio de Flink y su funcionamiento. Se ha conseguido profundizar totalmente en Flink, llegando incluso a consultar el código fuente disponible en GitHub; se han utilizado gran parte de los tipos de *streams* y ventanas de Flink, entendiendo su funcionamiento, y se ha adaptado con éxito el proyecto Sscheck para su funcionamiento sobre Flink. Por tanto, se ha alcanzado este objetivo satisfactoriamente.
- Diseño y desarrollo de la solución para Flink:
  - Generadores. Aunque no se consiguió desarrollar generadores de ventanas, debido a que esto no es posible en Flink, se consiguió crear un método a partir del cual obtener ventanas con los datos obtenidos de los generadores.
  - Función `test`. El uso de la función `test` ha conseguido ser el mismo que el de la función original de Sscheck, aunque desde el punto de vista de la implementación son funciones muy distintas.
  - Casos de prueba. Se han realizado dos pruebas complejas que usan *random testing* con los generadores y la función `test` desarrollados en este proyecto, con satisfactorios resultados.
- Documentación del desarrollo y de lo aprendido. El proyecto se fue documentando

a medida que se iba avanzando, y esta documentación se ha aprovechado para la redacción de esta memoria.

7.2 Trabajo futuro

En este apartado se van a enumerar las posibles mejoras que se pueden aplicar al proyecto. Se van a presentar estas mejoras según la parte a la que afectan.

- Generadores:
  - Se puede refactorizar la clase separando los métodos que devuelven generadores de **`ListStream`** de los métodos que devuelven generadores de listas. Otra opción sería separar los generadores que cumplen las propiedades de LTLss de los generadores auxiliares.
  - Revisar el modo en que se dividen los datos por instantes, intentando que el tiempo manejado sea más real.
- Función **`test`**:
  - Cambiar la función para que la salida por pantalla no sea estrictamente usando transformaciones del *stream*, ya que para realizar estas transformaciones todos los *streams* deben guardarse en memoria, lo que limita el número de *tests* que pueden realizarse. Una alternativa sería usar los *sink* que ofrece Flink para guardar los resultados del tratamiento de cada *stream* en algún destino persistente y posteriormente acceder a él para mostrar los resultados. Otra opción más compleja es implementar un *sink* propio que sobrescriba la interacción de Flink con la salida por pantalla, de tal manera que este *sink* actúe como almacén de los datos y en base a ellos decida qué mostrar.
  - Ampliar la función para dar la opción de guardar los contraejemplos de un test.
  - Implementar *shrinking*, una técnica que minimiza los casos de test que producen errores para que resulten más sencillos de depurar
- Pruebas de concepto:
  - Idear nuevos problemas sobre los que desarrollar más pruebas.
  - Se ha observado que todos los casos de prueba tienen una estructura similar: definición de los generadores, definición de la fórmula, y llamada a la función **`test`**. Estos casos de prueba podrían definirse como clases que hereden de una clase abstracta la cual obligue a implementar métodos para la definición de los generadores, definición de la fórmula, y llamada a la función **`test`**. De este modo, quedaría más claro las partes a seguir a la hora de implementar nuevas pruebas.
- FormulaTrigger:
  - Se podrían implementar una versión de **`FormulaTrigger`** que se dispare según el método **`onEventTime`**, lo que permitiría definir un evento propio que ejecute el disparador. También se puede usar el *timestamp* con el que llegan los elementos al disparador para usarlo en la fórmula. Por último, también es

posible cambiar la condición en la que se ejecuta el disparador para que esto suceda cuando la fórmula se evalúa a falso, lo que permitiría localizar los casos de prueba que hacen falsa una fórmula.
    - Este último cambio supondría solo cambiar cuándo se ejecuta el disparador y no la parte de uso y consumo de la fórmula. Podría hacerse que la condición por la cual se ejecuta el disparador sea parametrizable, o incluso diseñar una jerarquía de herencia de disparadores que tengan valores por defecto para estos parámetros.
- StreamTest:
    - Terminar de adaptar la función **`test`** que recibe *streams* para poder realizar varios *tests* y que muestre los resultados como la función original, y definir una condición según la cual decidir cuándo devolver un resultado final (ya que un *stream* no tiene final).
- Mejoras generales:
    - Varias clases utilizan funciones de agregación que aplican una fórmula de LTLss a una serie de datos. Estas funciones se diferencian solo en pequeños detalles, por lo que podría realizar un estudio para encontrar elementos comunes y ver si es posible refactorizarlas mediante herencia o algún otro método.

7.3 Experiencia personal

Durante el desarrollo del proyecto me he encontrado con diversos retos y dificultades. Primero tuve que aprender un nuevo lenguaje de programación, Scala, y aprender a utilizar la biblioteca ScalaCheck y sus generadores. También tuve que familiarizarme con el proyecto Sscheck y entender su funcionamiento, tarea que me resultó complicada puesto que es un proyecto muy extenso y no estoy habituada a trabajar con un proyecto ajeno de tal magnitud. Por otro lado, se recomienda utilizar IntelliJ para trabajar con Flink, un entorno de desarrollo con el que no había trabajado nunca antes y que es bastante diferente a lo que estaba acostumbrada.

El modo en que Flink está implementado también ha añadido dificultad al proyecto. Por ejemplo, el hecho de que no existan ventanas como tal y la división del *stream* sea solo lógica complicó la implementación de los generadores. También se encontraron problemas a la hora de devolver los resultados debido a que la función **`aggregate`** obliga a devolver un valor por ventana y Flink no ofrece ninguna herramienta para extraer los datos del *stream*. Por último, el tratamiento que hace Flink de las excepciones añadió dificultad a la depuración del código, ya que Flink captura las excepciones internamente. Debido a esto, había momentos en los que algunas partes de código no se ejecutaban y Flink no daba ninguna clase de información de por qué o por dónde se encontraba el problema, por lo que había que ver y depurar el propio código de Flink para localizar la excepción.

Como conclusión final, puedo decir que me siento contenta con el desenlace de este

proyecto. He aprendido a utilizar nuevas tecnologías, he adquirido un conocimiento bastante profundo respecto a Flink, a pesar de haber encontrado ciertas dificultades he sido capaz de solventarlas, y al final he logrado completar el proyecto de forma satisfactoria. Incluso se me ocurrieron nuevas ideas durante el proyecto que me pareció divertido e interesante llevar a la práctica.

Chapter 8: Introduction

In this first chapter, the most important points on which the project is based are explained, as well as their motivation, in order to provide an overview of the project.

8.1 Project description

Today, due to the expansion of new technologies, huge amounts of data are generated at any given time. Data of all kinds, from which a great deal of information can be obtained, can be given numerous utilities, finding among them different uses such as calculating which products will sell best in a supermarket from the tastes and habits of customers, or predict an infection thanks to data obtained from patient monitoring [1]. There are many people who are aware of the great benefit that can be obtained by handling data in the correct way and also many, in addition, already make use of it. For example, Amazon makes use of the information it has about its customers both to improve its advertising algorithms and to improve customer service, and Netflix takes advantage of the data on the popularity of each film and series to decide what new content to add [2].

However, mass storage systems are not always able to cope with this large amount of data. In addition, there are situations where it is even necessary to be able to handle data constantly as it is generated, as is the case of monitoring patients to prevent infections. That is why systems are increasingly needed to allow and facilitate the processing of data flows (in this memory referred to as streams). These systems are called stream processing systems, capable of processing data in motion, that is, managing the data at the moment it is generated or received, and thus avoiding having to store it in order to make use of it [3].

Flink is one of these stream processing systems; it receives a data stream, offers tools to the user to facilitate the handling and transformation of the information received through this stream, the data is processed as it enters the system, and the result is returned in another output stream, which can be processed again if desired. A more complete way of how Flink works will be explained in more detail in chapter 3 of this memory.

In the field of computer science, software testing refers to any activity carried out in order to verify that the results obtained match the expected results, and therefore the software works correctly and without bugs. In addition, testing also helps finding implementation errors, possible situations that were not taken into account or even missing requirements [4].

No testing is able to guarantee 100% that a software is completely free of bugs, since testing actually helps to confirm the existence of bugs, not the absence of them, but without testing, these errors in the implementation would not be detected. Therefore, having solid testing methods for stream processing systems and thus gaining confidence that the data processing is correct and, consequently, the information obtained from it is as reliable as possible, is of great importance.

However, in general there are very few testing alternatives for stream processing systems, which are so useful and necessary today. There are basically only unit tests available, which is unfeasible in the cases for which it is most interesting to make use of this type of systems, since in such situations there is an enormous amount of data and the unit tests require defining a realistic input stream and its corresponding output stream.

For this reason the main idea of this project arises, which is to adapt the Sscheck tool, an application that allows random testing in the stream processing system Spark Streaming, in order to do random testing in the distributed environment of Flink, which handles a more real time than Spark.

8.2 State of the Art

Below are the projects that preceded this one, highlighting the features they share with the current project.

8.2.1 QuickCheck

QuickCheck is a testing tool for Haskell developed by Koen Claessen and John Hughes. The idea of this tool is, instead of defining a series of input data to check that the output is as expected, to define a list of properties that the function must fulfill. In other words, it follows a property-based testing technique instead of using unit tests [5].

To test these properties, the tool generates a series of random data, thus allowing a large number of test cases to test the software. The random data is obtained from the generators provided by QuickCheck, from which more complex ones can be made, adapted to the needs of each program. A generator is simply a function that receives several or no parameters and generates random data dependent on them. When the term 'generator' is used in this memory, it means the same as it means here.

To perform the testing, QuickCheck uses the random testing; it automatically generates as much data as desired from the generators, the functions to be tested receive this data as input, and the outputs are checked for compliance with the defined properties.

As a result, we have a large number of tests performed with different data each time, making it easier to find unforeseen errors very quickly. There are many programming languages to which this tool has been extended, Python (Hypothesis), Java (QuickTheories) and Scala (ScalaCheck), the latter being the chosen one for this project.

8.2.2 Sscheck

Sscheck [6] is a testing tool developed by Adrián Riesco and Juan Rodríguez-Hortalá that, making use of linear temporal logic and the ScalaCheck library, allows testing Spark Streaming programs with random testing techniques. Sscheck's motivation is the one already explained in the previous part of this memory; to offer a viable alternative to be able to do testing in stream processing systems.

In addition, Adrián Riesco and Juan Rodríguez-Hortalá devised a logic for the development of Sscheck that fits with the application of random testing to stream processing systems, due to the problems encountered when using Temporary Linear Logic. This logic, called Linear Temporal Logic for Streaming Systems (LTLss) and also used in the present project, extends LTL3 by adding timers (represented as natural numbers) to the temporary operators (except for the next operator) to define the number of instants (which in Spark Streaming does not correspond to time, but to batches) within which a property must be fulfilled. In chapter 3 of this memory the motivation behind the LTLss logic and how it works is further developed.

On the other hand, there is also a previous project carried out by Max Arnulfo Tello Ortiz, which extends the functionality of Sscheck in order to make it possible to load and save test cases and thus be able to reproduce the errors found thanks to the testing carried out with Sscheck, otherwise a difficult task because the generated data during the testing is random [7].

The project discussed in this report is based on the Sscheck tool, without the previous extension present, adapting its code [8] so that it works on Flink.

The rest of the memory is structured as follows:

- Chapter 2 sets all the project objectives and the work plan followed to achieve them.
- Chapter 3 deals with the preliminaries, where the main concepts handled in the project are developed in depth.
- Chapter 4 explains all the details about the implementation of the developed testing tool.
- Chapter 5 contains detailed information about implementations that use testing based in temporal logic techniques without applying random testing.
- The obtained results can be found in Chapter 6, accompanied by an explanation about why those are the results obtained.
- Finally, chapter 7 contains the conclusions derived from the completion of the project, as well as possible ideas and improvements that could be included in the future.

The application code is available at:

https://github.com/Valcev/TFM

Chapter 9: Conclusions and Future Work

This chapter evaluates the project implemented and the objectives achieved, including a personal assessment. It also includes possible improvements and extensions that could be developed in the future.

9.1 Objectives met

To close this report, there is a retrospective of the objectives set at the beginning of the report, specifying for each of them the degree to which they have been fulfilled:

- In-depth study of Linear Temporal Logic and its practical applications in software testing. A study of Linear Temporal Logic, its extension developed for stream processing systems (LTLss), and its application in Sscheck, was made. It was also possible to develop data generators and test cases that use LTLss correctly, so the objective is considered fulfilled.
- Learning the Scala language and its functional characteristics. A Scala course was carried out and the project has been implemented in this language, taking advantage of differential Scala features such as higher order functions and implicit functions. There is also use of libraries specific to this language, such as ScalaCheck, and it was also possible to reuse classes from another project written in Scala, specifically Sscheck, so this objective can also be considered fulfilled.
- Study of Flink and the way it works. It has been possible to reaserch deeply into Flink, even consulting the source code available in GitHub; a large part of the types of streams and windows of Flink have been used, understanding the way they work, and the Sscheck project has been successfully adapted to operate on Flink. Therefore, this objective has been achieved satisfactorily.
- Design and development of the solution for Flink:
  - Generators. Although it was not possible to develop window generators, due to the fact that this is not possible in Flink, it was possible to create a method from which to obtain windows containing the data obtained from the generators.
  - **`Test`** function. The use of the **`test`** function has managed to be the same as that of the original Sscheck function, although from the implementation point of view they are very different functions.
  - Test cases. Two complex tests have been carried out using random testing with the generators and **`test`** function developed for this project, with satisfactory results.
- Reporting of the project development and the learning process. The project was documented as it progressed, and this documentation has been used to write this report.

9.2 Future work

This section will list the possible improvements that can be applied to the project. These improvements will be presented according to the project part they affect.

- Generators:
  - The class can be refactored by separating the methods that return ListStream generators from the methods that return list generators. Another option would be to separate the generators that meet the LTLss properties from the auxiliary generators.
  - Revisit the way in which data is divided by instants, trying to make the managed time more real.
- **`Test`** function:
  - Change the function so that the screen output is not strictly processed using stream transformations, since to perform these transformations all streams must be stored in memory, which limits the number of tests that can be performed. An alternative would be to use the sink offered by Flink to save the results of the treatment of each stream in some persistent destination and then access it to show the results. Another more complex option is to implement a custom sink that overwrites the interaction of Flink with the screen output, in such a way that this sink acts as a data store and decides on the basis of this data what to show.
  - Expand the functionality of the **`test`** function to give the option to save the counterexamples of a test.
  - Implement shrinking, a technique that minimizes error-producing test cases to make them easier to debug.
- Proofs of concept:
  - Think about new problems on which to develop more tests.
  - It has been observed that all test cases have a similar structure: definition of the generators, definition of the formula, and a call of the **`test`** function. These test cases could be defined as classes that inherit from an abstract class which forces to implement some necesasary methods for the definition of the generators, definition of the formula, and call to the **`test`** function. In this way, it would be clearer the structure to follow when implementing new tests.
- FormulaTrigger:
  - A version of FormulaTrigger could be implemented that triggers according to the onEventTime method, which would allow to define custom events that triggers the trigger. It is also possible to use the timestamp with which the elements reach the trigger to use it in the formula. Finally, it is also possible to change the condition in which the trigger is executed so that this happens when the formula is evaluated as false, which would allow locating the test cases that make a formula false.
  - This last change would only involve changing when the trigger is executed and

not the use and consumption part of the formula. It is also possible to make the condition by which the trigger is executed parameterizable, or even design a hierarchy of inheritance of triggers that have default values for these parameters.

- StreamTest:
  - Finish adapting the `test` function that receives streams to be able to perform several tests and show the results as the original function, and define a condition that decides when to return a final result (since a stream has no end).
- General improvements:
  - Several classes use aggregation functions that apply a LTLss formula to a series of data. These functions differ only in small details, so a study could be conducted to find common elements and see if it is possible to refactor them by inheritance or some other method.

9.3 Personal experience

During the development of the project I encountered several challenges and difficulties. First I had to learn a new programming language, Scala, and learn how to use the ScalaCheck library and its generators. I also had to familiarize myself with the Sscheck project and understand how it works, which was a complicated task for me as it is a very large project and I am not used to working with projects of such magnitude. On the other hand, it's recommended to use IntelliJ to work with Flink, a development environment tool that I've never worked with before and that is quite different from what I was used to.

The way Flink is implemented has also added difficulty to the project. For example, the fact that there are no windows as such and the division of the stream is only logical complicated the implementation of the generators. There were also problems to return the results because the aggregate function forces to return one value per window and Flink does not offer any tool to extract the stream data. Finally, Flink's treatment of exceptions added difficulty to the debugging of the code, as Flink captures exceptions internally. Because of this, there were times when some parts of the code didn not run and Flink did not give any kind of information as to why or where the problem was, so there was the need to check and debug Flink's own code to locate the exception.

As a final conclusion, I can say that I am quite happy with the outcome of this project. I've learned to use new technologies, I've acquired a pretty deep knowledge about Flink, despite having encountered certain difficulties I've been able to solve them, and in the end I've managed to complete the project satisfactorily. I even came up with new ideas during the project that I found fun and interesting to put into practice.

Bibliografía

[1] Ladrero, Iñaki: “10 ejemplos de aplicación de Big Data” (2017). Enlace: https://www.baoss.es/10-ejemplos-usos-reales-big-data/

[2] Portilla, Daniel: “Cómo las compañías más exitosas usan Big Data” (2017). Enlace: https://blog.centricodigital.com/big-data-companias-exitosas/

[3] "What is stream processing?" Enlace: https://www.data-artisans.com/what-is-stream-processing

[4] "What is Software Testing? Introduction, Definition, Basics & Types." Enlace: https://www.guru99.com/software-testing-introduction-importance.html

[5] Crettenand, Gilles: Functional PHP. Packt Publishing, 2017.

[6] A. Riesco y J. Rodríguez-Hortalá.Property-based testing for Spark Streaming.Theory and Practice of Logic Programming. Cambridge University Press, 2019.

[7] Tello Ortiz, Max Arnulfo: "Generadores ScalaCheck para property-based testing de programas Spark y Spark Streaming." (2016). Enlace: https://eprints.ucm.es/38484/

[8] GitHub Sscheck. Enlace: https://github.com/juanrh/sscheck

[9] Curso Scala. "Functional Programming Principles in Scala." Enlace: https://www.coursera.org/learn/progfun1?specialization=scala

[10] Curso Github. "Gestión de proyectos Software con Git y GitHub." Enlace: https://miriadax.net/web/gitmooc

[11] "Coverage-Based testing." Enlace: https://www.slideshare.net/ScioMx/coverage-based-testing

[12] Apuntes de la asignatura ACFI

[13] Bauer, A., Leucker, M., and Schallhart, C: Monitoring of real-time properties. InFSTTCS 2006: Foundations of Software Technology and Theoretical Computer Science,pages 260–272. Springer, 2006.

[14] Apache Spark (2014). Enlace: http://spark.apache.org/

[15] Apache Spark Streaming. Enlace: https://spark.apache.org/streaming/

[16] “Spark Streamming Programming Guide.”
Enlace :https://spark.apache.org/docs/2.2.0/streaming-programming-guide.html

[17] Biblioteca Scalaz. Enlace: https://scalaz.github.io

[18] Tanmay Deshpande: Learning Apache Flink. Packt Publishing, 2017.

[19] Apache Flink. "Windows." Enlace: https://ci.apache.org/projects/flink/flink-docs-stable/dev/stream/operators/windows.html

[20] Apache Flink. "Flink DataStream API Programming Guide". Enlace: https://ci.apache.org/projects/flink/flink-docs-release-1.7/dev/datastream_api.html

[21] Apache Flink. "DataStreams transformations". Enlace: https://ci.apache.org/projects/flink/flink-docs-stable/dev/stream/operators/

[22] Martin Odersky; Lex Spoon; Bill Venners: Programming in Scala, Third Edition. Artima Press, 2016.

[23] ScalaCheck. Enlace: https://www.scalacheck.org/index.html

[24] Specs2. Enlace: http://etorreborre.github.io/specs2/

[25] Scalacheck, licencia BSD 3-Clause. Enlace: https://github.com/rickynils/scalacheck/blob/master/LICENSE

[26] Specs2, licencia MIT. Enlace: https://github.com/etorreborre/specs2/blob/master/LICENSE.txt

[27] Apache License 2.0. Enlace: https://github.com/apache/flink/blob/master/LICENSE

[28] Scalaz licencia FreeBSD. Enlace: https://github.com/scalaz/scalaz/blob/series/7.3.x/LICENSE.txt

[29] Sscheck licencia Apache 2.0. Enlace: https://github.com/juanrh/sscheck/blob/master/LICENSE

[30] SBT licencia Apache 2.0. Enlace: https://github.com/sbt/sbt/blob/develop/LICENSE

Anexo I

Manual de usuario

Requisitos:

Para la ejecución del programa es necesario tener instalada una versión de Java 1.8 o superior.

Instrucciones

El ejecutable consiste en un archivo java ejecutable (extensión *jar*) que ha de ejecutarse desde la línea de comandos. Los argumentos admitidos son los siguientes:

Argumento	Explicación
-t <demo> | --test <demo>	Ejecuta una de las posibles pruebas según se indique en <demo>. Los posibles valores son: • Pollution : ejecuta los tests de *random testing* de polución • Race : ejecuta los tests de *random testing* de carreras y corredores. • KeyedStreamTest : ejecuta la prueba de la clase **`KeyedStream`**. • FormulaTrigger : ejecuta la concatenación de tuits de la clase **`FormulaTrigger`**. • PollutionFlinkApp : ejecuta la primera aplicación de Flink de alarmas por polución. Para detener su ejecución es necesario utilizar 'Ctrl + C'. • StreamTest : ejecuta la función **`test`** que funciona con *streams* con los datos de polución generados con la primera aplicación de Flink de alarmas por polución. Para detener su ejecución es necesario utilizar 'Ctrl + C'.
-h | --help	Muestra por consola los argumentos posibles

Como ejemplo, las posibles ejecuciones serían:

- java -jar tfm.jar -h
- java -jar tfm.jar -t Pollution
- java -jar tfm.jar -t Race
- java -jar tfm.jar -t KeyedStreamTest
- java -jar tfm.jar -t FormulaTrigger
- java -jar tfm.jar -t PollutionFlinkApp
- java -jar tfm.jar -t StreamTest