Institute of Parallel and Distributed Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Masterthesis

A Canonical Language for Complex Event Processing Systems

Saeed Mahmoodi

Course of Study: INFOTECH M.Sc.

Examiner: Prof. Dr. Bernhard Mitschang

Supervisor: Dr. rer. nat. Pascal Hirmer,
Ana Cristina Franco da Silva, M.Sc.

Commenced: May 6, 2018

Completed: November 6, 2018

Abstract

Complex Event Processing (CEP) systems have widely grown in recent years, as working efficiently with streaming data is getting a lot of attention. Many CEP query languages have been developed in order to realize and make use of CEP systems, each having a specific syntax, but producing the same output and growing every day. Creating a CEP system with a query language that supports them all is not feasible. To cope with this problem, in this thesis, a canonical query language is created in order to provide an abstraction layer of specific common CEP features for different languages. As a result, queries in the canonical language translated in each language, are then run on the corresponding engine separately.

Contents

List of Figures

List of Tables

List of Listings

1 Motivation and Introduction

Complex Event Processing (CEP) is the set of methods used to analyze complex events and derive meaningful information from them[8]. These events are complex combinations of incoming streaming data and in recent years, CEP has been widely used and incorporated over a variety of academic and industrial projects[9]. These projects aim to provide a solution to the problem of receiving, understanding, and handling complex events, and detecting various forms of combinations among them[8]. For example, considering the temperature in a chemical plant, one needs to trigger actions as soon as the temperature measured by specific sensors exceed a certain threshold to avoid critical danger or fire. In order to achieve the monitoring of environments using CEP, many event processing languages have been developed. These languages define, in form of CEP queries, how incoming streaming data is being processed by CEP engines.

Although nowadays, there is a large number of CEP systems, they each use their own query language for event processing. These languages differ in their syntax and their powerfulness. Consequently, once deciding to use a specific CEP system, changing it in the future leads to high effort due to the need to replace all CEP queries using another CEP query language. Hence, a vendor lock-in is a great risk when employing a CEP system. In order to cope with this issue, a standardized CEP query language is desired, similar to SQL for database systems. However, there are several issues in creating such a standardized language. Over specific stream processing use cases, some CEP languages are weaker or stronger. Moreover, there might be features that are not supported by one language at all. This is a big problem because, in practice, one might encounter multiple cases where a certain function is needed. In those cases, choosing the most efficient language is not easy because one might miss some features that are not included in a particular engine. Furthermore, existing well-established legacy systems cannot exchange their CEP language. Consequently, in this thesis, a canonical CEP language is proposed that can be transformed into existing languages so that a common language for CEP systems can be enabled without any required changes in existing legacy systems. In this way, a single query is created and outputs of different engines are expected. The powerfulness of the canonical language is limited by the weakest language.

In order to achieve a canonical language, an analysis of different CEP systems and corresponding languages is conducted. In addition, most common characteristics of CEP query constructs (stream definition, window, pattern, correlation, etc) are analyzed. By exploring the set of common characteristics of CEP languages, the canonical language is designed and translators for a set of CEP systems are developed.

The remainder of this master thesis is structured as follows: Chapter 2 gives an overview of the foundations of this work. Chapter 3 presents the analysis of CEP engines, CEP characteristics, and the Canonical language. Chapter 4 includes the implementation details and the unit tests. Chapter 5 discusses related work and, finally, Chapter 6 concludes this work.

2 Foundations

In this chapter, fundamental definitions around Complex Event Processing systems are presented. Nowadays, events are playing a major role in real-life situations and problems. An event in the scope of this thesis is a data tuple received by a system, that triggers a condition or a specific series of conditions that may lead to an action. Those specific conditions exist in many real-world working domains. For example, in case of a chemical plant, there might be a large number of sensors installed on the field level and from those, in milliseconds of time, data is being received in form of incoming events. Event Processing is the set of operations any system does over those incoming events. In other words, in order to understand and make use of events, we need special systems who are able to receive information and make it understandable for the user. An installed system on the field level for a local worker inside a chemical plant has the functionality to process those incoming event streams and then present it to the user.

Furthermore, in terms of the real-time behavior of systems, it is also important not only to recognize those events but also to work around their different occurrences. In other words, detecting the behavior of incoming events timely and being able to dynamically trigger actions for a specific series of events are needed as well. As events might also affect each other depending on the sources they are coming from, the processing of events on a large scale might get more complicated. In the example of a chemical plant, events coming from different processes are collected and observed at real-time. Many times at the process control level, it is required that based on values coming from different sensors a certain action is triggered.

2.1 Complex Event Processing (CEP)

Complex Event Processing is a set of processes on incoming events which derive a deeper understanding of those, who might have a much more complex relationship with each other[8]. In other words, complex event processing aims at detecting complex relationships between events or specific conditions where complex events are occurring; to detect them and provide means to create rules for each of them. Many systems include a series of complex events and it is, therefore, required for any CEP system to create those complex rules, monitor all events and detect those conditions when happening in real-time.

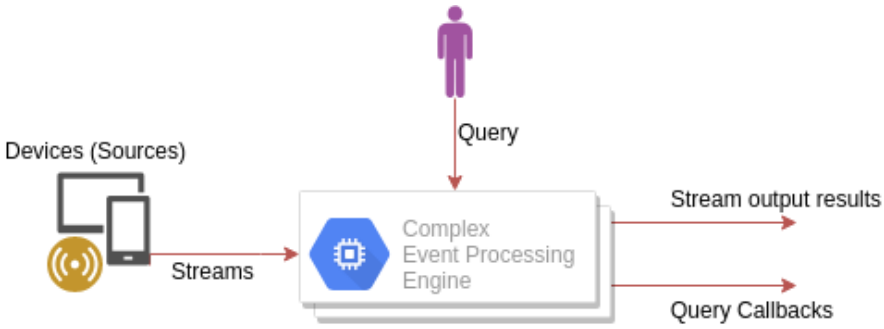

Figure 2.1: An overview of Complex Event Processing Systems

Figure 2.1 shows an overview of such systems. All streaming data are sent at different time frames and the engine receives them and produces more complex streams, and possibly function callbacks. The user works with the engine to define rules and queries, and create those callback functions and read from output streams. In this chapter, more details about CEP systems and their characteristics are discussed with a list of four stream processing languages and a quick overview of each of them.

2.2 Important Characteristics of CEP Languages

In this section, important characteristics of any Complex Event Processing system, such as windowing, correlation (join), filtering, detecting patterns, and different ranking operations are introduced and explained briefly. Each of these topics are very handy in working with complex event streams in different cases and combined together, they provide a good means of measurement and evaluation to the user. This section is mostly based on [3], [10] and [9].

2.2.1 Windows

Windows are intervals defined on streams. The very first important issue about windows is the policy on which the window is created on the stream. Some windows use time as a policy, moving over a stream for a specific period of time. For example, a window might be defined on a sensor stream input, with a time interval of 3 seconds. Another window policy is the count of the items on the stream. For example, a window might only consist of the last recent 10 received events from the stream. While the policy of creating a window over a stream plays a very important role in terms of how the user defines the stream, another important subject is how to make use of created windows. A Complex Event Processing engine usually is able to direct the output of each window to a new stream where many functions are created and developed using the window stream. For example, one might use each element of the window stream for a particular aggregation purpose or set a specific action at the beginning/end of a window.

2.2.2 Correlation

Every stream typically includes data tuples which are sent a receiver on a timed basis. A complex event processing system should be able to correlate two or more streams together based on a specific data field and then, change the output streams accordingly. In other words, when streams are correlated or joined in a complex event processing system, the CEP engine looks for the cases on a timed basis, where a certain condition between two data values from those streams matches. Then, the output stream would be the requested parameters from any of those streams on moments when the condition between those streams is met. For example, if a temperature sensor is installed on a couple of machines in a plant having the same process, and data are received from each sensor separately, one might be interested in times when there is a specific correlation between the temperature value coming from all sensors, and in cases of abnormalities, triggers actions. But in these cases, usually, the user is not interested in data coming from only one sensor, but rather only wants the action to be triggered at times when there is a special relation matching between data from all sensors.

2.2.3 Filtering

Filtering is another typical requirement in working with streams and in case of Complex Event Processing systems, any user is able to filter a stream or the output of streams together on a condition based on a specific value. In simple words, the output streams are only working and dealing with a filtered version of input streams which are based on a specific value. In the example of temperature sensors, one might only be interested in values over a certain threshold value at times, which means any value under that threshold is not processed.

2.2.4 Pattern Detection

One of the other important characteristics of Complex Event Processing systems is to be able to detect patterns amongst incoming stream data. In other words, when working with multiple streams sending data at different timelines, one might be interested in cases where a special pattern of events happens. Complex Event Processing languages provide the means to realize those patterns and detect them in different ways, but as a general ability, they are all able to define and detect specific patterns between events and perform different operations accordingly.

Typically what happens in CEP systems in dealing with patterns and streams, is that the user defines a starting condition on one of the input streams and then continues to extend the pattern with more conditions happening thereafter. Therefore, in defining patterns, the sequence in which the streams are defined and put into it is important. Patterns are matched according to that sequence which means it will only check the condition for the

second criteria in the sequence, if the first condition has met before that. For example, in creating a pattern between two temperature sensors, first sensor having a value under 30 degrees, the pattern will only check the second condition on times where the temperature coming from the first sensor is under 30 degrees.

2.2.5 Ranking and Aggregation Operations

Another set of operations which Complex Event Processing systems have, is to perform aggregation or rankings on streams. For example, one might want to sort the values in a specific input stream over a duration of time and send the average value of them to the output stream. One might be interested in checking only the first element of a window, or the last element of it. These typical operations are customizable for many CEP languages and the user can deal with different cases better and provide better solutions.

3 Canonical Language for Complex Event Processing Systems

In order to work more efficiently with existing complex event processing systems and languages, there is a need for a standardization among them. Since there is no common standard defined and each of those has been really expanding and developing on its own in recent years, creating a standardization for complex event processing query languages seems not to be reasonable and effective anymore.

The work of this thesis is to create an abstraction level, which starts with a list of stream processing characteristics and consequently creating an abstraction level. In this way, a means is created to write complex event processing queries which are translatable to all of those languages. Before the abstraction level itself, an important fact is the set of complex event processing characteristics which are included in it. As discussed in the previous chapter, there are many different operations in which those stream processing engines perform. Some of them are supported by all of the languages and some that are not. In order to create such abstraction level, a list of all those characteristics are presented. The abstraction level for all of the complex event processing languages needs to consider the weakest language among them because if an operation or a feature is not supported by any of the languages it is not worth including. While adding more stream processing languages to the canonical model, makes it more comprehensive, on the other hand, since not all of the languages have the same power and range of operations, a lot of complex event processing features need to be excluded.

Section 3.1 presents four complex event processing engines (Esper, Siddhi, Flink, Odysseus) and examples of their language syntax. In Section 3.2, a list of six basis CEP characteristics are discussed in more details. Those include defining input streams, joining streams, filtering streams, creating time and element windows, performing operations on windows, and pattern recognition. Section 3.3 introduces the canonical language grammar created to realize the abstraction layer above those four languages with a comprehensive description of how different grammar elements work. Finally, Section 3.4 describes the translation architecture.

3.1 CEP Engines

In this chapter, four examples of existing Complex Event Processing engines are introduced and briefly described. These include ESPER, WSO2 Siddhi, Odysseus, and Apache Flink. In terms of working with Complex Event Processing systems, choosing which engine to use is always a problem. All of the mentioned engines provide the means to work with complex event streams. Sections 2.3.1 to 2.3.5 will briefly describe each of them with a summary of their details.

3.1.1 Esper

Probably one of the most used engines and a powerful tool to work with Complex Event Processing systems is Esper[1]. Esper provides a SQL-like Query language which is very user-friendly and works perfectly in implementing the mentioned CEP Characteristics and in giving the user the maximum functionalities needed to work with streams and complex cases.

Esper is also enhanced in terms of performance measures which makes it one of the top choices in selecting a CEP engine to work with. There are many enhanced features provided by the Esper engine on different platforms which makes it easy for the end user to interact with and since the queries look much like database SQL queries, it is a well known complex event processing engine. Table 3.1 shows examples of Esper.

Table 3.1: Esper code examples.

topic	example
stream definition	`create schema InnerData (value string);`
filters	`select * from Withdrawal(amount >= 200);`
aggregation functions	`select count(*), sum(amount)` `from Withdrawal(amount >= 200);`
windows	`select * from Withdrawal#length(5);`
patterns	`StockTickEvent(symbol=ÏBM", price>80)` `where timer:within(60 seconds);`
sorting window	`select sum(price) from StockTickEvent#sort(10, price desc);`

[1] http://www.espertech.com/esper/

3.1.2 WSO2 Siddhi

Siddhi[2] is an open source Complex Event Processing engine which also has a SQL-like syntax. It also provides a set of different functions including defining different streams and windows for example and then performing different stream operations on them. In terms of performance, just like Esper, Siddhi provides fast stream processing and features many different stream data functions to the user. Table 3.2 shows some Siddhi examples.

Table 3.2: Siddhi code examples.

topic	example
stream definition	`define stream TempStream(deviceID long,roomNo int,temp double);`
filters	`from TempStream[(roomNo >= 100 and roomNo < 210) and temp > 40]` `select roomNo, temp insert into HighTempStream;`
windows	`from TempStream#window.length(10) select max(temp) as maxTemp` `insert into MaxTempStream;`

patterns	`from every( e1=TempStream ) -> e2=TempStream[ e1.roomNo == roomNo`
	`and (e1.temp + 5) <= temp ] within 10 min`
	`select e1.roomNo, e1.temp as initialTemp insert into AlertStream;`

stream correlation	`define stream TempStream(deviceID long,roomNo int,temp double);`
	`define stream RegulatorStream(deviceID long,roomNo int,isOn bool);`
	`from TempStream[temp > 30.0]#window.time(1 min) as T`
	`join RegulatorStream[isOn == false]#window.length(1) as R`
	`on T.roomNo == R.roomNo`
	`select T.roomNo insert into RegulatorActionStream;`

Siddhi provides a large range of stream operations including creating different streams, defining patterns across streams, performing different aggregation operations on stream data, working with data and time windows across them, and many others. Table 3.2 shows that Siddhi ends with putting all the matching data into an output stream which should also be defined to the engine before runtime.

3.1.3 Odysseus

Odysseus[3] is a standalone product which provides an integrated environment to work with continuous streams. In terms of its implementation, although it is mainly created based on Java, Odysseus provides an integrated development environment of its own called

[2] https://wso2.github.io/siddhi/
[3] https://odysseus.informatik.uni-oldenburg.de/

Odysseus Studio which provides a user interface for working with data stream operations and an application server that includes most of the core operations about the user and query management.

Moreover, Odysseus also uses CQL[4] and PQL[5] and it has rather a different style in defining streams and performing operations on them. Odysseus defines streams as sources and outputs as sinks and provides a list of operations in order to work with them.[6] Table 3.3 shows some examples of Odysseus in its procedural query language.

Table 3.3: Odysseus code examples.

topic	example

stream definition	`nexmark:auction:= ACCESS(source=‘nexmark:auction,`
	` wrapper=‘GenericPush‘,transport=‘TCPClient‘,`
	` protocol=‘SizeByteBuffer‘, dataHandler=‘Tuple‘,`
	`options=[[‘host‘,‘localhost‘],`
	` [‘port‘,‘65440‘],`
	` [‘ByteOrder‘,‘LittleEndian‘]],`
	`schema=[[’id’,’INTEGER’], [’name’,’STRING’]]`
	`});`

filters	`out = SELECT(predicate=’seller=1’,` `nexmark:auction);`
aggregation functions	`out=AGGREGATE(` `aggregations=[[’AVG’,’price’,’AVGPrice’]], nexmark:bid);`
stream correlation	`out = JOIN(PREDICATE=’nexmark:person.id=nexmark:bid.bidder’,` `nexmark:person,nexmark:bid);`
windows	`#PARSER PQL #ADDQUERY` `out= ELEMENTWINDOW(SIZE=10, nexmark:bid);`

patterns	`#PARSER PQL #ADDQUERY`
	`SASE(query = ’PATTERN SEQ(person p, bid b)`
	` WHERE skip_till_next_match(p,b)`
	` p.id = b.bidder, b.price > 200`
	` RETURN p.id, p.name, b.price’,`
	`schema=[[’id’,’Integer’], [’name’,’String’],`
	` [’price’,’double’]], type=’PersonEvent1’,`
	`person, bid);`

[4]Continuous Query Language
[5]Procedural Query Language
[6]https://wiki.odysseus.informatik.uni-oldenburg.de

3.1.4 Apache Flink

This section is mostly based on Apache Flink's DataStream API and Complex Event Processing library on Flink[7]. Apache Flink is a Java-based data flow and stream processing system that provides a set of Java objects and methods in order to work with event data. It provides a large set of methods and objects in DataStream API in order to create and work with continuous data streams and in case of complex event processing, it supports most of the characteristics mentioned before by enabling the programmer to define stream objects, windowed streams, pattern objects, etc. Because Flink is purely written in Java code, it is also possible to create methods and write user-defined operations. For example, Flink provides a filter method which can be applied to any stream, and one can define a filter function as an argument to it, which might have its advantages in comparison to other CEP languages.

Flink also provides methods to read the input data streams or write to output data stream in different formats and transform data streams or add keys to them for further operations. Its pattern library provides a series of methods to create a pattern object and then extend it with different methods which maintain the sequence of the pattern itself. Table 3.4 shows some examples of Flink's API code. Pattern and CEP libraries are included separately in FlinkCEP[8].

Table 3.4: Flink code examples.

topic	example

stream definition	`StreamExecutionEnvironment env =`
	` StreamExecutionEnvironment.getExecutionEnvironment();`
	`DataStream<Tuple2<String, Integer>> dataStream = env`
	` .socketTextStream("localhost", 9999);`

filters	`public boolean filter(Long value) throws Exception` ` {return (value <= 0);}`
windows	`dataStream.timeWindow(Time.minutes(5))` ` .process(new MyProcessWindowFunction());`

stream correlation	`dataStream.join(anotherStream)`
	` .where(<KeySelectorObject>)`
	` .equalTo(<KeySelectorObject>)`
	` .apply(<UserDefinedJoinFunction>);`

[7] https://flink.apache.org/

[8] more information on FlinkCEP: https://ci.apache.org/projects/flink/flink-docs-stable/dev/libs/cep.html

patterns	`DataStream<Event> input = ...`
	`Pattern<Event,?> pattern = Pattern.<Event>begin('start')`
	`.where(new SimpleCondition<Event>(){...})`
	`.followedBy('end')`
	`.where(new SimpleCondition<Event>(){...});`

3.1.5 Summary of Engines

In this chapter, four of the mostly used Complex Event Processing engines are introduced. It can clearly be seen from all of them that they provide the means to work on stream data processing. Esper and Siddhi have the advantage of having a SQL-like language syntax which makes them easier to work with, as they looks very similar to the typical database querying language, and therefore, they need less programming effort than Odysseus and Flink.

Esper, Siddhi, and Flink need core Java with a set of dependencies which should be provided in order to implement their engines, while Odysseus needs core Java installed on the system and Odysseus server and studio so that users can work with it. Table 3.5 shows a summary of CEP Characteristics and more implementation details of the engines discussed in this chapter.

Table 3.5: Summary of complex event processing engines.

name	input data	platform	query language	pattern	window	join	absence of events
Esper	tuple objects (stream data is sent via data tuple class objects)	Java, C#	SQL-Like (EPL[9])	Yes (keyword 'every')	Yes (time,length)	Yes	Yes
Siddhi	tuple objects (stream data is sent via data tuple class objects)	Java	SQL-Like (SiddhiQL[10])	Yes (keyword 'every')	Yes (time,length)	Yes (pairwise[11])	No
Odysseus	source definition (in studio)	Java	CQL, PQL	Yes (match and notify)	Yes (time,length)	Yes	Yes
Flink	stream objects	Java	Java, Scala	Pattern API (FlinkCEP)	DataStream[12]	Yes	No

[9] http://esper.espertech.com/release-5.2.0/esper-reference/html/epl_clauses.html
[10] https://wso2.github.io/siddhi/api/latest/
[11] https://wso2.github.io/siddhi/documentation/siddhi-4.0/#join-stream
[12] https://ci.apache.org/projects/flink/flink-docs-release-1.6/dev/datastream_api.html

3.2 Overview of Common Characteristics

In this section, a series of basic characteristics of stream processing are discussed. The general goal of this section is to provide more details of the CEP engines and discuss a list of six basis features of them and how those features are defined within each language in order to have a deeper understanding of each feature inside those languages.

Each subsection includes a detailed example and descriptions about each feature in each of the proposed languages, and a short conclusion about a possible abstraction for that feature.

3.2.1 Definition of Streams

One of the basic and initial operations needed to start working with event processing is creating streams. Table 3.6 provides an overview of stream creation in Esper, Siddhi, Odysseus, and Flink.

Table 3.6: Definition of Streams examples

language	example
Esper	Initially while setting the Esper engine up, Esper requires to have a configuration object sent to the Esper service provider instance. The streams should be defined and added in that configuration in form of pairs with their names and data types. In addition, Esper allows to define a **schema** inside the query. Following example creates a new stream as 'SecurityEvent'. Each event in this stream has an ipAddress and a useId value. Example: `create schema SecurityEvent as (ipAddress string, userId String);`
Siddhi	Siddhi has a more straightforward approach in defining streams; while is it also needed to define the input and output streams in the configuration while setting up the engine itself, the keyword **define stream** must be used at the beginning of each query to use the stream throughout the whole query. Following example is the same query from Esper in the previous part written in SiddhiQL. Example: `define stream SecurityEvent (ipAddress string, userId string);`

Odysseus	In Odysseus source objects need to be created by defining them in Odysseus studio. Thereafter, you can use those variables in further instructions and queries. In Odysseus' procedural query language, there is a rather different syntax and a higher amount of code to create a stream source. The source object needs specific values for server and data transfer values along with schema definition at creation. Therefore, it has higher amount of code to create a source comparing to Esper and Siddhi. Example:
	`nexmark:person:= ACCESS(source='nexmark:person',`
	`wrapper='GenericPush',`
	`transport='TCPClient',`
	`protocol='SizeByteBuffer',`
	`dataHandler='Tuple',`
	`options=[`
	`['host', 'localhost'],`
	`['port', '65440'],`
	`['ByteOrder', 'LittleEndian']],`
	`schema=[`
	`['timestamp', 'STARTTIMESTAMP'],`
	`['id', 'INTEGER'],`
	`['name', 'STRING']]);`

Flink	Flink receives stream sources in form of files, sockets, and collections from the environement object and creates a DataStream object. Example:
	`StreamExecutionEnvironment env =`
	`StreamExecutionEnvironment.getExecutionEnvironment();`
	`DataStream<Tuple3<String, Integer, String>> dataStream = env`
	`.socketTextStream("localhost", 9999); // or`
	`.fromCollection(Collection); // or`
	`.fromElements(<T>);`

In Esper and Siddhi, the stream definition can be included inside the query itself. Esper creates schemas and defined streams can be also included in siddhi. We will use these in the abstraction for stream creation in those languages. For Odysseus, it is more complicated as the instructions which need to be run are sufficient to be included in the abstraction, but the need to run them first on the Odysseus server to create the source objects might cause some problems. As for this stage, we will take the stream creation instructions from Odysseus and Flink as well (Although their implementation details might be different).

3.2.2 Correlation of Streams

Another basic feature in stream processing is the ability to correlate (join) streams. Table 3.7 describes stream correlation in Esper, Siddhi, Odysseus, and Flink.

Table 3.7: Correlation of Streams examples

language	example

Esper	Esper provides a very powerful and complete stream correlation ability.
	In Esper there is the "classic"joining over a certain common stream parameter as well as joining multiple streams. Moreover, you can use comma in Esper to show joined streams instead of using the **join** keyword. General rule for stream correlation in Esper:
	`...from streamdef [as name]`
	`((left|right|full outer) | inner) join streamdef`
	`[on property = property [and property = property ...] ]`
	`[ ((left|right|full outer) | inner)`
	`join streamdef [on ...]]...`

Siddhi	Siddhi has a straightforward way in correlating streams.
	As also seen in the example below, it looks like the "classic" join between two tables over a certain key in SQL. Example:
	`from TempStream[temp > 30.0]#window.time(1 min) as T`
	` join RegulatorStream[isOn == false]#window.length(1) as R`
	` on T.roomNo == R.roomNo`
	`select T.roomNo, R.deviceID, 'start' as action`

Odysseus	Odysseus has a join function which is able to correlate two streams.
	The predicate is the condition, and **Card** shows the correlation type.
	Example:
	`output = join(predicate = 'auction_id = auction', CARD='ONE_MANY'`
	` , left, right);`

Flink	Flink provides a **join** function between streams.
	Example:
	`dataStream.join(otherStream)`
	` .where(<key selector>).equalTo(<key selector>)`
	` .window(TumblingEventTimeWindows.of(Time.seconds(3)))`
	` .apply (new JoinFunction () ...);`

Esper and Siddhi have a SQL-like way in joining streams together over a certain condition. But as a general abstraction, each of these languages provides a join function in order to correlate streams together. The only concern is that Esper allows more streams to be

correlated at once, while the others need more usage of the join function internally as the join function in Siddhi, Odysseus and Flink each take only two streams at once to perform correlation among them.

We will take the "join"function as an abstraction for the canonical language. Although languages like Esper provide other forms of joining, it is not provided by all of the languages, and therefore, will not be included as an abstraction among them. Joining more than two streams at once is, however, of our interest in the canonical language, because it is possible to achieve that in any of the mentioned complex event processing languages.

3.2.3 Filtering Streams

One of the fundamental stream processing operations is filtering input streams based on a condition which includes any of the parameters defined in it. Table 3.8 shows stream filtering in Esper, Siddhi, Odysseus, and Flink.

Table 3.8: Filtering Streams

language	example

Esper	Using a valid condition inside paranthesis is required,
	right after the stream name to create a filter. Example:
	`select sum(volume)`
	`from org.esper.example.StockTick(symbol='IBM')`

Siddhi	Siddhi is providing a simple filtering on streams as well.
	The only difference is that Siddhi uses brackets after streams names.
	In general, Siddhi and Esper are very much alike when it comes to filtering;
	Example:
	`from TempStream[(roomNo >= 100 and roomNo < 210) and temp > 40]`
	`select roomNo, temp`
	`insert into HighTempStream;`

Odysseus	Odysseus provides a **select** function in order to select a certain part of the input stream with a predicate condition, which performs the filtering operation on the input stream. Example: `output = SELECT( predicate='price > 100' , input);`

Flink	Every **DataStream** object which is created in Flink has a filter method which is able to filter the datastream over a certain function. The user must define/override the filter function and provide a condition. As in the example below, the function receives stream data and returns data if it meets the condition. Example:
	`dataStream.filter(new FilterFunction<Integer>() {`
	` @Override`
	` public boolean filter(Integer value) throws Exception {return value != 0;}`
	`});`

The common part of every language in terms of filtering is to have a predicate condition over a stream. This can be abstracted and used in the canonical language. Having streams and predicates, one might have more difficulties in overriding a filter function in Flink rather than just putting the predicate inside parentheses of the query inside Esper. However, it is possible to perform filtering in those complex event processing languages, and therefore, it is added to the canonical language.

3.2.4 Windows on Streams

Streams are received in a continuous manner, and one of the basic needed operations in stream processing is to measure a specific window of information over a stream. A window usually has a type (elements, time, etc.) and according to that, the window is created over the stream. Here, time and element (length) windows are considered. The first one crops a certain section of the stream over a defined time period, and the second one, chooses a part of the stream based on a number of elements. Table 3.9 shows details of window creation in Esper, Siddhi, Odysseus, and Flink.

Table 3.9: Windows on Streams

language	example

Esper	In Esper, as shown in both examples, a hashtag after the stream name in the **FROM** part of the query must be inserted to define windows.
	If the stream is already filtered, the window definitions come afterwards.
	Window keywords are **time** and **length**,
	each accepting one parameter. Example:
	`select sum(price) from StockTickEvent(symbol='GE')#length(5);`
	`select sum(price) from StockTickEvent(symbol='GE')#time(1 sec);`

Siddhi	In terms of window creation, Siddhi looks pretty much like Esper in defining windows inside the query. There is only a slightly syntax difference as shown in the examples below. Example:
	`from TempStream#window.length(10) // or`
	`from TempStream#window.time(1minute)`
	`select max(temp) as maxTemp`
	`insert into MaxTempStream;`

Odysseus	Odysseus has two separate functions called **Timewindow** and **Elementwindow** in order to create a window from the input stream.
	Examples:
	`output = TIMEWINDOW( size = [5, 'SECONDS'], advance = 1, input);`
	`output = ELEMENTWINDOW( size = 5, advance = 5, input);`

Flink	Flink provides two different functions for time and element windows called **timeWindowAll** and **countWindowAll** which correspond to time and length windows. In order to use **timeWindowAll** you need to provide a valid **Time** value. Examples:
	`dataStream.timeWindowAll(Time.seconds(10));`
	`dataStream.countWindowAll(20, 10);`

Time and Length windows are common among all of the languages and an abstraction of both of them can be added to the canonical language. It can also be derived that performing a window operation on streams is rather easier in Esper and Siddhi. For example, consider the case of having a filter and a window together on a stream, while one only needs a small addition in Esper and Siddhi, Odysseus and Flink seem to need to have the output of the first operation and use it as an input to the other operation. By using the presented canonical query language, this complexity is reduced with the translations.

3.2.5 Operations on Windows

Stream processing languages often offer a series of additional functions which can be performed on windows. While the extent of these functions are various according to the abilities of them, Table 3.10 shows examples of them in Esper, Siddhi, Odysseus and Flink.

Table 3.10: Operation on windows

language	example

Esper	Esper offers a list of many different and various types of operations which can be performed on a window basis. Below are examples of those operations, which include sorting items and receiving the first or the last element in every window. Also, in order to count the number of elements in a window one can directly use **COUNT** in the query output parameters.
	Examples:
	`select sum(price) from StockTickEvent#sort(10, price desc);`
	`select * from ReferenceData#firstevent;`
	`select * from StockTickEvent(symbol='GE')#lastevent;`

Siddhi	In regard to Window operations, Siddhi is similar to Esper as it also is in window creation. One can also use **COUNT** in a query with Siddhi to count the number of elements in the created window. Example:
	`from TempStream#window.sort(2,volume,'asc') output all events;`
	`[define window](. . . )[. . . ] output first events;`
	`[define window](. . . )[. . . ] output last events;`

Odysseus	Odysseus uses aggregation functions to work with a created window. After defining a window, it can be used as an input to create outputs which sorts it or chooses a function to be performed over that window.
	Examples:
	`window = TIMEWINDOW(size=[5,‘MINUTES‘], advance=[1,‘MINUTES‘],input);`
	`output = AGGREGATION(aggregations =[[‘FUNCTION‘ = ‘FIRST‘]], window);`
	`output = SORT( attributes = [‘id‘], acending = [‘true‘], window);`

Flink	The easiest way in Flink to perform operations on windows is to attach a **process** function to it. Flink is implemented in Java, so after adding a process method, one can create custom process functions by extending **ProcessWindowFunction**. This will use the current context and an **Iterable** to go over a window and perform a function. Example:
	`dataStream.timeWindowAll(Time.minutes(1), Time.seconds(10))`
	`.process(new MyProcessWindowFunction());`
	In this way, one can programmatically implement all above mentioned operations inside the **process** method:
	`public class MyProcessWindowFunction extends ProcessWindowFunction<T>`
	`void process(..., Context, Iterable<T> input, Collector<T> out) ...`

The sorting operation, finding the first and last element of each window, and counting the number of elements in it, are operations which can be used in the canonical language. Although there might be other operations which are feasible in all the languages, In this thesis only these four basis operations are used.

3.2.6 Pattern Detection

Table 3.11 shows a summary of detecting patterns in Esper, Siddhi, Odysseus and Flink.

Table 3.11: Pattern Detection

language	example
Esper	Esper uses the keyword **every** to determine a sequence of events, combined with **where**, which shows the time period, during which the pattern should be detected. Example: `every a=A -> (B(id=a.id) -> C(id=a.id)) where timer:within(1 hour)`

Siddhi	Detecting patterns in Siddhi is almost similar to Esper.
	Example:
	`from every( e1=TempStream ) ->`
	` e2=TempStream[ e1.roomNo == roomNo and (e1.temp + 5) <= temp ]`
	`within 10 min`

Odysseus	Odysseus uses the SASE[13] operator to detect the sequencial patterns.
	The query, schema, event type and input streams are the input parameters of it.
	Example:
	`query = 'PATTERN SEQ(person p, bid b)`
	` WHERE skip_till_next_match(p,b)`
	`  p.id = b.bidder, b.price > 200`
	` RETURN p.id, p.name, b.price',`
	`schema=[['id','Integer'],['name','String'],['price','double']],`
	`type='PersonEvent1'`

Flink	Flink provides a pattern detection API which creates a **Pattern** instance and defines how the sequence should look like, by using methods **begin**, **next**, **followed by**, and finally using the **within** method setting the **Time** parameter. Each of those methods can override a **filter** function which can be used to put a condition on that step of the overall sequence.
	Example:
	`Pattern<Event, ?> pattern = Pattern.<Event>begin(ßtart")`
	`.next("middle").where(new SimpleCondition<Event>() @Override filter)`
	`.followedBy(ënd").where(new SimpleCondition<Event>() @Override filter)`
	`.within(Time.seconds(10));`

In terms of detecting patterns, Esper and Siddhi use a simpler approach which includes it in the query which is sent to the engine. Detecting patterns in Odysseus requires extra operators and parameters and in Flink, the condition must be used first to create the pattern object and then it must be used on the streams in order to create the whole pattern detection

[13] https://wiki.odysseus.informatik.uni-oldenburg.de/display/ODYSSEUS/SASE+operator

system. As a general outcome, including the sequence condition and the time in which the pattern should be detected, all of these languages are able to detect patterns, and therefore, it is used in the canonical language.

3.3 Canonical Language

After studying each feature among those languages in the previous section, an abstraction layer which includes all of them as a canonical language for complex event processing is created.

3.3.1 Language Definition

In this section, the canonical language based on the foundings from features of previous sections is introduced. The language is created based on the combination of features presented in the previous chapter, and checked using EBNF[14]. Rules 9 and 10 imply that a stream source in a query could be defined as a simple name which might have a filter statement attached to it with a possible window definition or just be a pattern. Rules 12 until 18 are about the later possibilities of having a stream in the FROM part of the canonical query; consisting what the rules considering patterns and windows are. Rules 25-28 are to create a time input to the canonical query. Most of the languages use a very simple time input based on hours, minutes and seconds and here it will also follow the same structure. Rules 29 to 31 are simply put to create different words for any stream name or parameter name or numbers wherever they are needed inside the query, or to know what to use for any of the operators where a condition is needed inside the query.

Listing 3.1 Canonical language definition rule: stream

```
1. stream => 'DEFINE STREAM' stream_name '(' stream_argument ')';
```

In order to use any stream in the query, all streams must be defined at the beginning of it. The rule simply requires any stream definition to start with **DEFINE STREAM** and afterward a stream name from rule 11. Right after stream name, stream arguments should be put in parentheses according to rule 2.
Example: `DEFINE STREAM TemperatureSensor2(...)`

[14] http://www.icosaedro.it/bnf_chk/bnf_chk-on-line.html

Listing 3.2 Canonical language definition rule: stream argument

```
2. stream_argument => argument_name argument_type
  [ { ',' argument_name argument_type } ];
```

Every stream argument consists of name and type. In order to complete the stream definition from rule 1, at least one pair of those must be put as arguments of stream creation. In case of a stream having more than one arguments, they must be separated from each other with a comma.
Example: `DEFINE STREAM TemperatureSensor2(TimeStamp string, Temp integer)`

Listing 3.3 Canonical language definition rule: argument name

```
3. argument_name => { word | number };
```

This determines the name of each stream parameter at its definition in rule 2 inside the paranthesis, and can be a combination of words and numbers from rules 30 and 29.
Example: `TimeStamp, Temp1, ...`

Listing 3.4 Canonical language definition rule: argument type

```
4. argument_type => 'integer' | 'string';
```

This rule determines the type of the parameters defined for a stream at its creation. In order for the parammeters to be simple to be used in all complex event processing languages, the types are limited to **Integers** and **Strings**.
Example: `... (TimeStamp string, Temp integer)`

Listing 3.5 Canonical language definition rule: query

```
5. query => 'SELECT' query_output 'FROM' source [ condition ];
```

The query rule is the starting point of the canonical query after creating the stream definitions. It starts with the keyword **SELECT** and then the query outputs based on rule 21, followed by the keyword **FROM** followed by source definition from rule 8 and conditions of the query from rule 6. In other words, this rule maintains that the general form of the query includes selecting a set of output parameters from some stream source which can include conditions. Using general conditions for the query over certain parameters is optional as the sources can also be filtered themselves.
Example: `SELECT temp1 FROM TemperatureEvent`

Listing 3.6 Canonical language definition rule: condition

```
6. condition => 'WHERE' filter_condition ;
```

A condition inside query refers to a general condition which can be defined over an output parameter. This is optional and must be started with the keyword **WHERE** and then using filter conditions from rule 7.
Example: `SELECT ... WHERE temp > 27`

Listing 3.7 Canonical language definition rule: filter condition

```
7. filter_condition => parameter operator (number | word)
  [{ 'and' parameter operator (number | word)}];
```

This rule defines the conditions defined in a **WHERE** statement more in details. There must be at least one condition defined as a parameter according to rule 24, followed by one of the operators from rule 31, and ending with a number or a text. This means that a simple Where-condition must at least have a typical condition phrase where two operands and an operator exist. In addition, when more conditions are needed they must come with the keyword **and** and added in the same way to the rule.
Example: `SELECT ... WHERE temp1 > 27 and temp2 < 33`

Listing 3.8 Canonical language definition rule: source

```
8. source => source_basis {[ aggregation ]};
```

This determines that as a general rule, each source must start with a source basis from rule 9, and can have aggregations with other sources. This means that the query can simply only be using one source defined or be aggregating multiple streams together and using that as a source.
Example: `SELECT ... FROM TemperatureEvent`

Listing 3.9 Canonical language definition rule: source basis

```
9. source_basis => ( stream_def [ window ] ) | pattern;
```

The source definition is presented here. The basis of each source itself can be a simple stream, or a filtered stream which can be followed by a window, or a patterned stream. Because patterns already provide a window of time within which the pattern is detected, the source must be either using a pattern. If a simple stream or a filtered stream is used, a pattern can be possibly added to the source definition. In following rules, the definitions of

Patterns and Windows are presented.
Example: `... FROM TemperatureEvent[Window...] or ... FROM Pattern[TemperatureEvent...]`

Listing 3.10 Canonical language definition rule: stream definition

```
10. stream_def => stream_name | stream_name '(' filter_condition ')';
```

As also mentioned in the previous rule, the source basis from rule 9 can use simple stream definitions with a possible filtering on them. In that case, the stream definition consists of a stream name from rule 11, or a stream name and filter conditions from rule 7 added right after the stream definition in parentheses which basically creates a filter on the stream.
Example: `SELECT ... FROM TemperatureEvent or ... FROM TemperatureEvent(temp>=21)`

Listing 3.11 Canonical language definition rule: stream name

```
11. stream_name => { word | number };
```

This is the name of the stream which is used in rules 1 (while creating a stream) and 10 (while using a simple stream as a source basis for the query). The rule simply implies that the name of the stream can be a combination of words and numbers.
Example: `TemperatureEvent, Sensor2, ...`

Listing 3.12 Canonical language definition rule: pattern

```
12. pattern => 'EVERY (' pattern_sequence 'WITHIN' [ window ] ')';
```

The rule implies that in order to define a pattern as stream source in a canonical query, one needs to use the keyword **EVERY** and then inside parentheses, pattern sequence from rule number 13 followed by the keyword WITHIN and then a window must be put. In general, any pattern in canonical query language uses the general form of 'every' sequence of patterns followed by 'within' a certain time period and rule 15 is used to define a window.
Example: `... FROM EVERY (... WITHIN Window...)`

Listing 3.13 Canonical language definition rule: pattern sequence

```
13. pattern_sequence => word '=' stream_name ['(' pattern_condition ')'] |
  'NOT' stream_name '->' word '=' stream_name ['(' pattern_condition ')'] |
  'NOT' stream_name;
```

A pattern sequence is defined as the occurrence of two events after each other. It starts with the first stream name which can include a condition on it, or it can start with not happening of an event from a stream. The second occurrence after this can be exactly like the first one and includes not occurring of an event using the keyword **NOT**, or a simple event happening, or a coming event with a certain condition on it.

Example: `SELECT ... FROM EVERY (a=Sensor1(Condition...) -> b=Sensor2) WITHIN ...`

Listing 3.14 Canonical language definition rule: pattern condition

```
14. pattern_condition => parameter operator (number | word) |
    parameter operator parameter;
```

The condition mentioned in pattern definition can be defined as a relation among two different parameters or a simple condition over one parameter.

Example1: `... EVERY (Sensor1(temp > 12) -> Sensor2) WITHIN ...`

Example2: `... EVERY (Sensor1 -> Sensor2(temp2 != temp1)) WITHIN ...`

Listing 3.15 Canonical language definition rule: window

```
15. window => window_basis [ window_op ] | '#' 'sort(' sort_clause ')';
```

A window is defined using a window basis which defines the window itself, followed by one of the selected operations on the window. This means in order to have a window over a stream one needs to use rule 16 and create the basis of the time or length window, but defining an operation on the window is optional here. Alternatively comes the sorting function, which creates a window and sorts the items based on input parameters.

Example: `... FROM TempEvent(Window...)(Operation...) or ... TempEvent#sort(...)`

Listing 3.16 Canonical language definition rule: window basis

```
16. window_basis => '#' 'length(' { number } ')' | 'time(' time ')';
```

In order to define the basis of window after a stream, the keyword # follows time with a time input or length with a number input. Time value is also created according to rule 25.

Example1: `... FROM TemperatureEvent#length(2)`

Example2: `... FROM TemperatureEvent#time(2seconds)`

Listing 3.17 Canonical language definition rule: window op

```
17. window_op => '#' ( 'first()' | 'last()' );
```

When a time or a length window is defined, two functions can be inserted after the window itself to be performed on the window. Here using keywords **first()** and **last()** after character # will imply choosing first or the last element of the created time or length element.
Example1: `... FROM TemperatureEvent#length(2)#last()`
Example2: `... FROM TemperatureEvent#time(2seconds)#first()`

Listing 3.18 Canonical language definition rule: sort clause

```
18. sort_clause => { number } ',' { word } ',' ( 'asc' | 'desc' );
```

The input to the sorting window includes a number, a word which determines the parameter inside the stream which the sorting should use, and the sorting direction which is either ascending or descending. This function must be put directly on the stream, and number and word inputs are based on rules 29 and 30 respectively.
Example: `... FROM TemperatureEvent#sort(5, temp 'asc')`

Listing 3.19 Canonical language definition rule: aggregation

```
19. aggregation => { 'JOIN' source_basis 'ON' aggregation_condition };
```

Aggregation starts with the keyword **JOIN** followed by another source basis which was defined before, followed by **ON** and the aggregation condition between streams which is defined in rule 20. Source basis is already defined in rule 9 and this rule can be repeated multiple times which means in order to add more streams to the join statement, one must follow the same rule and define correct aggregation conditions for each pair of the streams.
Example: `SELECT ... FROM Sensor1#time(2minutes) JOIN Sensor3#time(23seconds) ON Condition...`

Listing 3.20 Canonical language definition rule: aggregation condition

```
20. aggregation_condition => parameter '=' parameter;
```

This rule completes the join statement above and basically implies that two parameters from the different streams must be set equal in order to know on what criteria are the streams related together. The parameters follow rule 24.
Example: `SELECT ... ON Sensor1.temp = Sensor2.temp`

Listing 3.21 Canonical language definition rule: query output

```
21. query_output => (functional_parameter | parameter) ['AS' stream_name]
  {[',' (functional_parameter | parameter) ['AS' stream_name] ] };
```

This rule comes between **SELECT** and **FROM** keywords in the overall canonical query structure and determines the outputs of the query itself. Each query output can be made of either a parameter inside a function or just a simple parameter, with an optional alias to it. Moreover, one can add more query output parameters by using a 'comma' between them, all following the same rule. It is clear that at least one output must exist for the query but more can be added optionally.
Example: `SELECT Sensor1.temp AS Temp1 FROM ...`

Listing 3.22 Canonical language definition rule: functional parameter

```
22. functional_parameter => function '(' parameter ')';
```

One possible output for the query can be a parameter inside one of the functions defined in rule 23. However, the function should take the parameter from the stream inside paranthesis and the parameter is defined in rule 24.
Example: `SELECT avg(Sensor1.temp) AS Temp1 FROM ...`

Listing 3.23 Canonical language definition rule: function

```
23. function => 'avg' | 'min' | 'sum' | 'count' | 'max';
```

List of function which are supported by the canonical language.
Example: `SELECT avg(temp) FROM ...`

Listing 3.24 Canonical language definition rule: parameter

```
24. parameter => stream_name '.' word;
```

A stream parameter is created with a stream name and its parameter separated with a 'dot'; Name of the streams follows rule 11.
Example: `Sensor1.temp , Sensor2.id, ...`

Listing 3.25 Canonical language definition rule: time

```
25. time => [ hours ] [ minutes ] [ seconds ];
```

Time value is generated using hours, minutes or seconds, or a combination of them. This means that only the sequence in which they appear matters. They follow rules 26, 27, and 28 respectively.
Example: `SELECT temp FROM Sensor3#time(2minutes 23seconds)`

Listing 3.26 Canonical language definition rule: hours

```
26. hours => { number } 'hours';
```

Any number followed by the keyword hours showing the number of hours.
Example: 2hours

Listing 3.27 Canonical language definition rule: minutes

```
27. minutes => { number } 'minutes';
```

Any number followed by keyword minutes showing the number of minutes.
Example: 2minutes

Listing 3.28 Canonical language definition rule: seconds

```
28. seconds => { number } 'seconds' ;
```

Any number followed by keyword seconds showing the number of seconds.
Example: 2seconds

Listing 3.29 Canonical language definition rule: number

```
29. number => { '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9' };
```

Listing 3.30 Canonical language definition rule: word

```
30. word => { 'A' | 'B' | 'C' | 'D' | 'E' | 'F' | 'G' | 'H' | 'I' | 'J' | 'K' |
   'L' | 'M' | 'N' | 'O' | 'P' | 'Q' | 'R' | 'S' | 'T' | 'U' | 'V' | 'W' |
   'X' | 'Y' | 'Z' };
```

Listing 3.31 Canonical language definition rule: operator

```
31. operator => '=' | '>' | '<' | '<=' | '>=' | '<>' ;
```

3.3.2 Query Examples

In this section, examples of canonical query which are all generated from the rules in the previous section, are presented and described.

```
DEFINE STREAM Sensor1 (temp integer, timeStamp string)
SELECT temp FROM Sensor1(temp<=27);
```

This query defines a stream with two parameters and puts a simple filter on the stream as the value of the temperature should be less than or equal to 27. Therefore the output will include only those values who match the filter criteria.

```
DEFINE STREAM Sensor1 (temp integer, timeStamp string)
SELECT temp FROM Sensor1#time(9seconds)#first();
```

This query defines a stream with two parameters and creates a window of 9 seconds, adding a first element function which will output only the first element of each created window in every period of 9 seconds.

```
DEFINE STREAM Sensor1 (temp integer, timeStamp string)
SELECT avg(temp) FROM Sensor1(temp<=27)#length(3);
```

This query defines a stream with two parameters and creates a window of the last 3 elements which had a temperature value of less than or equal to 27, and outputs the average value of temperature in each window.

```
DEFINE STREAM Sensor1 (temp integer, timeStamp string)
DEFINE STREAM Sensor2 (temp integer, timeStamp string)
SELECT a.temp AS temp1, b.temp AS temp2
FROM EVERY(a=Sensor1-> b=Sensor2(temp = a.temp) WITHIN#time(5seconds))
```

This query defines two streams with each having two parameters and creates a pattern using them. The pattern needs a Sensor1 event to happen and right after that, a Sensor2 event both having the same temperature value within a time interval of 5 seconds. In case of any match, both temperatures will be sent to output as Temp1 and Temp2.

```
DEFINE STREAM Sensor1 (temp integer, timeStamp string)
DEFINE STREAM Sensor2 (temp integer, timeStamp string)
SELECT Sensor1.temp AS temp1, Sensor2.temp AS temp2 FROM Sensor1(temp<26)#time(9seconds)
JOIN Sensor2(temp<26)#time(9seconds)
ON Sensor1.temp = Sensor2.temp
```

This query defines two streams with each having two parameters. Each stream has a filter on the temperature value which only counts if it is less than 26. The streams also have a time window of every 9 seconds and are joined together upon their temperature value.

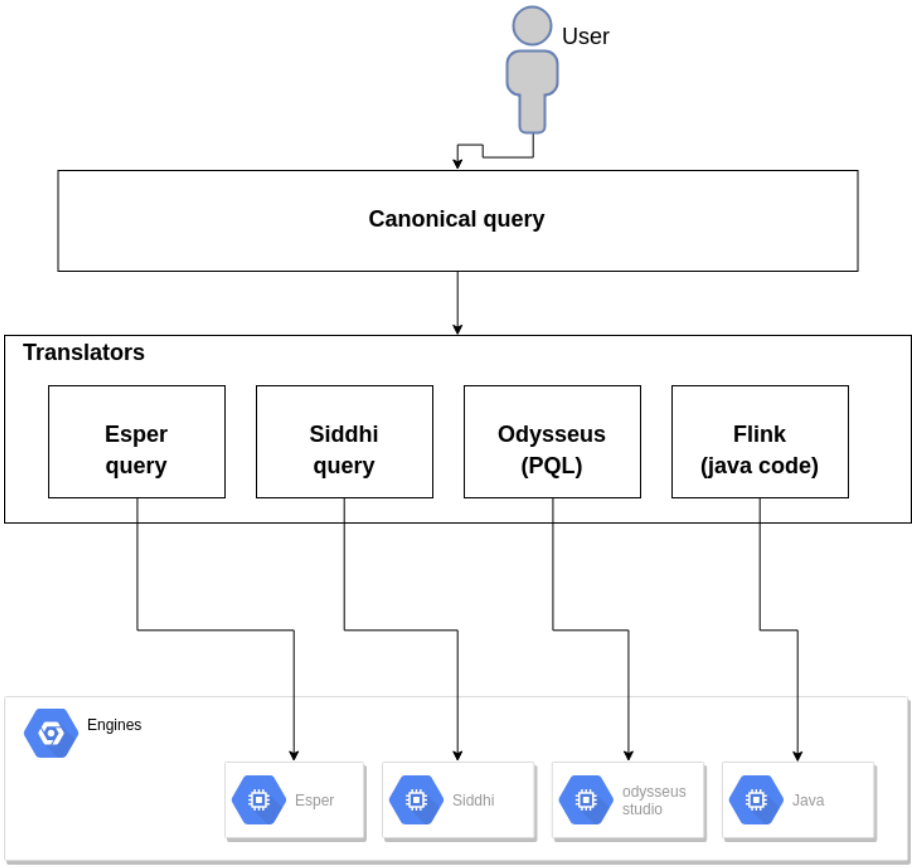

Figure 3.1: Overview of Translation.

3.4 Translation Architecture

In this section, the overall translation architecture of the canonical language is presented. After creating a query based on the grammar from the previous section, it needs to be translated to each CEP language.

The overall architecture is presented in Figure 3.1. According to that, the user creates a query based on the canonical language and then the translation starts. Based on the specified CEP system, the canonical query is sent to one of the translators and the translated query is created and sent to the respective engine, which is already set up and waiting for the query.

Each translator receives the same canonical query from the user and processes it in a different way, in order to create the semantically same query in the specific language.

3.4.1 Architecture

The query processing starts with a canonical query being passed to translation. At this step, the translation begins in one of the translators which is producing the same query in the specific language. This includes all the streams that are defined in the canonical language and the written query itself. In Esper and Siddhi, for example, the definition of the streams must be passed through to the engines itself and be put into the configurations of them, while Odysseus and Flink create outputs that must be implemented in a different way.

According to the definitions in each language, the translators parse the input canonical query and detect different language constructs inside them. The translators break the query into elements, which have been introduced before, e.g., patterns, correlations, streams definitions, query outputs, etc., and then translate them to their counterparts in the specified CEP language and finally put them together for the general outcome of the translation. Each translator parses the canonical query to the known elements (e.g. source definitions, query outputs, patterns, windows, streams, and joins) and translates them to the corresponding definitions in the requested language. After creating each smaller translated parts, they are put together inside the translator to create the translated output query.

3.4.2 Query Translation

In this section, the translation details for each language are explained in more details. After each description, an example from a sample translated canonical query is provided.

Esper translator parses the canonical query into known elements and then translates them back into the Esper query language (EPL) version of it. In terms of the streams defined in the canonical query, Esper does not need to include it in the query itself, but the translator passes the stream information to the Esper engine directly so that it is included inside the configuration of Esper engine. This needs to be done before Esper runs the translated query because the query uses those streams. Without definition in the configuration, the Esper engine is not able to identify them. The output of the Esper translator is also a query, which is sent to the Esper engine.

Here is an example of the translated output in Esper EPL:

```
select Sensor1.temp AS temp1, Sensor2.temp AS temp2
from Sensor1.win:time(9 seconds),
Sensor2.win:time(9 seconds)
where (Sensor1.temp=Sensor2.temp)
```

Siddhi translator works like Esper translator due to the similarities between the two languages. After parsing the canonical query to the known elements including streams definitions, patterns, joins, windows, etc., the translator builds a SiddhiQL query and sends it to the Siddhi engine. The difference to Esper in terms of the stream definitions is that Siddhi needs them to be added to the query. In Siddhi, only the query name should be defined in the engine before running a query statement and all stream definitions are made

at the beginning of the query statement itself. Here is a translated output example:

```
define stream Sensor1 (temp int, timeStamp string);
define stream Sensor2 (temp int, timeStamp string);
@info(name = 'query1')
from Sensor1#window.time(9 seconds) join Sensor2#window.time(9 seconds)
on Sensor1.temp==Sensor2.temp
select Sensor1.temp AS temp1, Sensor2.temp AS temp2
insert into outputStream;
```

Odysseus translator creates a Procedural Query Language (PQL) version of the canonical query. It also parses the query into known elements and then translates them into their definitions. In doing so, Odysseus translator creates a PQL query which is sent to Odysseus studio and run there. The difference is that at this stage, the output of the canonical query are stream definitions, operations and functions in PQL and they must be sent to Odysseus studio and run there. Odysseus studio creates the stream definitions and source objects, and performs the rest of the queries using them. Here is a translated example:

```
#PARSER PQL
#RUNQUERY
Sensor1 := ACCESS(source='Sensor1', wrapper='GenericPush',
transport='TCPClient' ,protocol='SizeByteBuffer',dataHandler='Tuple',
options=[['host','localhost'],['port','65440'],['ByteOrder','LittleEndian']],
schema=[['temp','INTEGER'], ['timeStamp','STRING']]);

firstSensor = SELECT( predicate = 'temp <= 27', Sensor1);
output_3 = SELECT( predicate = '(temp <> 21)', firstSensor);
```

Flink translator parses the canonical query into the known elements, like the other translators, and creates Java methods based on DataStream and FlinkCEP libraries, and combines these methods. These generated methods need to be put into Java classes and compiled with the required Flink dependencies. Therefore, the output of the translator must be copied to another environment in which the necessary dependencies are provided. Here is an example:

```
StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();
DataStream<SensorTuple<Integer, String>> Sensor1DataStream = env
  .fromCollection(Sensor1);
DataStream<SensorTuple<Integer, String>> filteredSensor1DataStream = Sensor1DataStream
   .filter(new FilterFunction<>(){
  @Override
  public boolean filter() throws Exception {
    return (temp <= 27) && (temp != 21);
  }
});
```

4 Implementation

In this section, the implementation of the canonical language is done. A java project is used in order to implement the canonical language. This project includes the *translator package*, *engines package* and *unit tests package*.

The *translator* package includes all classes created in Java for the translation. The canonical query string is used as an input to all of the translator classes. There are different classes for Esper, Siddhi, Odysseus, and Flink. Each translation class includes a *translate* method and a series of helper methods. The *translate* method breaks down the input query into different elements, and the helper methods translate each part and return it. Afterward, the translate method combines smaller translations into the complete output of the class.

The *engines* package includes classes needed to implement engines in order to run query tests. There are classes for Esper and Siddhi which include a basic engine setup that receives events and queries and triggers callback functions. The translations from Odysseus and Flink need more implementation effort. The output of the translators cannot be directly implemented in another existing class. Odysseus code needs to be put in Odysseus studio and Flink code needs to be compiled as Java classes separately.

The *unit tests* packages includes a series of Esper and Siddhi tests created to examine different complex event processing features. Each test is created for a specific use case. In other words, each test uses a canonical query for a certain event processing feature and provides the output for it. For example, the sorting stream test uses canonical language rules to create a stream and a sorting query operation on it. It goes into different stages of the architecture from the input query to translation and implementation and produces results in each of the languages.

Section 4.1 provides more details about the java project for implementation. Section 4.2 includes details about the unit tests.

4.1 Project details

Packages, classes, and methods used in the Java implementation of this thesis are explained in this section. There are three main packages for *engines*, *translators* and *unit tests*, each having a list of different classes. Translator classes have many common methods, however, there are some additional methods defined to complete the translation process. Engine

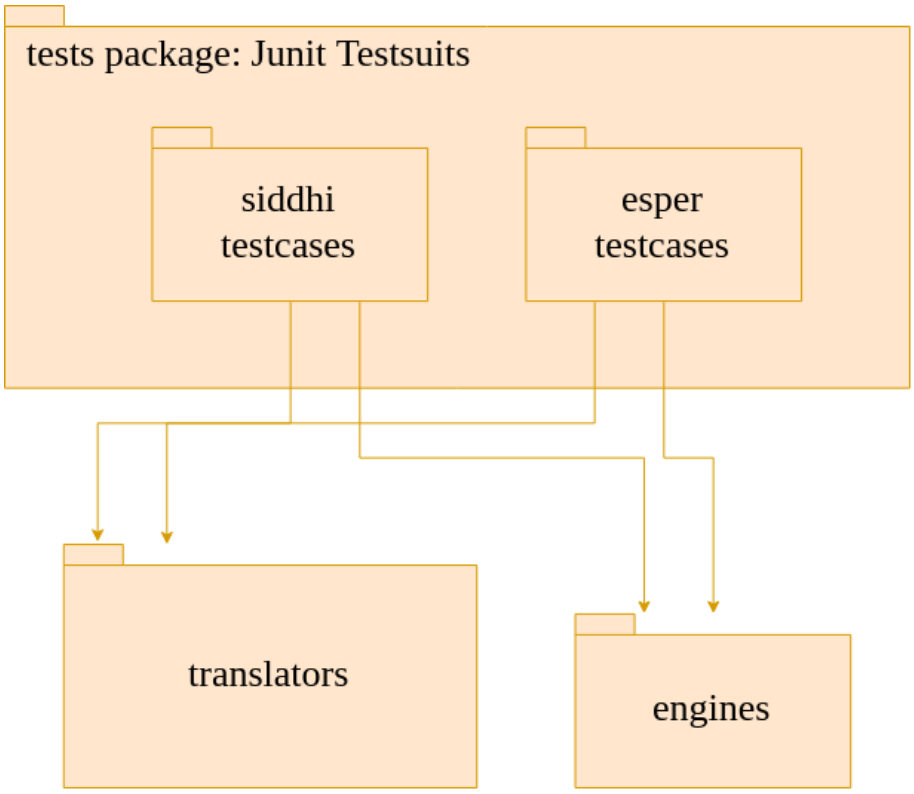

Figure 4.1: An Overview of packages

classes have different methods and Unit tests are defined in Test suits for Esper and Siddhi, separately. In this section, packages from the java project are presented and afterward, their classes and methods are explained.

4.1.1 Packages

As already presented and explained in the architecture, there are three project packages created for the implementation of the canonical languages. The *translation* package includes classes for translating the canonical query into each language. The *engines* package includes classes needed to implement Esper and Siddhi engines, and the *test* package includes test suits for various defined unit tests. Figure 4.1 shows an overview of all packages.

Here, a list of all packages with a description and their classes are explained.

Table 4.1: Packages: Translators.

Package Name:	translators

Package Description:	This package includes all classes and methods needed for the translation of the canonical language queries. Each class has different methods for translation, parsing the input canonical query and producing the translation.

Package Classes:	`EsperTranslator.java`
	`SiddhiTranslator.java`
	`OdysseusTranslator.java`
	`FlinkTranslator.java`

Table 4.2: Packages: Engines.

Package Name:	engines
Package Description:	This package includes classes to implement Esper and Siddhi engines along with a class to create an event object. The *tuple* class is used then in each engine specific class.

Package Classes:	`Esper.java`
	`Siddhi.java`
	`tuple.java`

Table 4.3: Packages: Tests.

Package Name:	tests
Package Description:	This package includes two test suits having test cases in Esper and Siddhi. Each test suite has 13 test cases.
Inside Packages[1]:	`tests.Esper (including 13 junit test cases)` `tests.Siddhi (including 13 junit test cases)`

4.1.2 Classes and Methods

In this section, all existing classes in different packages are explained. Each class includes variables and methods which are explained in tables, along with a short description of the class itself.

Table 4.4 gives variables and methods defined in *EsperTranslator* class. There are two main string variables for storing stream names and the translated output query. There are also different methods to parse specific elements of the canonical query. The *translate* method combines their outputs and creates the final translation in Esper.

[1]Classes of each internal test package are explained in details in the next section. Here for simplicity, only the names of test suits are mentioned.

Table 4.4: Classes: EsperTranslator.

Class Name:	translators.EsperTranslator.java
Class Description:	This class translates a canonical query input to Esper EPL.
Class Variables:	`_streams:String` `_translatedQuery:String`

Class Methods:	`-void translate(String _query);`
	`-String ParseCondition(String _filterConditionBasis);`
	`-String ParseSource(String _sourceBasic);`
	`-String ParseQueryOutput(String _queryOutputBasis);`
	`-String ParsePattern(String _pattern);`
	`-String ParseWindow(String _windowBasic);`
	`-void ParseStreams(String _schema);`

Table 4.5 gives variables and methods defined in *SiddhiTranslator* class. There are three main string variables for storing stream names, window names and the translated output query. There are also different methods to parse specific elements of the canonical query. The *translate* method combines their outputs and creates the final translation into Esper.

Table 4.5: Classes: SiddhiTranslator.

Class Name:	translators.SiddhiTranslator.java
Class Description:	This class translates a canonical query to SiddhiQL.

Class Variables:	`_streams:String`
	`_translatedQuery:String`
	`_outputRule:String`
	`_windowDefinitions:String`

Class Methods:	`-void translate(String _query);`
	`-String ParseCondition(String _filterConditionBasis);`
	`-String ParseSource(String _sourceBasic,`
	`String _queryCondition);`
	`-String ParseQueryOutput(String _queryOutputBasis);`
	`-String ParsePattern(String _pattern);`
	`-String ParseWindow(String _windowBasic);`
	`-void ParseStreams(String _schema);`

Table 4.6 gives variables and methods defined in *OdysseusTranslator* class. There are string variables for storing streams and their parameters, defined windows, defined patterns, etc., and there are also different methods to parse specific elements of the canonical query. The *translate* method combines their outputs and creates the final translation in Odysseus.

Table 4.6: Classes: OdysseusTranslator.

Class Name:	translators.OdysseusTranslator.java

Class Description:	This class translates a canonical query to its equivalant in Odysseus.

Class Variables:	`_streamParameters:String`
	`_numberOfPatterns:String`
	`_numberOfWindows:String`
	`Alias:String`

Class Methods:	`-void translate(String _query);`
	`-String ParseCondition(String _filterConditionBasis);`
	`-String ParseSource(String _sourceBasic);`
	`-String ParseQueryOutput(String _queryOutputBasis);`
	`-String ParsePattern(String _pattern);`
	`-String ParseWindow(String _windowBasic,`
	`String _inputStreamName, String _outputWindow);`
	`-String ParseStreams(String _schema);`

Table 4.7 gives variables and methods defined in *FlinkTranslator* class. There are string variables for storing streams, defined java classes, and defined data types. There are also different methods to parse specific elements of the canonical query. There are also additional methods to create internal methods for windows, parse aggregation of streams, generate time variables in Java and create key selector objects in java. The *translate* method combines their outputs and creates the final translation in Flink.

Table 4.7: Classes: FlinkTranslator.

Class Name:	translators.FlinkTranslator.java
Class Description:	This class translates a canonical query to its equivalant in Flink.

Class Variables:	`_definedStreams:String`
	`_definedClasses:String`
	`_dataType:String`
	`_finalStreamNames:String`

Class Methods:	-void translate(String _query); -String ParseCondition(String _filterConditionBasis); -String ParseSource(String _sourceBasic, ; String _queryCondition); -String ParseQueryOutput(String _queryOutputBasis); -String ParsePattern(String _pattern, String _alias); -String ParseWindow(String _windowBasic, String _source, String _sink); -String ParseStreams(String _schema); -String GenerateWindowOperationClass(String _source, String _windowOperationType, String _windowFunctionName, String _operationClause); -String ConvertTimeToJava(String _parameter); -String ParseAggregationCondition(String _aggregationConditions, String _finalStreamNames, String _strOutput); -void CreateKeySelector();

Table 4.8 gives variables and methods defined in the *Esper engine* class. There is a string variable for storing the running output. There are also methods for creating a listener, random event data, and the class constructor method to implement the engine.

Table 4.8: Classes: Esper Engine.

Class Name:	engines.Esper.java
Class Description:	This class provides the engine setup for Esper.
Class Variables:	_output:String
Class Methods:	-int[] GetRandomNumberList(); -void update(EventBean[] newData, EventBean[] oldData); (this method belongs to the internal class: CEPListener) -Constructor:: esper(String _queryStatement, String _streams);

Table 4.9 gives variables and methods defined in Siddhi engine class. There is a string variable for storing the running output. There are also methods for creating random event data, and the class constructor method to implement the engine.

Table 4.9: Classes: Siddhi Engine.

Class Name:	engines.Siddhi.java
Class Description:	This class provides the engine setup for Siddhi.
Class Variables:	_output:String

Class Methods:	-int[] GetRandomNumberList(); -Constructor:: siddhi(String _queryStatement, String _streams);

Table 4.10 gives a sample Java object class for event data which are sent to Esper and Siddhi streams. As an example, it has a temperature and a timeStamp value with their getter methods.

Table 4.10: Classes: Tuple.

Class Name:	engines.Tuple.java
Class Description:	This class creates event tuples for Esper and Siddhi.
Class Variables:	temp:int timeStamp:String
Class Methods:	-int getTemp(); -String getTimeStamp(); -Constructor:: Tuple(int a, String c);

4.2 Unit Tests

In this section, there are 13 different JUnit test cases defined and presented for each engine. Each of these tests is examining a certain feature of streams (filtering, windows, aggregation, patterns, etc). Each set of 13 test cases for an engine (Esper e.g.) are put in a test suite inside the *project* package. In order to cover both test suits, each test case is separately presented below with both results in Esper and Siddhi.

4.2.1 Test cases

In this section, 13 test cases from the Esper and Siddhi test suite are presented. Each test case is defined for both Esper and Siddhi and the results of those engines are also presented. In each table, the input canonical query and the array of sample input data for both streams are included. Based on the input data, the results of running that test on Esper and Siddhi are presented.

Table 4.11 describes a test case on correlating two streams. Each of them has a filter attribute and the data is used in a time window of 9 seconds. Sensor 1 uses the input array from zero to the end and Sensor 2 uses it vise-versa. The results for Esper and Siddhi are included in the table.

Table 4.11: Test cases: AggregateFilteredStreamsTest.

Test case Name:	AggregateFilteredStreamsTest.java
Test case Description:	Defines two streams and tests the aggregation. Using the array of input values for both streams, the outputs from both engines appears at the same timestamp with the same value.

Input Query:	`DEFINE STREAM Sensor1 (temp integer, timeStamp string)`
	`DEFINE STREAM Sensor2 (temp integer, timeStamp string)`
	`SELECT Sensor1.temp AS temp1, Sensor2.temp AS temp2 FROM`
	` Sensor1(temp<26)#time(9seconds)`
	`JOIN Sensor2(temp<26)#time(9seconds)`
	` ON Sensor1.temp = Sensor2.temp`

Input Data:	[26,30,22,20,28,21,29,30,26,27]
Esper Output:	`{temp2=21, temp1=21}`
Siddhi Output:	`21,21`

Table 4.12 describes a test case on a length window. Sensor 1 uses the input array from the zero index to the end. The test sums up the temp value for each window of 5 elements. The results for Esper and Siddhi are included in the table.

Table 4.12: Test cases: LengthWindowTest.

Test case Name:	LengthWindowTest.java
Test case Description:	Defines a stream and tests a length window on it. Using the input array for stream values, the outputs for Esper and Siddhi are the same.
Input Query:	`DEFINE STREAM Sensor1 (temp integer, timeStamp string)` `SELECT sum(temp) FROM Sensor1#length(5))`
Input Data:	[26,30,22,20,28,21,29,30,26,27]

Esper Output:	`{sum(temp)=27},{sum(temp)=53},{sum(temp)=83},{sum(temp)=112,}`
	` {sum(temp)=133},{sum(temp)=134},{sum(temp)=128},`
	` {sum(temp)=120},{sum(temp)=121},{sum(temp)=126}`

Siddhi Output:	`27,53,83,112,133,134,128,120,121,126`

Table 4.13 describes a test case for aggregating more than two streams. The test correlates streams based on *Temp* value. Sensors 1 and 3 use the input array from the zero index to the end and Sensors 2 and 4 use it vise-versa. The results for Esper and Siddhi are included in the table.

Table 4.13: Test cases: MultipleStreamsAggregationTest.

Test case Name:	MultipleStreamsAggregationTest.java
Test case Description:	Defines four different streams and joins them with each pair of streams having different data and looking for the moments in time where all four streams have the same *Temp* value.

Input Query:	`DEFINE STREAM Sensor1 (temp integer, timeStamp string)`
	`DEFINE STREAM Sensor2 (temp integer, timeStamp string)`
	`DEFINE STREAM Sensor3 (temp integer, timeStamp string)`
	`DEFINE STREAM Sensor4 (temp integer, timeStamp string)`
	`SELECT Sensor1.temp as tmp FROM Sensor1#time(9seconds)`
	`JOIN Sensor2#time(9seconds) ON Sensor1.temp = Sensor2.temp`
	`JOIN Sensor3#time(4seconds) ON Sensor1.temp = Sensor3.temp`
	`JOIN Sensor4#time(5seconds) ON Sensor1.temp = Sensor4.temp`

Input Data:	[26,30,22,20,28,21,29,30,26,27]
Esper Output:	`{tmp=26},{tmp=30},{tmp=28},{tmp=21},{tmp=30},{tmp=26}`
Siddhi Output:	`26,30,28,21,30,26`

Table 4.14 describes a test case on a Pattern. Sensor 1 uses the input array from zero to the end. The test sums up the *Temp* value for each window of 5 elements. The results for Esper and Siddhi are included in the table.

Table 4.14: Test cases: PatternTest.

Test case Name:	PatternTest.java
Test case Description:	Defines two streams and a pattern which detects the *Temp* values if the value from the first stream and the value from the second stream, which comes 5 seconds after it, are the same.

Input Query:	`DEFINE STREAM Sensor1 (temp integer, timeStamp string)`
	`DEFINE STREAM Sensor2 (temp integer, timeStamp string)`
	`SELECT a.temp AS temp1, b.temp AS temp2`
	`FROM EVERY(a=Sensor1-> b=Sensor2(temp = a.temp)`
	` WITHIN#time(5seconds))`

Input Data:	[26,30,22,20,28,21,29,30,26,27]
Esper Output:	`{temp2=21, temp1=21}`
Siddhi Output:	`21,21`

Table 4.15 describes a simple filter test case. Sensor 1 uses the input array from zero to the end. The test puts two different conditions together on a stream; one on the stream as a filter and one using the **WHERE** condition. The results for Esper and Siddhi are included in the table.

Table 4.15: Test cases: SimpleFilteredStreamTest.

Test case Name:	SimpleFilteredStreamTest.java
Test case Description:	Defines a stream and tests simple filters on it.

Input Query:	`DEFINE STREAM Sensor1 (temp integer, timeStamp string)`
	`SELECT temp FROM Sensor1(temp<=27) AS firstSensor`
	` WHERE firstSensor.temp<>21`

Input Data:	[26,30,22,20,28,21,29,30,26,27]
Esper Output:	`{temp=27},{temp=26},{temp=20},{temp=22},{temp=26}`
Siddhi Output:	`27,26,20,22,26`

Table 4.16 describes a pattern for the absence of streams. Sensor 1 uses the input array from zero to the end and Sensor 2 vise-versa. The test creates a pattern where a first stream does not occur, and after that, a second stream happens. The results for Esper and Siddhi are included in the table.

Table 4.16: Test cases: StreamAbsenceTest.

Test case Name:	StreamAbsenceTest.java
Test case Description:	Defines two streams and tests a pattern for absence detection using them. Here the values from second sensor are sent with a delay, so this pattern is detected everytime Sensor1 stream sends events, because the value of Sensor2 is null i.e. not sent yet. The query output parameter *b.temp* is used only for Esper, because Siddhi cannot define pattern parameter before an event arrives. That value is changed to *a.temp* in canonical language query for this test.

Input Query:	`DEFINE STREAM Sensor1 (temp integer, timeStamp string)`
	`DEFINE STREAM Sensor2 (temp integer, timeStamp string)`
	`SELECT b.temp AS temp2 FROM EVERY(a=Sensor1-> not Sensor2`
	` WITHIN#time(2seconds))`

Input Data:	[26,30,22,20,28,21,29,30,26,27]
Esper Output:	`{temp2=null},{temp2=null},{temp2=null},{temp2=null},{temp2=null},` `{temp2=null},{temp2=null},{temp2=null},{temp2=null},{temp2=null}`
Siddhi Output:	`27,26,30,29,21,28,20,22,30,26`

Table 4.17 describes a correlation on streams. Sensor 1 uses the input array from zero to the end and Sensor 2 vise-versa. The test joins two streams using each of them in a time window of 9 seconds and over the *Temp* value. The results for Esper and Siddhi are included in the table.

Table 4.17: Test cases: StreamAggregationTest.

Test case Name:	StreamAggregationTest.java
Test case Description:	Defines two streams and tests their aggregation. Using the array of input values for both streams, the outputs from both engines appear at the same timestamp with the same value.

Input Query:	`DEFINE STREAM Sensor1 (temp integer, timeStamp string)`
	`DEFINE STREAM Sensor2 (temp integer, timeStamp string)`
	`SELECT Sensor1.temp AS temp1, Sensor2.temp AS temp2 FROM`
	` Sensor1#time(9seconds) JOIN Sensor2#time(9seconds)`
	` ON Sensor1.temp = Sensor2.temp`

Input Data:	[26,30,22,20,28,21,29,30,26,27]
Esper Output:	`{temp2=26, temp1=26},{temp2=30, temp1=30},{temp2=28, temp1=28},` `{temp2=21, temp1=21},{temp2=30, temp1=30},{temp2=26, temp1=26}`
Siddhi Output:	`26,26,30,30,28,28,21,21,30,30,26,26`

Table 4.18 describes sorting on a stream. Sensor 1 uses the input array from zero to the end. The test takes every 5 elements and sorts the values descending. The results for Esper and Siddhi are included in the table.

Table 4.18: Test cases: StreamSortTest.

Test case Name:	StreamSortTest.java
Test case Description:	Defines a stream and tests the average of the *Temp* attribute from each window of 5 elements that are sorted on a descending basis.
Input Query:	`DEFINE STREAM Sensor1 (temp integer, timeStamp string)` `SELECT avg(temp) FROM Sensor1#sort(5,temp,desc)`
Input Data:	[26,30,22,20,28,21,29,30,26,27]

Esper Output:	`{avg(temp)=27.0},{avg(temp)=26.5},`
	`{avg(temp)=27.666666666666668},{avg(temp)=28.0},`
	`{avg(temp)=26.6},{avg(temp)=28.0},`
	`{avg(temp)=28.0}, {avg(temp)=28.0},`
	`{avg(temp)=28.8},{avg(temp)=28.8}`

Siddhi Output:	`27.0,26.5,27.666666666666668,28.0,26.6,26.833333333333332,` `26.666666666666668,27.0,28.333333333333332,28.333333333333332`

Table 4.19 describes a time window on a stream. Sensor 1 uses the input array from zero to the end. The test creates a time window of 29 seconds on a simple stream. The results for Esper and Siddhi are included in the table.

Table 4.19: Test cases: TimeWindowTest.

Test case Name:	TimeWindowTest.java
Test case Description:	Defines a stream and tests the maximum of received *Temp* values received so far in time windows of 29 seconds.
Input Query:	`DEFINE STREAM Sensor1 (temp integer, timeStamp string)` `SELECT max(temp) FROM Sensor1#time(29seconds)`
Input Data:	[26,30,22,20,28,21,29,30,26,27]

Esper Output:	`{max(temp)=27},{max(temp)=27},{max(temp)=30},{max(temp)=30,}`
	`{max(temp)=30},{max(temp)=30},{max(temp)=30},{max(temp)=30,}`
	`{max(temp)=30},{max(temp)=30},{max(temp)=30}`

Siddhi Output:	`27,27,30,30,30,30,30,30,30,30`

Table 4.20 describes retrieving the first element of a time window on a stream. Sensor 1 uses the input array from zero to the end. The test creates time windows of 9 seconds on a simple stream and aims only for the first element of each window. The results for Esper and Siddhi are included in the table.

Table 4.20: Test cases: WindowFirstElementTest.

Test case Name:	WindowFirstElementTest.java
Test case Description:	This test defines a window on a stream with the first function. 4 windows are created and the first elements of each are detected.
Input Query:	`DEFINE STREAM Sensor1 (temp integer, timeStamp string)` `SELECT sum(temp) FROM Sensor1#time(9seconds)#first()`
Input Data:	[26,30,22,20,28,21,29,30,26,27]
Esper Output:	`{sum(temp)=27},{sum(temp)=85},{sum(temp)=69},{sum(temp)=78}`
Siddhi Output:	`27,85,69,78`

Table 4.21 describes creating windows from pattern results. Sensor 1 uses the input array from zero to the end. The test creates a window of every 3 elements, each being a detection from a pattern on a stream, and returns the last one among them. Every time the pattern matches, the deteced values are added to an element window and from each 3-element window which is created, the last one is sent to the output. The results for Esper and Siddhi are included in the table.

Table 4.21: Test cases: WindowFromPatternTest.

Test case Name:	WindowFromPatternTest.java
Test case Description:	The test defines a pattern and a window based on pattern detection. From the element window, only the last element is retrieved.

Input Query:	`DEFINE STREAM Sensor1 (temp integer, timeStamp string)`
	`DEFINE STREAM Sensor2 (temp integer, timeStamp string)`
	`SELECT a.temp AS temp1, b.temp AS temp2`
	`FROM EVERY(a=Sensor1-> b=Sensor2(temp > a.temp)`
	`WITHIN#time(3seconds))#length(3)#last()`

Input Data:	[26,30,22,20,28,21,29,30,26,27]
Esper Output:	`{temp2=29, temp1=20}`
Siddhi Output:	`20,29`

Table 4.22 describes retrieving the last element of an element window on a stream. Sensor 1 uses the input array from zero to the end. The test creates length windows of 5 elements on a stream and aims only for the last element of each window. The results for Esper and Siddhi are included in the table.

Table 4.22: Test cases: WindowLastElementTest.

Test case Name:	WindowLastElementTest.java
Test case Description:	The test defines a stream and a length window of 5-elements with a condition of retrieving only the last element of each window.
Input Query:	`DEFINE STREAM Sensor1 (temp integer, timeStamp string)` `SELECT min(temp) FROM Sensor1#length(5)#last()`
Input Data:	[26,30,22,20,28,21,29,30,26,27]
Esper Output:	`{min(temp)=21},{min(temp)=20}`
Siddhi Output:	`21,20`

Table 4.23 describes retrieving the last element of an element window on a stream which already has a filter condition on it. Sensor 1 uses the input array from zero to the end. The test creates length windows of 3 elements on a stream, from values that meet the filter criteria. It then aims only for the last element of each window. The results for Esper and Siddhi are included in the table.

Table 4.23: Test cases: WindowOnFilteredStreamTest.

Test case Name:	WindowOnFilteredStreamTest.java
Test case Description:	This test defines a filtered stream and an element window on it. The last element of each window is sent to the output.
Input Query:	`DEFINE STREAM Sensor1 (temp integer, timeStamp string)` `SELECT sum(temp) FROM Sensor1(temp<=27)#length(3)#last()`
Input Data:	[26,30,22,20,28,21,29,30,26,27]
Esper Output:	`{sum(temp)=74},{sum(temp)=68}`
Siddhi Output:	`74,68`

In all defined test cases in canonical language, the results for Esper and Siddhi are the same. The only exception is for the *Absence Test* where the output parameter must change. Although the results seem to be different, the canonical language creates the same output. Siddhi, unlike Esper, is unable to include an event definition which is not received. For that, the output parameter in canonical input query must change. But that is irrelevant to the absence of events and the test itself succeeds, as the canonical language for event absence creates a correct translation for both Esper and Siddhi and both engines are able to understand and detect the absence.

5 Related Work

Nowadays, Complex Event Processing, as discussed by Buchmann and Koldehofe [2], is one of the important areas both in research and industry, as it deals with monitoring and understanding large numbers of complex events in stream processing and extracting information. Luckham et al.[7], apply a complex event processing approach to network communication systems to create a message-based distributed system. Eugene Wu et al. [5] use complex event processing to create a comprehensive system to work with RFID systems and real-time devices. SASE is used as query languages, which is a high-level SQL-like language for stream processing. Pablo Graubner et al. [4] have incorporated Complex Event Processing in mobile devices with a continuous query language to create a multimodal CEP system.

Many CEP systems have been developed that are able to receive and process complex streams and they each provide a different query language to the end user. Systems and frameworks like Esper, WSO2 Siddhi, Apache Flink, and Microsoft StreamInsight are the most famous vendors. Regardless of the performance analysis of each individual system, there is a need to have a standardization among all those systems in order to obtain a more comprehensive benchmarking and performance evaluation.

Lars Brenna et al. in [1] have developed a CEP system called *Cayuga* that provides its own query language to the user. The language provided is unique, and it covers stream processing features including filtering, pattern detection, and aggregation of events and lacks window definition and operations on windows along with the detection of absent events. It uses a specific query language and provides no translation from other languages.

Mendes et al. [10] discusses CEP features in order to create a standardized benchmark called *BiCEP* that assesses the abilities of different CEP engines. The benchmarking tool succeeds in data and query generation and implementation of the queries in a CEP engine. The translation of the generated query files into vendor-specific languages is still done outside of the benchmarking tool. There is still a need to combine other CEP languages in order to use the benchmarking tool.

Li [6] uses another CEP benchmarking tool called *CEPBen* to analyze the performance of Complex Event Processing systems. While CEPBen offers many of the stream processing features, it is based on Esper. There is no support for other CEP languages and all input queries must be written in Esper's EPL.

There is a general problem in supporting more CEP languages as no canonical standardization among the CEP languages is available. Most of the mentioned works have either created their own language or have selected one specific CEP language which works best. Consequently, creating a benchmarking tool for CEP languages or defining a standard for CEP systems is very much limited to that one language or is done using a newly defined language. A canonical language standard for CEP languages, as described in this thesis, creates input queries which are converted to multiple CEP languages and can be run on specific vendor engines.

6 Conclusion

In this thesis, a canonical language for complex event processing systems is created to provide a standardization for CEP. The canonical language enables a user to write CEP queries which are then translated into different CEP languages. An analysis of well-known complex event processing systems including Esper, Siddhi, Odysseus, and Flink is conducted. A selection of required complex event processing features including window creation, stream definition, pattern detection, stream correlation, ranking and aggregation operations are conducted. Each feature is analyzed in different CEP languages and, as a result, an abstraction for that feature is obtained. A collection of those abstractions is used to create a canonical grammar for complex event processing languages. Having input queries from canonical language, a translation mechanism for each CEP language is created. Afterward, a Java project with packages for translators, test cases, and engines is implemented. A series of 13 tests based on different use cases, including aggregating streams, creating element and time windows on streams, detecting patterns, detecting the absence of a stream, sorting stream events, and more combinations are created.

In the future, in order to extend the canonical language, one can add more stream processing languages. In doing so, it is important to analyze the abilities of the new language and its strengths comparing to the existing languages. Another possible improvement is to add more stream processing abilities. This would also extend the strength of the canonical language to perform CEP queries. In the same way, it is important to analyze the feasibility of the specific new operation in all languages. Moreover, the canonical language system presented in this thesis can be used and integrated into CEP Benchmarking tools, which extends their ability to support multiple CEP systems.

Bibliography

[1] Lars Brenna et al. “Cayuga: a high-performance event processing engine”. In: *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*. ACM. 2007, pp. 1100–1102.

[2] Alejandro Buchmann and Boris Koldehofe. “Complex event processing”. In: *IT-Information Technology Methoden und innovative Anwendungen der Informatik und Informationstechnik* 51.5 (2009), pp. 241–242.

[3] Nikolaus Glombiewski et al. “Event Processing on your own Database.” In: *BTW workshops*. 2013, pp. 33–42.

[4] Pablo Graubner et al. “Multimodal Complex Event Processing on Mobile Devices”. In: *Proceedings of the 12th ACM International Conference on Distributed and Event-based Systems*. ACM. 2018, pp. 112–123.

[5] Daniel Gyllstrom et al. “SASE: Complex event processing over streams”. In: *arXiv preprint cs/0612128* (2006).

[6] Chunhui Li. “Performance management of event processing systems”. PhD thesis. Aston University, 2014.

[7] David C Luckham and Brian Frasca. “Complex event processing in distributed systems”. In: *Computer Systems Laboratory Technical Report CSL-TR-98-754. Stanford University, Stanford* 28 (1998).

[8] David Luckham. *The power of events*. Vol. 204. Addison-Wesley Reading, 2002.

[9] Marcelo RN Mendes, Pedro Bizarro, and Paulo Marques. “A performance study of event processing systems”. In: *Technology Conference on Performance Evaluation and Benchmarking*. Springer. 2009, pp. 221–236.

[10] Marcelo Mendes, Pedro Bizarro, and Paulo Marques. “Towards a standard event processing benchmark”. In: *Proceedings of the 4th ACM/SPEC International Conference on Performance Engineering*. ACM. 2013, pp. 307–310.

A Canonical Language

```
1. stream => 'DEFINE STREAM' stream_name '(' stream_argument ')';
2. stream_argument => argument_name argument_type [{',' argument_name argument_type}];
3. argument_name => { word | number };
4. argument_type => 'integer' | 'string';
5. query => 'SELECT' query_output 'FROM' source [ condition ];
6. condition => 'WHERE' filter_condition ;
7. filter_condition => parameter operator (number | word)
        [{ 'and' parameter operator (number | word)}];
8. source => source_basis {[ aggregation ]};
9. source_basis => ( stream_def [ window ] ) | pattern;
10. stream_def => stream_name | stream_name '(' filter_condition ')';
11. stream_name => { word | number };
12. pattern => 'EVERY (' pattern_sequence 'WITHIN' [ window ] ')';
13. pattern_sequence => word '=' stream_name ['(' pattern_condition ')'] |
         'NOT' stream_name '->' word '=' stream_name ['(' pattern_condition ')'] |
         'NOT' stream_name;
14. pattern_condition => parameter operator (number | word) |
         parameter operator parameter;
15. window => window_basis [ window_op ] | '#' 'sort(' sort_clause ')';
16. window_basis => '#' 'length(' {number} ')' | 'time(' time ')';
17. window_op => '#' ( 'first()' | 'last()' );
18. sort_clause => { number } ',' { word } ',' ( 'asc' | 'desc' );
19. aggregation => { 'JOIN' source_basis 'ON' aggregation_condition };
20. aggregation_condition => parameter '=' parameter;
21. query_output => (functional_parameter | parameter) ['AS' stream_name]
         {[',' (functional_parameter | parameter) ['AS' stream_name] ] };
22. functional_parameter => function '(' parameter ')';
23. function => 'avg' | 'min' | 'sum' | 'count' | 'max';
24. parameter => stream_name '.' word;
25. time => [ hours ] [ minutes ] [ seconds ];
26. hours => { number } 'hours';
27. minutes => { number } 'minutes';
28. seconds => { number } 'seconds' ;
29. number => { '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9' };
30. word => { 'A' | 'B' | 'C' | 'D' | 'E' | 'F' | 'G' | 'H' | 'I' | 'J' | 'K' |
         'L' | 'M' | 'N' | 'O' | 'P' | 'Q' | 'R' | 'S' | 'T' | 'U' | 'V' | 'W' | 'X' |
 'Y' | 'Z' };
31. operator => '=' | '>' | '<' | '<=' | '>=' | '<>' ;
```

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

---

place, date, signature