Master in Informatics Engineering
Internship
Final Report

State-Based Programming in PQL

Ricardo Bernardino
rjrocha@student.dei.uc.pt

Supervisors:
Maria José Marcelino
Ricardo Ferreira
1st July 2014

FCTUC DEPARTAMENTO
DE ENGENHARIA INFORMÁTICA
FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE DE COIMBRA

Abstract

Complex Event Processing (CEP) has become increasingly popular within organizations. The financial industry used to be the sole beneficiary of the CEP capabilities, however nowadays we see it increasingly being adopted by non-financial companies, especially for Business Activity Monitoring software (BAM). One of the most distinctive features of CEP is pattern matching. In this field, pattern matching should not be mistaken for the pattern matching in character strings, nor pattern matching from the functional programming paradigm. Rather, it specifies temporal relationships between events, such as sequences, restrictions of event occurrences, just to name a few. On the other hand, one feature which has yet to be widely adopted amongst the CEP engines is entity state and lifecycle modeling, i.e., the ability to correlate events into instances of an entity, where each entity will have different states and transitions representing its lifecycle.

The objective of this work is to investigate and implement this type of functionality in Feedzai Pulse's query language - PQL.

Keywords: "Complex Event Processing" "Entity Modeling" "Entity Lifecycle" "Event Correlation" "Pattern Matching" "State-Based Programming"

Contents

List of Tables

List of Figures

What are the three most important ideas in programming?
Abstraction, Abstraction, Abstraction.

by Paul Hudak

Chapter 1

Introduction

This thesis describes the work developed in the area of *event pattern matching* and *state-based programming*, at Feedzai, S.A., as part of the Thesis/Project discipline of the Master of Computer Science at the University of Coimbra. This work was supervised by Prof. Maria José Marcelino, and Eng. Ricardo Ferreira.

In this chapter we describe its motivation, background, main objectives and goals, providing also a brief introduction to event pattern matching and entity state and lifecycle modeling, which are fundamental concepts of this work.

1.1 Motivation and Scope

Pulse is a real-time business intelligence platform developed by Feedzai. In this context, real-time does not stand for zero latency. Rather, it denotes the ability of deriving metrics and react accordingly, from the data being received in a timely manner - near zero latency. These metrics in Pulse are called Key Performance Indicators (KPIs), and they can be compared with historical values as well as predicted future values.

At the core of Pulse is a Complex Event Processing (CEP) engine called PKernel, and its query language Pulse Query Language [14] (PulseQL or PQL for short). CEP is defined as the "Computing on inputs that are event streams. For example: Applications that use stock market feeds as inputs and process events in their order of arrival to compute running average stock prices, volume weighted average prices over time windows, etc." [28]. PKernel also deals with events and event streams. Events are represented as *tuples* and they are the basic unit of data in this engine, representing a change in the state of something, e.g., "a financial transaction, a sensor output, a stock price tick" [28]. Event streams are "a set of associated events. It is often a temporally totally ordered set (that is to say, there is a well-defined timestamp-based order to the events in the stream)" [41]. In PKernel, a stream has a data schema associated with it and every associated event must conform to it.

The syntax of PQL is very similar to SQL and Microsoft's LINQ[9], so we have the usual clauses - `from`, `select`, `where`, `group by`, and so forth. Nevertheless, one of its shortcomings is the inability to define and perform pattern matching in sequences of events. Currently, this can only be achieved through

recourse to Pulse's rule engine, but in many other CEP engines pattern matching is done through queries. A possible pattern in a fraud detection environment is the detection of a transaction from a credit card at location `X`, followed by another transaction from the same credit card at a different and quite distant location `Y` within 10 minutes, for instance. If one such sequence of events occurs, we would like to take action (block the transaction) and possibly generate an alarm.

Moreover, there are a number of use cases and business processes which cannot be modeled in PQL when we need events to be related, and we are interested in knowing the *state* of a given entity - what we call *state-based programming* and *entity state and lifecycle modeling*. Imagine a network management environment where we have a huge number of networks, routers, switches, etc. In such an environment, we might be interested in knowing the state a connection might be in - idle, connected, disconnected - as well as how many times it has transitioned from idle to connected, for instance. Another example can be the shipping industry, where we might have the following states for the items: shelf, in transit, lost, delivered; and we would like to know how many items were lost in the last 24 hours. Although the `group by` clause is capable of correlating events in a given stream by an identifier, we have no way of specifying the states and transitions of an entity.

1.2 Goals

In this thesis, we will augment the Pulse Query Language in order for it to be able to express event patterns, as well as the state-based programming. Both of these features are not new in the CEP field, with some people considering the pattern matching to be *the jewel in the crown* [41] and the foundation of CEP systems [52]. We will create a syntax for specifying the pattern sequence, as well as its conditions and operators, that is both simple to write and easy to understand.

Concerning the state-based programming, it still has very little support amongst the major CEP engines. There is some research on the topic [41][35], but not at the same amount level as pattern matching. The goal will be to create a syntax for defining the entities and declare its states and transitions, similar to declaring a class in a Object Oriented Programming (OOP) language, as well as enabling those entities to be used in PQL queries.

1.3 Document Organization

This thesis is organized as follows:

- Chapter 2, "State of the Art" In this chapter, we will analyze some of the most well-known CEP engines available, both commercially and open-source.
- Chapter 3, "PKernel" We will give a brief introduction and overview of Pulse's CEP engine, the PKernel.
- Chapter 4, "Pattern Matching" In this chapter, we will give an overview of common algorithms for doing Pattern Matching, present the language

drafts for the new additions to PQL, and finally we will introduce the implementation of the Pattern Matching capabilities in PQL.

- Chapter 5, "State-Based PQL" In this chapter, we will present the work realized on State-Based programming in PQL.
- Chapter 6, "Work Plan and Methodology" In this chapter we present the planning made for this thesis, as well as the adopted strategies and tools related with software development.
- Chapter 7, "Conclusion and Future Work" As the chapter name suggests, we will state the conclusions of our work, and provide possible directions for further developments.

Chapter 2

State of the Art

In this section, we will analyze some systems (mainly CEP engines) which have Pattern Matching and/or Entity State and Lifecycle capabilities incorporated in them. It would be an almost impossible task to do a thorough research into every CEP Engines that exist, so we will only analyze six engines, both commercial and open-source, that through our research were referenced the most, and also some that we had already some experience during our academic curriculum - Esper and Streambase. In Figure 2.1 we can see a CEP market player analysis, which includes the majority of the engines we will analyze. The systems analyzed will be: Snoop, Streambase, Oracle Event Processing, Esper, Siddhi CEP, and TIBCO Business Events.

2.1 Snoop

Snoop [37][38] is an event specification language for the rules in active databases. An active database is a database in which we execute rules on data being manipulated. The rules follow the structure *Event Condition Action* (ECA), i.e., when the *Event* happens and if the *Condition(s)* apply, we will execute the *Action(s)*. What Snoop provides is a specification for several event types (refer to Figure 2.2), operators to modify events and construct *complex* events. Even though it is not a CEP engine, its operators served as the basis for many of the clauses which, as we will see, are included in almost every CEP engine. Similar event specification languages include SAMOS [44], ODE [45], and EPL [55].

2.1.1 Pattern Matching

We can define pattern matching rules in Snoop through the use of Event Operators. The following are the operators available:

Disjuntion - Or : We use this operator when we want one of the events to occur. Example: $E_1 \vee E_2$, in this example either E_1 occurs or E_2 occurs.

Conjunction : The syntax for this operator is: $Any(I,\ E_1,\ E_2, \ldots,\ E_N)$, where I is the number of events we want to occur in any order, so $I \leq N$ must apply. The authors also provided another operator *All* which is just a shorthand for $Any(N,\ E_1,\ E_2, \ldots,\ E_N)$.

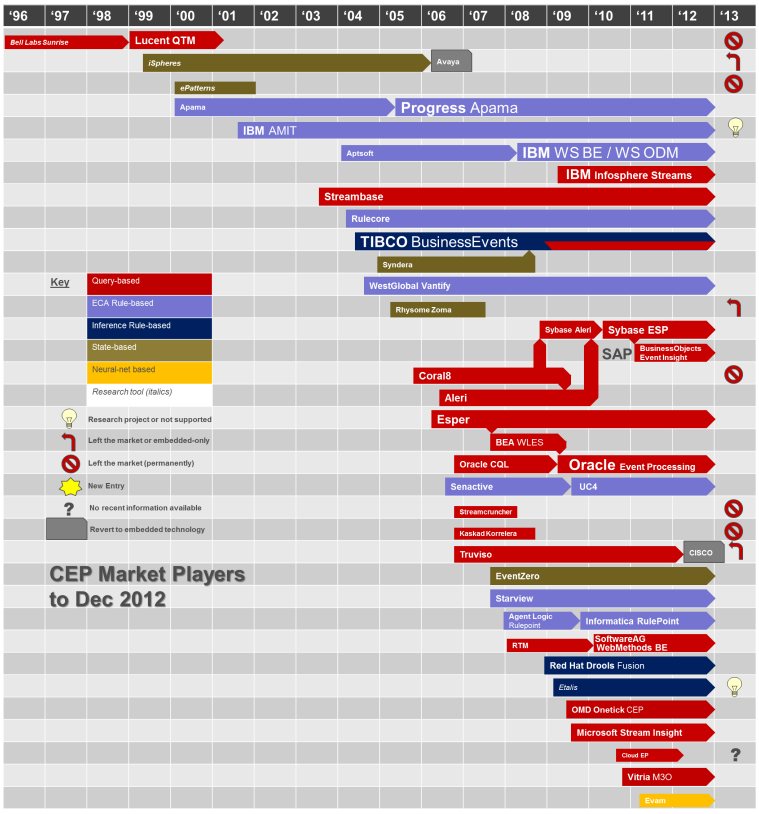


Figure 2.1: CEP market players from 1996 until 2012 - taken from [3]

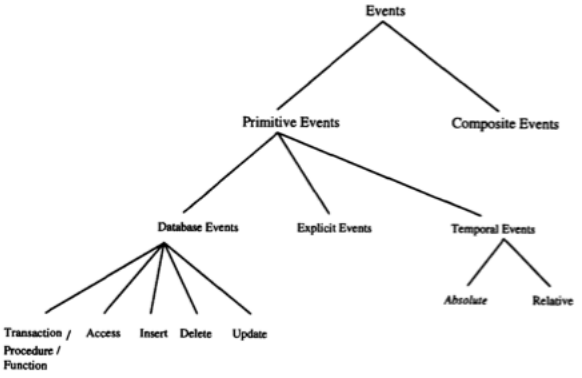


Figure 2.2: Snoop's Event classification[37]

Sequence : This operator specifies a temporal constraint on the events, where one event must occur after the other. Example: $E_1; E_2$, means event E_1 must occur before E_2.

Aperiodic : This operator expresses the occurrence of an *aperiodic* event between two arbitrary events (usually to provide an interval). The syntax is: $A(E_1,\ E_2,\ E_3)$, here event E_2 must occur between E_1 and E_3, yielding results each time an E_2 event happens - *non-cumulative* variation. There is also a syntax for the *cumulative* variation: $A^*(E_1,\ E_2,\ E_3)$, meaning that in the previous example multiple E_2 events would be accumulated and it would yield the results only when E_3 occurred.

```
A(8 a.m., Any(2, modify-IBM, modify-DEC, modify-
    Boeing), 5 p.m.)
```

Listing 2.1: Aperiodic example taken from [38]

In listing 2.1, we have a more common real-life use for this clause, where we want to compute the new Dow Jones index average when any two of IBM, DEC, or Boeing stock prices change during the day[37].

Periodic : The syntax of this operator is very similar to the previous one. It is also a ternary operator, where the first and third parameters have the same functionality, i.e., the second event(E_2) must occur within the interval bounded by the first(E_1) and third(E_3) events - $P(E_1,\ E_2,\ E_3)$. Despite this, in E_2 we must specify a temporal event to indicate the periodicity at which we will yield results, enclosed in square brackets.

```
P(8 a.m., [30 min]: IBM-stock-price, 5 p.m.)
```

Listing 2.2: Periodic example taken from [37]

In listing 2.2 we have an example of this clause, where we will yield the stock price of IBM every 30 minutes, between 8a.m. and 5p.m.

There is also the *cumulative* variant of this operator - denoted P^* - which instead of yielding every 30 minutes the result, it would accumulate the events and yield a single result when E_3 occurs.

After the original paper for Snoop was published, there were some new additions to the Snoop specification[36][31]:

Not : This is a ternary operator with the syntax: $!(E_1,\ E_2,\ E_3)$, meaning event E_2 must not occur between the interval specified by E_1 and E_3.

Plus : This operator is used to specify a relative time period. Example: $Plus(E_1,\ [30min])$, this will signal an event 30 minutes after the occurrence of event E_1.

2.1.1.1 Parameter Contexts

The creators of Snoop decided to create four parameter contexts in order to cope with different application requirements. These parameter contexts can be thought of as event consumption policies, since they are in essence the rules for the eviction of events. In order to comprehend the four different parameter

contexts, we will use the examples provided in [37]. For a sequence belonging to three primitive events ($E1,\ E2,\ E3$):

$E2_1,\ E1_2,\ E1_3,\ E2_4,\ E1_5,\ E3_6,\ E3_8,\ E2_9$

and the following event expression:

- $A = Any(2, E1, E2)\ ;\ E3$

the four parameter contexts are:

Recent : In this context, only the most *recent* event belonging to the primitive event type will be used for the result. Thus, the expression A will have the events $(E2_4,\ E1_5,\ E3_6)$, because $E2_4$ and $E1_5$ are the last occurrences of $E2$ and $E1$, respectively, before an event of $E3$ occurs.

Chronicle : In this context, we will keep every event from the three primitive events ($E1$, $E2$ and $E3$). When an event occurs which signals the expression, it will use the oldest instance of every primitive event, and delete it to prevent further usage. The expression A will have the events $(E2_1,\ E1_2,\ E3_6)$ and $(E1_3,\ E2_4,\ E3_6)$.

Continuous : In this context, every event which matches a *start* event for an expression will set that event as the first event of a new possible result, as well as a new event in a partial result. This can be more easily explained event by event with the example provided below, where we will represent the partial results within ():

1. $(E2_1)$;
2. $(E2_1,\ E1_2)$, $(E1_2)$;
3. $(E2_1,\ E1_2)$, $(E1_2)$, $(E1_3)$;
4. $(E2_1,\ E1_2)$, $(E1_2,\ E2_4)$, $(E1_3,\ E2_4)$;
5. $(E2_1,\ E1_2)$, $(E1_2,\ E2_4)$, $(E1_3,\ E2_4)$, $(E1_5)$;
6. $(E1_5)$; Results: $\mathbf{(}E2_1\mathbf{,}\ E1_2\mathbf{,}\ E3_6\mathbf{)}$, $\mathbf{(}E1_2\mathbf{,}\ E2_4\mathbf{,}\ E3_6\mathbf{)}$, $\mathbf{(}E1_3\mathbf{,}\ E2_4\mathbf{,}\ E3_6\mathbf{)}$.

In this parameter context an event may appear at least once in the results, whereas in both Chronicle and Recent it will appear at most once.

Cumulative : In this context, after the first event to start a *match* occurs, we will add all the events until an event terminates the expression. Thus, the result in this context is: $(E2_1,\ E1_2,\ E1_3,\ E2_4,\ E1_5,\ E3_6)$.

2.1.2 Entity State and Lifecycle

As stated earlier, Snoop is only a language specification for event operations within rules, it was not designed to cope with this functionality.

2.2 Streambase

Streambase is one of the most popular CEP engines, and was recently acquired by TIBCO, also a major player in the CEP field. Part of its popularity is related to their decision to have also a Graphical User Interface (GUI) - called EventFlow - to build the queries, inputs, outputs (refer to Figure 2.3). However, it is still possible to write them in their query language - StreamSQL. Streambase is available both as a 30-day free trial, as well as a paid *Enterprise Edition*.

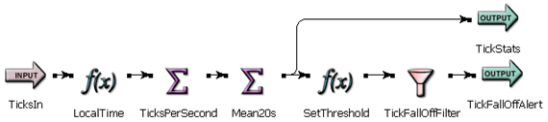


Figure 2.3: Streambase EventFlow application example

2.2.1 Pattern Matching

In StreamSQL we can specify patterns by using the clause `from pattern`. There are five different operators to define a pattern:

`subpattern`$_1$ **THEN** `subpattern`$_2$: With this operator a temporal ordering is imposed on the subpatterns, in which `subpattern`$_2$ must occur after `subpattern`$_1$: $timestamp_{sp2} >= timestamp_{sp1}$. Can also be written as `subpattern`$_1$ `-> subpattern`$_2$;

NOT `identifier` : Indicates the absence of a tuple, i.e., a match occurs only when an event from the stream *identifier* does not happen. Can also be written as `!identifier`;

`subpattern` **AND** `subpattern` : There is a match only when the two subpatterns occur. Can also be written as `subpattern && subpattern`;

`subpattern` **OR** `subpattern` : There is a match if one of the two subpatterns occur. Can also be written as `subpattern || subpattern`;

`pattern` **WITHIN** `x units` : By using this operator, a temporal constraint is imposed on the whole pattern, meaning that a pattern must match within x units of time.

As you may have noticed, we do not have any way of specifying constraints on the attributes of the event received in the stream, within a pattern related clause. To do that, we have to use the global *where* clause, as shown in listing 2.3.

```
SELECT A.id AS fi, C.id AS fo
FROM PATTERN A -> !B -> C WITHIN 5 TIME
WHERE B.id == A.id
INTO out;
```

Listing 2.3: Streambase pattern query with filtering

One should keep in mind that the patterns in this engine are *eager*, i.e., for every event being received, and as long as it passes the first condition, it will start a new pattern as well as matching the already started patterns.

2.2.2 Entity State and Lifecycle

At the time of this writing, there is no support for this functionality.

2.3 Oracle Event Processing

This is the CEP engine developed by Oracle. Oracle has long been a major player in the relational database world, but as more and more CEP engines appeared, it was only a matter of time before there would be a product from Oracle. One of the key features of this product is its Graphical User Interface (GUI), which enables the users to create their queries with a simple drag-and-drop of the query components, and joining them together (refer to Figure 2.4).

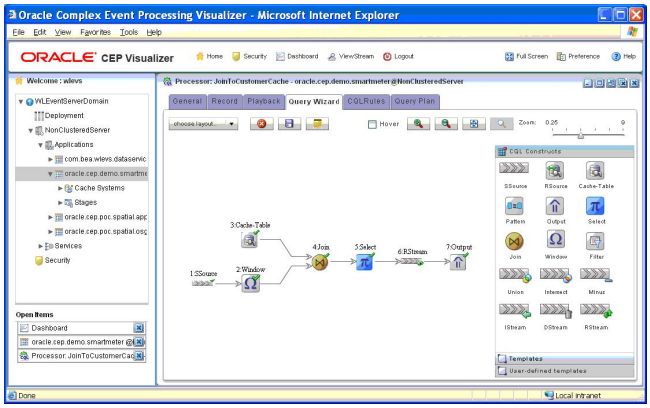


Figure 2.4: Oracle Event Processor's query wizard

2.3.1 Pattern Matching

As for the pattern matching, the Oracle Event Processing has in its query language - Oracle CQL [13] - a clause named `MATCH RECOGNIZE` for this very purpose. `MATCH RECOGNIZE` is a proposal for the SQL standard [12], however it has still to gain traction both on the DBMS side, as well as among the CEP engines - up until this writing only Oracle and Esper implement it. Inside this clause we can use the following clauses:

MEASURES : This identifies the fields to be used by the select clause;

DEFINE : In this clause we specify the conditions we want to match, as well as its identifier to be used in the `PATTERN` clause. Do note that we can reference other identifiers in the conditions - refer to the Listing 2.4.

```
DEFINE
    A as A.price > 20,
    B as B.price > A.price
```

Listing 2.4: Define example

prev : The syntax is the same as a function call, and it returns the previous event in the identifier. Example: `DEFINE B as B.price > prev(B.price)`, this means that an event will be evaluated as a B if its price is greater than the previous event evaluated to B.

PATTERN : As the name suggests, this will be used to specify the pattern we want. The pattern must be defined within parenthesis and it may use an identifier which is not present in the `DEFINE` clause, which will match any event that arrives on the stream;

PARTITION BY : This is an optional clause, but what it does is similar to a group by clause in SQL. Basically, it will apply the pattern independently for each distinct value;

ALL MATCHES : This is an optional clause, enabling the engine to match overlapping patterns;

WITHIN and WITHIN INCLUSIVE : A pattern must match within the time frame provided, otherwise there will be no output;

DURATION : This is an optional clause. Its purpose is to specify a time frame where the pattern must be satisfied completely, i.e., there can be no other events aside from the ones specified in the `PATTERN` clause. With this clause, the pattern only yields results after the specified time frame.

SUBSET : This is an optional clause. This is used to aggregate one or more identifiers from the `DEFINE` clause, which can then be used in aggregation functions, such as average or sum for instance. It is possible to use an identifier in multiple subsets.

Let us focus now on the `PATTERN` clause, and analyze its operators:

Concatenation : This operator is used to define a sequence of events. Unlike all the other operators, it does not have any character to represent it, i.e., if we want to express the pattern A followed by B (concatenation) we simply need to type: pattern (A B).

Alternation : This operator is denoted by the vertical bar character '|'. The function of this operator is to represent the logical or. If we have a pattern: (A | B) we will match either A or B, but not both.

Quantifiers : This operator is used to express the cardinalities expected for the elements in the pattern. There are three quantifiers available, and they are all based on the *regex* syntax:

1. *: The element must repeat zero or more times;
2. +: The element must repeat one or more times;
3. ?: The element must occur zero or one time.

When we specify the quantifiers * or + we can then use aggregation functions on the attributes of those elements:

first : Retrieves the specified attribute of the first matched event;

last : Retrieves the specified attribute of the last matched event;

count : Returns the number of matched events;

sum : Returns the sum of the specified attribute for the matched events;

2.3.2 Entity State and Lifecycle

At the time of this writing, there is no support for this functionality in Oracle Event Processing.

2.4 Esper

Esper is also a very popular CEP engine, and its popularity may lie in its Open Source approach - there is also an enterprise edition though. It is available under the GNU GPL license, and is written in Java. Esper is currently in version 4.10. In Figure 2.5 we have a high-level architecture of Esper, including the components included in its enterprise edition (colored in red), and the EsperHA component which provides high availability to Esper's engine[1].

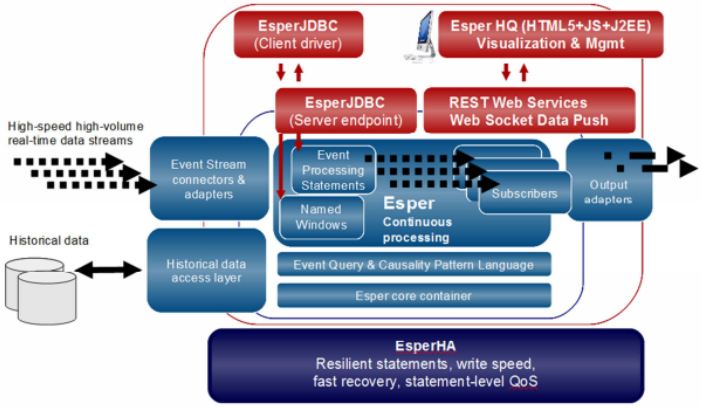


Figure 2.5: Esper's Architecture - taken from [8]

2.4.1 Pattern Matching

Esper and its query language - Event Processing Language (EPL) - has probably the most complete set of operators for pattern matching. EPL is based of project Rapide[51][50] which was developed by the reknown member of the CEP community, David Luckham. In Rapide there were already operators for pattern matching, despite having been developed almost 20 years ago.

In EPL there are two ways of expressing pattern matching queries: Esper's custom one [7], and the above mentioned Match Recognize [6].

[1] EsperHA is only available as an add-on for the enterprise edition.

Examples	Matches
every (A ->B)	$(A_1,\ B_1)$ $(A_2,\ B_3)$ $(A_4,\ B_4)$
A ->every B	$(A_1,\ B_1)$ $(A_2,\ B_3)$ and $(A_3,\ B_3)$ $(A_4,\ B_4)$
every A ->B	$(A_1,\ B_1)$ $(A_1,\ B_2)$ $(A_1,\ B_3)$ $(A_1,\ B_4)$
every A ->every B	$(A_1,\ B_1)$ $(A_1,\ B_2)$ $(A_1,\ B_3)$ and $(A_2,\ B_3)$ and $(A_3,\ B_3)$ $(A_1,\ B_4)$ and $(A_2,\ B_4)$ and $(A_3,\ B_4)$ and $(A_4,\ B_4)$

Table 2.1: Esper's *every* clause examples

2.4.1.1 Esper's

The following are the clauses available:

Followed-by : Indicates that the expression on the left-hand side must occur before the expression on the right-hand side. Example: A ->B;

Every : In Esper a pattern is only tried to be matched once. The way of circumventing this is to add the clause *every* to the desired pattern sub-expression. Let us look at the following examples, taken from the Esper's documentation [7], to better grasp the functioning of this clause.

The table 2.1 is applied for the following sequence of events:

$A_1,\ B_1,\ C_1,\ B_2,\ A_2,\ D_1,\ A_3,\ B_3,\ E_1,\ A_4,\ F_1,\ B_4$

Every-distinct : Same as the previous one, but we only match, as the name suggests, distinct events of the expression.

Repeat : Specifies the cardinality of the expression that follows. Example: $[5]A$, means that five events from A must occur.

Bounded and Unbounded Repeats : We can also have these kinds of cardinalities where we can specify upper and lower bounds, with the syntax: [*lower*: *upper*]. For the unbounded we have the syntax [:*upper*] and [*lower*:].

And : Both the left-hand and right-hand expressions must occur. There is one caveat: if we have pattern A *and* A, then a single A event will match this pattern.

Or : Either the left-hand or the right-hand expressions must occur.

Not : We use this when we do not want the expression to occur for the pattern to match. Example: A *and not* C, means we want to match A without occuring events matching C.

Within : When we want to impose a time limit on the expression we use this clause with the syntax *timer:within(x units)*. There is also a clause *timer:withinmax(x units, max count expression)* which can also bound the number of expression runs.

Timer interval : This clause is used when we want to wait for a while before matching. Example: *A ->timer:interval(10 seconds)*, we match A but only notify of the match after 10 seconds. It can be useful when we want to match an expression and then wait to verify if another expression does not occur within the specified amount of time.

Timer at : This is used when we want to notify of the match at a given time. The syntax is *timer:at(minutes, hours, days of month, months, days of week, seconds(optional), timezone)* - if you are familiar with *cron* expressions you will notice that the syntax is the same.

Filtering : In order to filter the events from different streams, the syntax *stream name*(*attribute 1 operator value*, *attribute 1 operator value*, ...) is used. Example: *a=A ->B(id = a.id)*, here we have a pattern where an event from stream A must be followed by an event from stream B, where its id is equal to the one in the event from stream A.

2.4.1.2 Match Recognize

This clause has the same functionality as in the Oracle Event Processing (refer to 2.3.1), since Match Recognize was a proposed inclusion to the SQL standard [12]. Do note however, that in the case of Esper we have a simple way of matching from multiple Streams, which is available in the public documentation [6].

2.4.2 Entity State and Lifecycle

At the time of this writing, there is no support for defining the states an Event might be in, and its transitions in Esper.

2.5 Siddhi CEP / WSO2 Complex Event Processor

Siddhi CEP was a research project initiated at the University of Moratuwa in Sri Lanka [20]. It is now being developed at WSO2 Inc, and is included in one of their products - WSO2 Complex Event Processor [27]. The WSO2 Complex Event Processor is an open source project, and was developed using the Java programming language.

2.5.1 Pattern Matching

Despite being a research project, Siddhi already had support for pattern matching queries before it was integrated into WSO2. Because of that, we have some research papers describing its architecture and algorithms. In order to have pattern matching queries, the authors of Siddhi decided to implement it using state machines [62]. In Siddhi we have two types of pattern matching related queries:

Pattern Queries : These queries do a *relaxed matching*, i.e., suppose we have a pattern as follows: `A -> B -> C`, meaning an event in stream A, followed by an event from stream B, followed by an event from stream C, and a sequence of events:

$A_1,\ B_1,\ A_2,\ A_3,\ C_1,\ A_4,\ B_2,\ B_3,\ C_2$

The result of this query will be the events A_1, B_1 and C_1. This means that any event in between partial matches can be ignored if they do not match the query.

Sequence Queries : Unlike the previous query type, this one will not match any event given the same pattern and sequence of events, since an A event must be strictly followed by a B event, followed by a C event. (Examples taken from [62])

It is important to note that in Siddhi the pattern queries are not continuous by default, i.e., once a pattern matches the query is no longer re-evaluated. If we do not want the latter behavior, we must add the clause 'every' to the query. This is similar to what is done in Esper. Let us take a closer look at the clauses available for the *Pattern Queries* in Siddhi/WSO2 Event Processing[26]:

Every : As stated earlier, without this clause before a sequence expression, the pattern will no longer be re-evaluated after the first match;

And : When this clause is used, the two events must occur in any order, i.e., if we have the expression A and B it will be matched either with A_1, B_1 as well as B_1, A_1.

Or : When this clause is used, we only need a single event from one of the sides of the expression to occur.

Counting partial matches : In Siddhi we can specify the cardinalities of the events, just like in Esper. The syntax however, is different from the latter. Where in Esper we enclose the minimum and maximum limits within square brackets (e.g. `[2:3]A`), in Siddhi we use the less and greater characters (e.g. `A<2:3>`).

Within : With this clause we restrict the timespan of the pattern matching, meaning that all the events for the pattern to match must occur in the specified time window, defined in milliseconds.

Filtering : In Siddhi we have the possibility of matching events from different streams. The way this is done is through the syntax: *stream_name*[*attribute op value*] - example: `s1 = StockExchangeStream[price > 20]`. In the previous example, we introduced an *alias* to the event received - s1 - which can then be used for the *select* clause, or even when we want to filter from a stream based on an event already successfully matched.

The *Sequence Queries* are a little more restrictive, and thus do not have the same clauses as in the *Pattern Queries*:

Every : Same as in the previous query type.

Counting partial matches : Since this is a *strict* matching, the authors of Siddhi decided to have a syntax more similar to regular expressions:

*: The element must repeat zero or more times;

+: The element must repeat one or more times;

?: The element must occur zero or one time.

Within : Same as in the previous query type.

2.5.2 Entity State and Lifecycle

At the time of this writing, there is no support for defining the states an Event might be in, and its transitions in Siddhi.

2.6 TIBCO Business Events

TIBCO Business Events[22] is a commercial CEP engine solution with several add-on modules to suit different needs and use-cases (see Figure 2.6). The current version of this CEP engine is version 5.1.1.

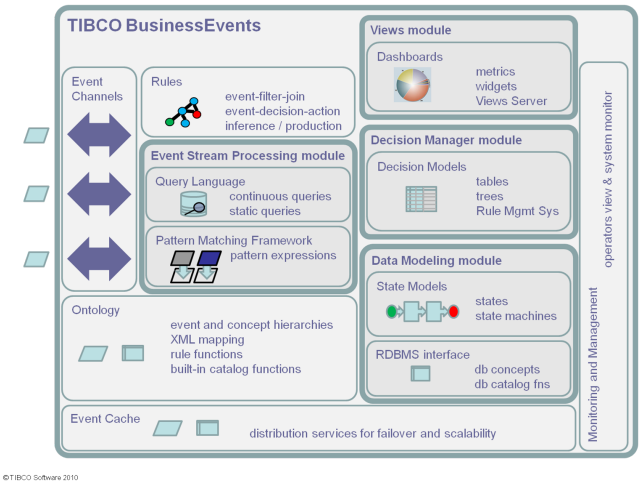


Figure 2.6: TIBCO Business Events 5.1.1 modules

In this thesis, we will only cover the *Event Stream Processing* module and the *Data Modeling* module, for Pattern Matching and Entity State and Lifecycle, respectively. We chose to analyze only these two modules because the remaining do not actively contribute to the features we are interested in analyzing. Since the previous modules are add-ons, these are not provided *out of the box* and must be purchased separately.

2.6.1 Pattern Matching

The pattern queries are syntactically different from the other CEP engine we covered before. In this CEP engine, the authors chose to provide a language more similar to the English language, although it still has some similarities with SQL and regular expression languages. Another difference is the way we insert the queries into the engine. In this engine, we have to explicitly start a *Pattern*

Matcher Service, and then we also have to explicitly send events to that service, otherwise they will not be evaluated.

Let us look at the clauses provided[30]:

define pattern : This clause is used to give an unique identifier to the pattern.

using : In this clause we enumerate the event streams we want to subscribe, i.e., the ones that will be used for event input. We must also provide an alias for the stream to reference the event coming from it.

with : This clause is perhaps the more difficult to grasp, especially if we are already in the pattern matching mindset of the other CEP engines. Basically, with this clause we specify correlation variables, which are variables that relate the different event streams. Example:

with a.id and b.id and c.id = "some string" - this means that we will relate the stream of event *a* with the stream of event *b* according to their *id* value; also, we will relate the stream of event *c* according to the *id* value "some string" - if the value is different from "some string" we will not consider that event for matching.

Another caveat of this clause is that the equal operation (=) is the only one provided, so we cannot have a correlation defined as *a.price > 20*, for instance, which is a significant limitation. Furthermore, an exact match operation cannot belong to the first element in the pattern sequence.

starts with : In this clause we specify the ordering of the events by using the *then* clause. Example: *starts with a then b then c*

We can also have *then* clauses inside other *then*. Some examples[2]:

1. *starts with a then ((a then b))*
2. *starts with a then any one (a, b) then all (a, b)*
3. *starts with a then within 10 milliseconds — seconds — minutes — hours — days b*
4. *starts with a then repeat 10 to 20 times a*
5. *starts with a then all ((a then b), b)*

From the second example onward, we have new clauses to analyze, but also a few which are missing:

any one : This states that any of the events specified must occur.[3]

all : This states that every event in this clause must occur.

within : This has the same behavior as in the other CEP engines, so it enforces the occurrence of the events in the provided time frame.

after : This clause is used to specify that no event should occur during the provided time frame.

[2]These examples were taken from the official documentation[30]

[3]In the documentation there is no explanation as to whether this clause may match all the events, or only one must match.

during : This has the same behavior as the *within* clause. However, it enforces that the next event should occur after the provided time frame, i.e., even though the subpattern already matched, it will still stay in this clause until the time frame expires.

repeat : This clause imposes a lower and upper bound on the number of repeats of the events that follow. In item 4 we have an example of this clause, where first we match an event a and then it must occur 10 to 20 more times.

Negation : Although there is no clause to express the need of absence of an event from a stream, all we have to do is *subscribe* to the stream by specifying it in the *using* clause. If an event from that stream does occur, and since the reference to that event will not be used in any other clause, it will make the pattern fail.

2.6.2 Entity State and Lifecycle

TIBCO Business Events is the first of the CEP engines analyzed to provide us with Entity state and lifecycle modeling, through its *Data Modeling* module.

2.6.2.1 Concept

In this engine, we have an important feature which is a *Concept*. A Concept can be seen as a class in OOP, it has different properties and those same properties can refer to other concepts. There are three types of relationships between Concepts:

Inheritance : This acts just like inheritance in OOP, so every property in the super class will be inherited by its subclasses. Nevertheless, there are some restrictions which are not applied in OOP. Firstly, if two Concepts inherit from the same Concept they cannot have properties (non-inherited) with the same name. Secondly, there cannot exist an inheritance loop, i.e., Concept C1 inherits from Concept C2, which in turn inherits from Concept C1.

Containment : This happens when a Concept is contained inside another Concept. The classic example is of a car which has wheels, doors, engine, windows, etc. There is one caveat however, which is that a Concept may only be contained by a single Concept.

Reference : Unlike *Containment*, this relationship has no ownership over the Concept, so as the name suggests a Concept C1 only holds a reference to an instance of Concept C2 with the provided unique identifier.

An unexpected feature of Concepts is properties history, that is, we can define how many previous values we want to store for a given property, and when that limit is reached the oldest value will be evicted according to the timestamp and date - both these values are generated when a new value is inserted. This behavior is common in some NoSQL data-stores, namely HBase[4].

In Figure 2.7 we have an example of a concept definition for modeling a bank account - the full example is available at [29].

[4]https://hbase.apache.org/

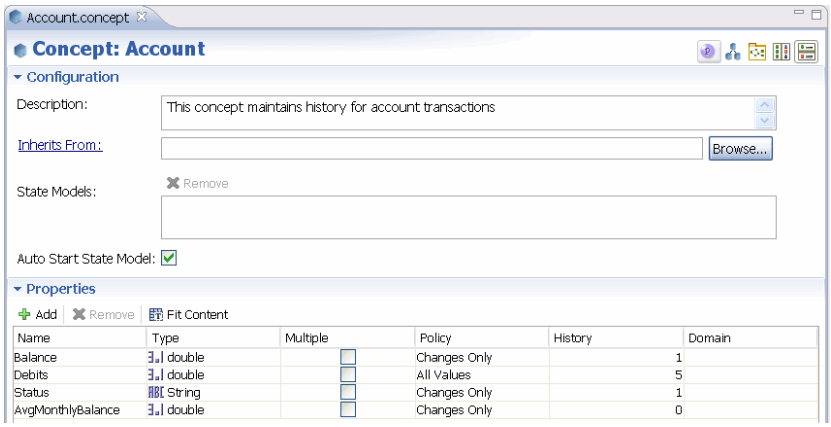


Figure 2.7: Example of a concept definition - taken from `https://docs.tibco.com/pub/businessevents/5.0.0_april_2011/html/tib_be_getting_started/images/concept-editor.gif`

2.6.2.2 State Modeler

As for the state definitions and transitions of a concept instance, they are done in the *State Modeler*, which is based on the UML standard[21]. A concept may have several state models, which contains the states and its transitions, but there can be only a single *main* state model. Also, the state models cannot be shared between concepts - it can only be owned by one concept. Every state model of a concept can only have one start state, but can have multiple end states.

From the previous section, we know that a concept can inherit from another concept. Aside from the property inheritance, a concept will also inherit the state models of its *ancestors*. If we add a new main state model to a concept which inherits from another, we will never use the inherited one. Since we can have multiple state models, it is also possible to use state models which were defined in the concepts which it inherits from. However, a concept cannot use the state models defined in a concept which inherits from it.

Every state can have an entry and exit actions. The only limitation that exists is in the start and end state, where the start state cannot have an entry action and the end state cannot have an exit action. The actions to be executed are all defined using the rules language.

The transitions between different states are done in the rules language of this engine. If a transition has no rule, then it is a *lambda* transition, and after the exit action (if any) executes, the concept will transition to the *next* state. It is also possible to specify a timeout on the transition, where if the rule defined does not execute within the provided timeout value, there will be a transition to the provided state. One caveat is that we cannot use timeouts on the start and end states of a state model. The possible state timeout transitions are:

Current : The concept will continue in the state that it is in, and the timeout timer will be reset. When this occurs, we will not execute the entry and exit actions.

Specified : In this option, after the timeout occurs, the concept will transition

to the state provided. However, we cannot transition to any state that is not linked to the current one.

All : A state can have multiple *next* states. If we choose this option, it will simulate the transition of the concept to every *next* state, and the one which transitions first to another state will be the chosen one.

All the state and transition definitions are created using a GUI - an example can be seen in Figure 2.8.

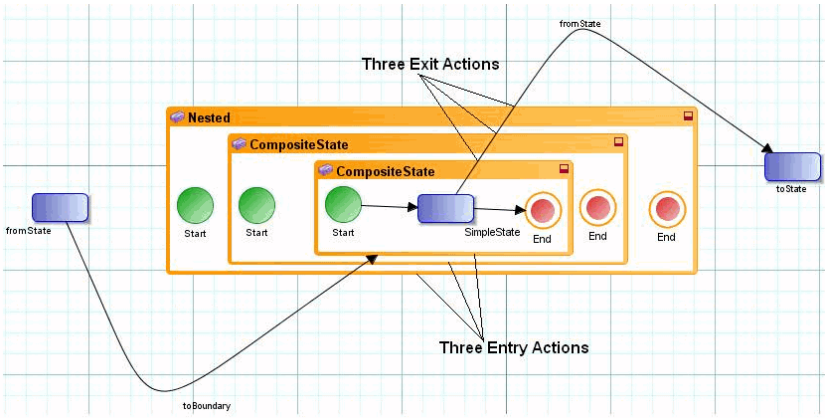


Figure 2.8: Example of state and transitions definition - taken from `https://docs.tibco.com/pub/businessevents_data_modeling/5.0.0_april_2011/html/tib_be_data_modeling_developers_guide/images/state_transition.gif`

2.7 Summary

		Snoop	Streambase	Oracle	Esper	Siddhi	TIBCO
And		■	■		□[5]	■	■
Cardinalities	Exact				■	■	■
	Interval				■	■	■
	Unbounded			■	■	■	
Not		■	■		■	■	■
Or		■	■	■	■	■	■
Previous Event				■			
Referencing Elements					■	■	
Sequence		■	■	■	■	■	■
Within		■		■	■	■	■
Multi-stream			■		■	■	■

Table 2.2: Comparison of Pattern Matching clauses available in the CEP engines analyzed.

[5]Do note that if the left-hand side and right-hand side expressions are the same, a single event will match the pattern, which does not happen in the other systems.

In table 2.2, we summarize this chapter with a comparison of the Pattern Matching capabilities of the different systems analyzed, according to the clauses which we believe are the most relevant ones. We clearly observe that Esper, Siddhi and TIBCO are the engines which provide the most features. Overall, there are some operations which are common amongst the majority of the engines studied, such as: *And*, *Not*, *Or*, *Sequence* and *Within*.

There is no comparison of the Entity State and Lifecycle capabilities, since we can only accomplish it in TIBCO Business Events.

Chapter 3

PKernel

In this chapter, we will provide a brief introduction to PQL, and then to the internals of the CEP engine of Feedzai's Pulse - PKernel. PKernel is composed of five main modules: *frontend*, *driver*, *backend*, *api*, and *test-framework*. We will not analyze the test-framework here, since it does not take part in the compilation or runtime of a query, it is exclusively used for testing purposes. In Appendix F, we have a high-level representation of the architecture of PKernel.

3.1 PQL

PQL has a syntax very similar to that of the standard SQL and Microsoft's LINQ [9] but, because it is specifically tailored for CEP, every query we declare is *continuous*, i.e., once we declare a query, the engine will keep evaluating each time an event occurs until it is removed from the engine. This behavior is not specific to Pulse, rather it is common amongst the CEP engines.

```
stocks_apple_tesla_price_double =
    from stocks
    where symbol == "AAPL" or symbol == "TSLA"
    select symbol, price_double: price * 2;
```

Listing 3.1: PQL query example

The query in listing 3.1 is an example of a PQL query, where we first filter the events from the event stream *stocks* with the symbol *AAPL* - Apple Inc - or *TSLA* - Tesla Motors Inc - and then, we project the symbol field as well as the double of the price field. One thing that will stand out for people who are familiar with SQL is that the order of the clauses is different. Indeed, in PQL we take a sequential approach, i.e., the order of the clauses will be the order in which they will be evaluated and performed.

The variable `stocks_apple_tesla_price_double` is now a new stream with three fields:

timestamp This is done implicitly by the engine, which copies this field from the original event;

symbol This is the same as the symbol field in the original event;

price_double This is the value of the computation `price * 2`.

Furthermore, since it is a stream, it can be used in other `from` clauses to perform any operation we like.

3.2 Frontend

This module is the entry point for the queries and stream (hereinafter referred to as statements) definitions in PQL, and it generates an intermediate representation of the statements of this language. This intermediate representation will be a dependency graph of the different operators. In Figure 3.1, we can examine the different components of this module, as well as the workflow for any PQL statements. We will now provide a brief introduction for each of the former components:

Lexer and Parser This is the first step in this module, and it performs the *lexing* and *parsing* to analyze the syntax of the statements, and build its Abstract Syntax Tree (AST).

Type Checker The AST which resulted from the previous step, is then passed to this component which, as the name suggests, analyzes the types of the statements and compares them to the expected ones.

Query Simplifier In this component we add to the AST new constructs for the implicit fields, by making them explicit fields. As an example the query:

```
from stocks
where price < 100
```

is replaced by:

```
from $ev in stocks
where let price = $ev.price in
    price < 100
```

The `$ev` is later used in the code generation (refer to 3.3.1) to reference the current event received, and the `let` clause can be used to simplify the declaration of variables also in the code generation phase. Finally, it is important to note that the latter query is also a valid one, albeit the former is more user-friendly.

Plan Builder This step processes the updated AST and creates the logical plan for a query. Every clause (`from`, `where`, `group by`, etc) will have its own operator, and those operators can depend on other operators. By creating a logical plan, we are able to abstract the dependencies of each clause, thus providing an ordering for a correct execution. Let us look at an example:

```
nasdaq_index = from indexes
    where name == "NASDAQ"
    select price;

only_aapl = from stocks
    where nasdaq_index.last().price > 4500 and symbol == "AAPL";
```

These queries will create the graphs portrayed in Figure 3.2. As we can

see, every time the `nasdaq_index` changes, the update will be propagated to the `only_aapl` query, through the `OpEventLast` operator.

Plan Simplifier The idea behind the plan simplifier is minimizing redundancy in the plans. For example, when using the `group by` clause, for every distinct *key* value there will be a sub-plan, and without the plan simplifier the use of an external variable would be replicated over all the sub-plans.

Graph Builder This is the final step where we create the dependency graph for the operators from the query plan.

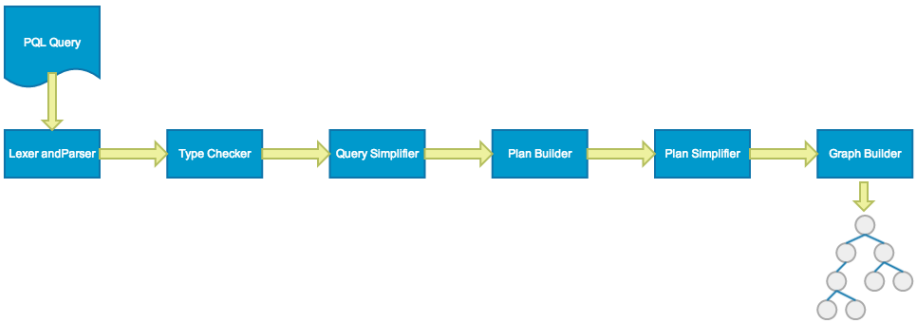


Figure 3.1: PKernel frontend workflow.

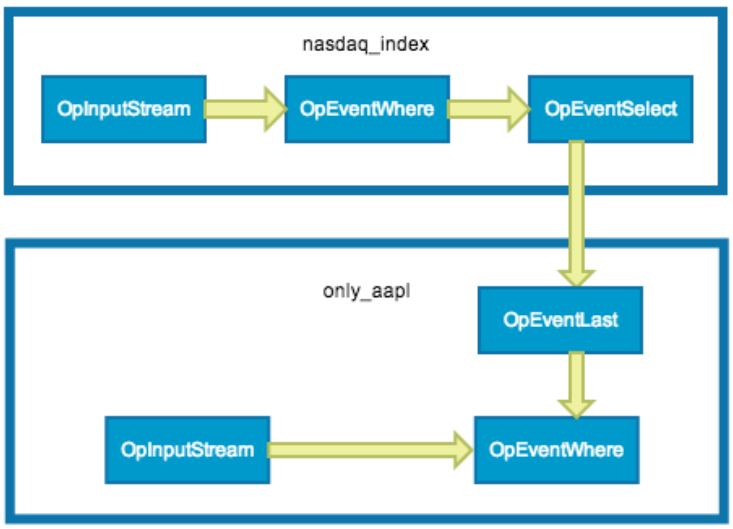


Figure 3.2: The plan graphs generated for each query.

3.3 Driver

This module is the core of PKernel, and it is responsible for all the engine runtime logic, and coordinating the different compilation steps of a PQL statement.

In other words, this module manages all the statement operations - submitting, removing, validating, updating; as well as posting events to the different streams.

3.3.1 Code Generation

The templates for the code generation of a query logic are available in this module - one for each operator. After the *Graph Builder* phase (Frontend), we will return the operator's dependency graph, and using the information contained in the operators, we will fill the templates using this data. The gist of generating Java code is the reduction in the number of function calls and memory usage, since we will not need to have the operator graph in memory, nor have the events passing from operator to operator. This is all the more important in a CEP engine, where queries may be evaluated every time a new event arrives in their inputs. Figure 3.3 illustrates the workflow of the code generation phase.

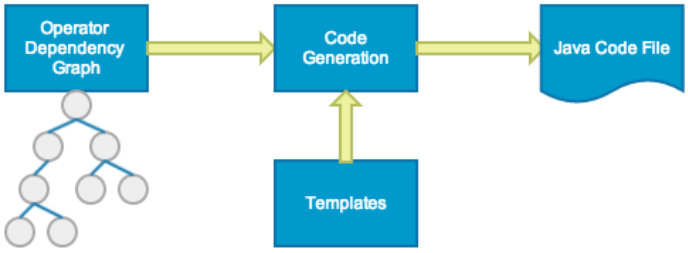


Figure 3.3: Code generation workflow.

3.4 Backend

It is this PKernel module that contains the queries' runtime logic, data structures, and other intermediate objects (POJOs[6]) to be used within the generated code.

3.5 API

As the name suggests, this module provides the interfaces for interacting with the different functionalities supported by the engine. In other words, it is PKernel's entry point for the other modules of Pulse which need to interact with the engine.

[6]Plain Old Java Object

Chapter 4

Pattern Matching

In this chapter we will provide the implementation details for the first new PQL capability - *Pattern Matching*.

This chapter's sections will follow the order we took when starting the implementation. First, we started with language drafts. After that, we identified the requirements for the new additions to PQL. Then, we investigated the algorithms other CEP engines used for pattern matching queries. And finally, the last step was the implementation itself.

4.1 Drafts

The Pattern Matching capabilities were first found as a requirement to the *State-Based Programming* (expounded in the next chapter) in order to have the means to express more complex transitions between the states. As part of the last draft (refer to Appendix H), we also defined how the pattern queries would be used in a PQL query. There are two new clauses:

define
: We will define the `pattern elements` which will be used in the next clause. A `pattern element` is nothing more than an identifier followed by a condition. If one event passes its condition, then it will be assigned that `pattern element`.

pattern
: After defining the `pattern elements`, we will use the elements defined in the previous clause in order to specify the sequence of events that will be matched.

Some examples of possible patterns were also drafted, taking into account the way pattern matching queries were made in other CEP engines (refer to chapter 2):

`A -> B`
: The first A which appears, followed by B

`last A -> B`
: The last A which appears, followed by B

`every A -> B`
: For every A followed by B

`A -> (B or C)`
: The first A followed by the first B or C

`[4] A -> [3] B`
: 4 A's, followed by 3 B's

`[:4] A -> [3:] B`
: At most 4 A's, followed by at least 3 B's

`A -> B and !C`
: The first A followed by B, where a C does not happen between A and B

`A -> B within 10 seconds`
: The first A where a B appears within 10 seconds

`A -> B after 20 seconds`
: The first A where B appears after 20 seconds

`A -> B(a.bandwidth > bandwidth)`
: A followed by B where its bandwidth is less than A's

4.2 Requirements

In any software development project, the requirements phase is probably one of the most important ones, since it helps to get an early view of the system, and it will help in validating the work done. Based on the language drafts created and the study done in Chapter 2, we created the requirements for the new capabilities in PQL - which can be found in Appendix B.

4.2.1 Functional Requirements

The functional requirements of this work are defined using user stories [23][2], with the following structure:

As a(n) *type of user*

I want *some action*

So that *achievement* (optional).

In our work, there is only one type of user which is the developer of the PQL statements, so we will omit this field in the user stories defined. We will also divide the user stories into two groups: the ones used for pattern matching, and the ones used for state-based programming.

The prioritization of the user stories will be done by using the MoSCoW method [10][11]. In MoSCoW, we have four different prioritization levels:

MUST (M) "Defines a requirement that has to be satisfied for the final solution to be acceptable."[10]

SHOULD (S) "This is a high-priority requirement that should be included if possible, within the delivery time frame. Workarounds may be available for such requirements and they are not usually considered as time-critical or must-haves"[10]

COULD (C) "This is a desirable or nice-to-have requirement (time and resources permitting) but the solution will still be accepted if the functionality is not included"[10]

WON'T (W) "This represents a requirement that stakeholders want to have, but have agreed will not be implemented in the current version of the system. That is, they have decided it will be postponed till the next round of developments"[10]

4.2.2 Nonfunctional Requirements

Nonfunctional requirements define "how well the software will work (...) how easy the software is to use, how quickly it executes, how reliable it is, and how well it behaves when unexpected conditions arise" [61]. The nonfunctional requirements for this work are available in Appendix C.

4.3 Implementation Approaches

There are two main approaches when implementing Pattern Matching queries amongst the CEP engines. One is through the use of rules, usually using an algorithm called RETE[42], or one of its variants. JBoss Drools, for instance, uses a custom implementation of RETE called RETE-OO[5]. TIBCO Business Events' (refer to 2.6) rules engine is also based on RETE. The other approach is Finite State Machines (FSM), in a way similar to the implementation of regular expressions. This approach is the one taken by the creators of Siddhi and, according to [49], Snoop. There are other approaches [56][65], but some are application-specific and thus, would not make sense applying them in other domains, e.g. there are algorithms which only apply for events representing spatial relations. Furthermore, in a CEP engine we have an implicit requirement which is real-time capabilities, so any algorithm which relies on stored data should not be used in this context.

In Esper (refer to section 2.4), the pattern matching is done using dynamic state trees [24]. However, there is no documentation detailing the implementation. In the case of Esper's *match-recognize* queries they are built using nondeterministic finite automaton (NFA) - which is a type of FSM.

4.3.1 RETE and its variants

In this section we will briefly analyze the RETE algorithm, as well as some of its variants, which were proposed throughout the years in order to improve its performance.

4.3.1.1 RETE

RETE [42] (pronounced [ree-tee]) is a well known algorithm amongst the rules engine community. It was developed by Dr. Charles L. Forgy in 1979. In RETE

we have data stored in memory and a set of rules, and the objective is to match the rules against the data. However, the main advantage of this algorithm is that upon a new data entry, update or delete, it will only re-evaluate the rules which can be applied to that specific entry, instead of all the rules available, as well as all the previous elements received.

In order not to evaluate all the rules, RETE creates a network of nodes, which will form a direct acyclic graph (DAG). These nodes can be of four distinct types:

Root This is the first node in the network, and forwards the data to all the *Kind* nodes.

Kind The Kind node receives the data and only forwards it if the kind/type of the data matches the one associated with it.

Alpha In an *Alpha* node, the data will be filtered according to the conditions of the rule. For example, if we have a *Kind* People with attributes: age, name and country; then in the *Alpha* node we could have a condition such as $age > 30$.

Beta We can think of *Beta* nodes as join nodes, since they are responsible for joining the data received from the *Alpha* and *Beta* nodes linked to it. Do note that in this node type, there must exist two inputs, when we only have one input there is a *dummy* element in the input that would otherwise be empty, to circumvent this demand.

At the end of the paths in the network we have the rules which successfully applies to the data received. For every *Alpha* node path we also have *alpha memories* (AM) for storing the *working memory elements* (WME), thus eliminating the need for re-evaluating all the past elements. *Beta* nodes also have *beta memories*, however they exist after each node, unlike the *Alpha* nodes, and they store *tokens* which represent a sequence of *working memory elements*.

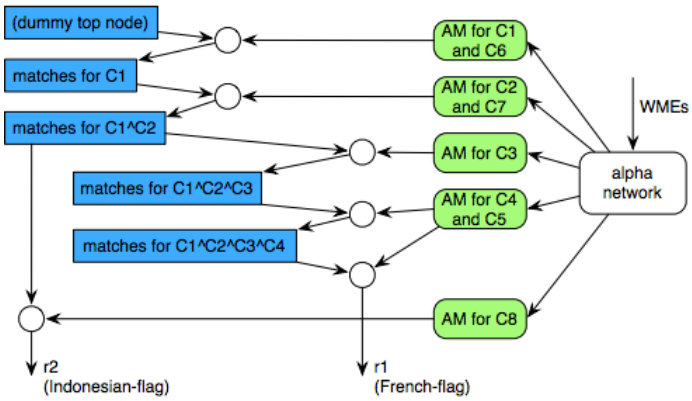


Figure 4.1: Example of a RETE network with two rules (r1 and r2) - taken from [39]

In Figure 4.1 we have an example of a RETE network for two rules which globally specify eight conditions: rule r1 has conditions *C1, C2, C3, C4 and C5*;

and rule r2 has the remaining *C6, C7 and C8*. We can see the *alpha memory* re-usability when the conditions are the same between rules, such as conditions *C1* and *C6*.

After the initial version of the RETE algorithm, the author developed performance improved versions of the algorithm [17][15] - RETE-II [18], RETE-III [25], and RETE-NT [16], in this order. Nevertheless, these newer algorithms are all proprietary and consequently, there is no publicly detailed information as to the optimizations that were made.

It is important to note however, that RETE does not possess temporal semantics and operators out of the box. In order to tailor RETE for a CEP environment, we could adapt some proposals of extensions to RETE, such as [58][63].

According to [1] and [53], the space complexity of RETE per iteration is $O(RFP)$, where R is the number of rules, P the average number of patterns per rule, and F is the number of facts on the knowledge base. The naïve approach space complexity, according to [1], is $O(RF^P)$. This naïve approach consists in keeping the rules in a list and continuously evaluating them, regardless of a need to re-evaluate a rule.

4.3.1.2 RETE/UL

RETE/UL [40] is a variant of RETE created in 1995 by Robert B. Doorenbos. The basic idea of this algorithm is to limit the number of joins we have to perform in the original RETE, by *unlinking* beta memories and alpha memories in two approaches:

Right Unlinking In this approach, we unlink the alpha memory from the join with an empty beta memory. This happens when we have a lot of conditions, and the ones further down the network are still waiting for the conditions above it to be matched.

Left Unlinking This approach is similar, albeit the unlinking is done in the *left* node, i.e., the beta memory is unlinked from an empty alpha memory.

When both the nodes are empty, which might occur when we add the rule to the RETE network, the author states in [40] that we "'can pick one side by any convenient method". The performance benefits of this approach become more noticeable when we have a lot of rules in the RETE network, because if a WME is added to the network, then we will have a large number of nodes affected by that single WME, and it can happen that many of the join nodes will contain an empty input.

4.3.1.3 TREAT

Created by Daniel P. Miranker, the objective of this algorithm was to be a better algorithm than RETE - in the conclusion of [54] the author states that "TREAT is a better production system algorithm in both time and space". In this algorithm, there are no *beta nodes* nor *beta memories*, and thus there exists a single generated network for a given rule, whereas in RETE there can be different networks depending on how we arrange the *beta nodes*, i.e. the order of the conditions in the rule. Furthermore, as a result of the removal of *beta*

nodes, every time a new *working memory element* enters the network it will have to join with all of the *alpha nodes*. This behavior will also make the delete operation faster than in RETE, since we do not have to remove it from every *beta memory*. The observation made by the author was that, in some cases, the *beta memories* would have the same elements stored, thus storing redundant information. By removing the *beta memories* there will no longer be redundant information.

Despite the conclusion made by Daniel P. Mirander in [54], in [57] their results show that in most cases the performance of RETE is better than TREAT's. In another study comparing RETE and TREAT performance [64], the authors conclude that TREAT outperforms RETE "as a rule condition testing algorithm for a database rule system".

4.3.1.4 LEAPS

LEAPS [34] stands for Lazy Evaluation Algorithm for Production Systems, and was developed by Don Batory in 1994. In this algorithm, we are interested in firing one rule as soon as possible. In RETE and TREAT, when there is a new element in the network the fundamental idea is that any rule might fire. Nevertheless, when a rule is fired, the actions executed by this rule might change an element already in the network, possibly making other rules that would fire useless.

One limitation of this algorithm is that it does not take into account a possible ordering of the rules priority, since it will fire the first rule that matches which may restrict the use of this algorithm in certain applications. This behavior is not present in RETE, where it tries to match all the possible rules and then choose one according to the *agenda*.

4.3.1.5 Gator

This algorithm was created by Eric N. Hanson and Mohammed S. Hasan in 1993. The main idea of Gator (Generalized TREAT/RETE) [47][48] is the removal of the double input constraint in *beta memory* nodes. With this, a better network can be built taking into account heuristics related to the costs of joins - just like it happens in a DBMS when building the query plan. Also, since we can have multiple inputs, we are no longer constrained to a binary-tree structure (refer to Figure 4.2).

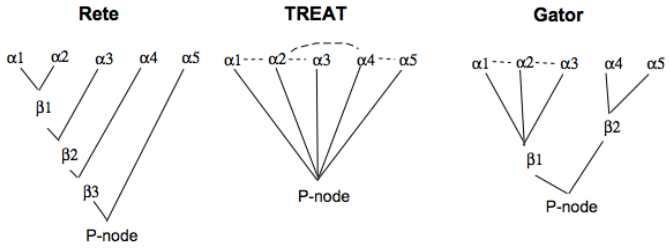


Figure 4.2: Example of RETE, TREAT and Gator networks - taken from [48]

4.3.2 Finite State Machines

4.3.2.1 Siddhi

Siddhi implements FSM in their *sequence* and *pattern* queries [62][60], already analyzed in this thesis - refer to section 2.5. Even though this approach is used in both query types, there are some differences in the way the processing is done.

In a *pattern* query, every time an event is received we will send it to the active *executors* of the event's stream[7] - this behavior deviates from the definition of a FSM where a single state is active. The *executors* are the units responsible for evaluating the event. If the event passes its conditions all the previous events that matched the pattern will be copied to the next *executor* if there is one, otherwise we reached the end of the pattern and we will output according to the projection made in the `select` clause.

In the *sequence* query, the first change is that the event is forwarded to all active *executors*, even though some of them are not related to the event's stream. According to [62], this decision was made due to the functional restrictions of this query type - remember from section 2.5 that in a *sequence* query the order of the events must be equal to the one defined in the pattern, there cannot be other events in-between otherwise the pattern fails. So, instead of storing the executors in a *Map* data structure, the *executors* are stored in a *Linked List* data structure, and all the active *executors* will be removed whether the pattern succeeds or fails.

4.3.2.2 SASE

SASE (Stream-based And Shared Event processing) was a CEP engine and query language developed at the University of Massachusetts [19]. In [46] the authors propose a new version of the language, called SASE+, which extends SASE with *Kleene* closure[8]. In the same paper, the authors introduce the name of the algorithm they created - NFA^b - albeit without going into great detail on the implementation. It is in [32] that we have a more comprehensive overview of the algorithm. NFA^b is an nondeterministic automaton and a match buffer (thus the name). The states in this automaton correspond to the pattern sequence provided in the query, e.g. for the query in Listing 4.1 we would get the automaton displayed in Figure 4.3.

```
PATTERN SEQ(Stock+ a[ ], Stock b)
WHERE skip_till_next_match(a[ ], b) {
    [symbol]
    and a[1].volume > 1000
    and a[i].price > avg(a[..i-1].price)
    and b.volume < 80% * a[a.LEN].volume
}
WITHIN 1 hour
```

Listing 4.1: SASE Query 3 from [32]

[7] In Siddhi each event is represented by a tuple containing the stream name, timestamp of the event, and all the attributes of the event.

[8] Kleene closure is a notation which has two operators - the *plus* '+' and the *star* '*' - to denote the cardinalities of an element, one or more occurrences, and zero or more occurrences, respectively. This notation is also used in regular expressions.

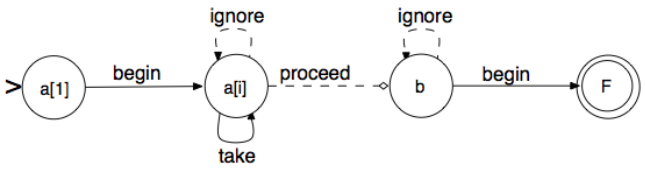


Figure 4.3: Automaton generated - taken from [32]

Moreover, the *ignore* transitions simply illustrate the fact that we may ignore incoming events which do not match the conditions of the query - similar to the *pattern* queries in Siddhi.

In order to construct the automaton, the first step is to create the states by analyzing the *PATTERN* clause. Then, it uses the conditions defined in the *WHERE* clause to derive the edges between the states and to rewrite them into conjunctive normal form (CNF). A sort is also performed in the rewritten conditions according to their *last identifiers* - an identifier being the variables in the *WHERE* clause. In the case of the query presented, the order will be $a[1]$, $a[i]$, $a[a.LEN]$, b. Since there are different event selection strategies, we need to adapt the *ignore* edges according to the one chosen. Finally, if there is a *WITHIN* clause, we will add the respective timestamp constraint to the *begin* and *proceed* edges.

4.4 PQL

In the following subsections we will analyze (in this order): the new clauses introduced in PQL, and the operators for defining temporal relationships between events. As defined in drafts section 4.1, the two new clauses are: `define` and `pattern` - Listing 4.2 contains the grammar rules for these clauses. The *lexing* and *parsing* for PQL is done using ANTLRv3[9], and throughout the next subsections we will provide some snippets of the grammar rules. The complete grammar definition is available in Appendix I.

```
clause: ...
    | DEFINE^ defineExpr partitionBy?
    | PATTERNMATCH^ patternMatchExpr
    ;
```

Listing 4.2: The two new clauses

The `clause` rule is where we have the rules for all the other available query clauses - `select`, `where`, `group by` etc.

4.4.1 Define Clause

In the rule for the `define` clause we have two additional rules - `defineExpr` and `partitionBy`. The former is just a simple rule which states that we must have an identifier followed by an expression, and these two components separated by the character ':'. Moreover, we have to specify at least one identifier and condition.

[9]ANTLR stands for *ANother Tool for Language Recognition*, and the parser generated is a LL(*) parser. `www.antlr.org`

The latter rule states that we must write the tokens `'partition'` and `'by'`, followed by one or more identifiers. However, this latter rule is optional.

```
defineExpr
    : patternElements+
    ;
patternElements
    : id ':' expr ';'
    ;
partitionBy    : 'partition'! 'by'! groupKeys
    ;
```

For example, the following:

```
define
    A: price > 200;
    B: price > A.price;
```

means that we will have two `pattern elements` A and B, and for an event to be evaluated to a `pattern element` it must pass its condition. It is important to note that the condition must always return a boolean value, i.e., either `true` or `false`, but that restriction check is not done in the grammar, rather it is done in the *Type Checking* phase - refer to section 3.2. Another check performed is the existence of repeated `pattern elements`, since they must be unique within a query.

The gist of the `partition by` is to have distinct patterns executing for every distinct fields provided. This is useful because otherwise a single pattern would be active. For example, in a stock market scenario this means different stock quotes can have each their own pattern if we specify a `partition by` according to the stock symbol, so a stock price tick for Apple does not interfere with a pattern for the stock price ticks from Tesla - refer to Figure 4.4, where the three bottom lines refer to individual patterns for each stock symbol, while the first line *'All Stocks'* would be the sequence of events evaluated had we not specified the `partition by`. When using the `partition by` we will also check in the *Type Checking* phase if all the fields specified belong to the *stream* or *window* we are performing the pattern matching over.

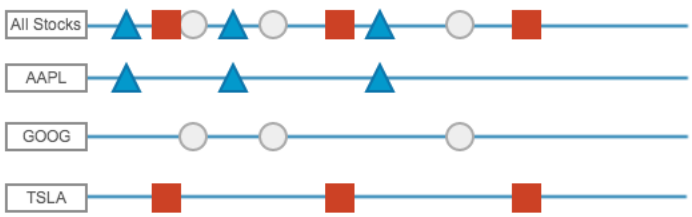


Figure 4.4: *Partition By* behavior example.

In the condition of a `pattern element` we can have two special token types. The first one is `prev`, which contains the previous event which matched an element in the pattern [10]. The other special token(s), is the name of each of the `pattern elements` defined. So, if we defined a `pattern element` A, then `A.price` will correspond to the price field of the last event matching the pattern

[10]This may be different from the last event which occurred in the stream, since the default match strategy is *relaxed* - this topic will be addressed in the next section concerning the `pattern` clause.

with the A `pattern element`. One caveat is that the pattern elements which depend on the `prev` operator are the last to be evaluated - refer to Table 4.1.

4.4.2 Pattern Clause

The `pattern` clause, as the name suggests, is where we specify the pattern we want to match. We will now analyze the operations available for pattern matching added to PQL. Do note that in the examples we will use the convention `pattern element`$_i$ to indicate the ith event received which matched the corresponding pattern element.

Sequence This is the main operation, since the objective of the new clauses is to define temporal relations between the events. This operator is denoted by a right arrow "`->`" - e.g. `pattern A -> B -> C`.

```
patternMatchExpr
    : patternMatchElement ('->' patternMatchElement)*
    ;
patternMatchElement
    : matchPolicy? retentionPolicy? (ALL)=>matchElementExpr timePeriod?
    ;
matchPolicy
    : STRICT^
    ;
retentionPolicy
    : LAST^
    ;
timePeriod
    : AFTER^ INT_LIT ID?
    | WITHIN^ INT_LIT ID?
    | ALL WITHIN INT_LIT ID?
    ;
matchElementExpr
    : matchElementOr
    ;
matchElement
    : cardinalityExpr? id
        -> ^(PATTERNELEMENT id cardinalityExpr?)
    | NOT cardinalityExpr? id
        -> ^(PATTERNELEMENT id NOT cardinalityExpr? )
    | '('! matchElementExpr ')'!
    ;
```

And We use this operator when we want every pattern element to occur, regardless of their order. For example, in `pattern A -> B and C and D`, after an A event, B, C and D must occur but the order may be (B, D, C), (D, C, B), etc. This should not be confused with the logical *and* operator, since each `pattern element` is a different event. However, the logical *and* may be applied in the condition of a `pattern element`.

```
matchElementAnd
    : matchElement (AND^ matchElement)*
    ;
```

Or With this operator we provide pattern elements that can occur, but only the first one will be matched. For example, in a sequence B_1, C_1, A_1 and with a pattern `A or B or C`; only B_1 will be added to the final result. As can be seen from the grammar, we give priority to the `And` operation.

```
matchElementOr
    : matchElementAnd (OR^ matchElementAnd)*
    ;
```

Cardinalities When we need to specify that an event matching a given pattern element must occur more than once, we will use this operator. Its syntax is equal to Esper's equivalent - `[2]A` means two events A must occur, while `[2:]A` means at least two events A must occur, and `A[2:5]` means between two and five occurrences of A.

```
cardinalityExpr
    : '[' INT_LIT ']'
    | '[' strt=INT_LIT? ':' end_=INT_LIT? ']'
    ;
```

Negation There may be cases where we need to test the non-occurrence of a given pattern element, so we will use this operator. For example, $!A$ means an event A must not occur. Do note that this operator can only be used in conjunction with the *And* operator, but this restriction is not done at the grammar level.

There are five optional keywords *strict*, *last*, *within*, *all within* and *after* that can be used in a *Sequence* - as can be seen in the grammar snippet for this element. These are used to specify *policies*. The first one - *strict* - is of help when we need the matching to be the exact match of the pattern, i.e., there cannot occur events other than the ones specified. One should be mindful that this only applies for a single sequence in the pattern, e.g., if we have a pattern `strict A and B -> C`, and a sequence of events $A_1,\ C_1,\ B_1,\ A_2,\ D_1,\ C_2$, the pattern will not match with events $(A_1,\ B_1,\ C_2)$ since C_1 occurs after the A_1, instead of a B event. However, this pattern does match with events $(B_1,\ A_2,\ C_2)$, and D_1 does not make the pattern restart like it happened with C_1, owing to the fact that the second sequence in the pattern is not a *strict* one.

The second keyword - *last* - represents a *Retention Policy*, i.e., it indicates that we want to keep the last occurrence of a particular pattern element[11], if there is no *last* clause, a *first* retention policy is implicit. For example, in a pattern `last A -> B` and a sequence of events $A_1,\ A_2,\ A_3,\ B_1$, the pattern will match with the events $(A_3,\ B_1)$. In the previous example, if we removed the *last* keyword, the pattern would match with the events $(A_1,\ B_1)$.

The last three keywords - *within*, *all within* and *after* - constitute *Time Policies*. The *within* has a similar behavior as in the CEP engines analyzed in Chapter 2, i.e., the pattern only matches if the events *fit* the provided time frame. Nevertheless, this policy is applied on a pattern sequence basis, so if we have a pattern with two contiguous sequences $(seq_n,\ seq_{n+1})$ and seq_{n+1} has a *within* time policy, then this condition must be met:

$$t_{last\ event\ seq_n} + \Delta t_{within} > t_{last\ event\ seq_{n+1}}$$

the timestamp of the last event of seq_{n+1} must be lower than the timestamp of the last event of seq_n plus the provided time frame. The *all within* policy is similar to the previous one. The idea behind this policy is to have the whole pattern match inside a timespan, and not just limit the timespan on a sequence basis. If we have N sequences then this condition must be followed:

[11] Currently, we can only apply this clause on a single pattern element, i.e., we cannot have a pattern such as: `last A and B -> C`, for instance.

Events	E_1	E_2	E_3	E_4	E_5	E_6
Events' price	210	220	230	225	228	205
PE attributed	A	$(A,\ B)$	$(A,\ B)$	A	A	C
PE chosen	A	B	B			C
prev'price		210	220	230	–	–
A's price		210	–	–	–	–
B's price			220	230	–	–
C's price						

Table 4.1: Query pattern elements and *prev* evaluation step by step. The event's occurrences are ordered from left to right, and the character "–" means there is no update in the value.

$$t_{first\ event\ seq_1} + \Delta t_{all\ within} > t_{last\ event\ seq_N}$$

Because of this, when using this policy we must specify it in the last sequence of a pattern. The *after* policy states that a particular sequence should only start *after* the given time units from the previous sequence. In a *strict* sequence if an event arrives before the time frame provided, the matching will break and restart, whereas in a *relaxed* sequence the event will simply be ignored:

$$t_{last\ event\ seq_n} + \Delta t_{after} < t_{last\ event\ seq_{n+1}}$$

In all these *Time Policies*, a time unit must be provided (milliseconds, seconds, minutes, hours, days, weeks or months).

Listing 4.3 contains a pattern matching query using the previously described operators, and in table 4.1 we provide a step-by-step evaluation for the values of these operators in the query - where the *Event's price* row corresponds to the *price* attribute six different events arriving in that order (from left to right). For every event we will also indicate the `pattern element` (PE) attributed and the one matched (chosen).

```
from stocks
define A: price > 200 and price <= 300;
       B: price > prev.price;
       C: price < A.price;
pattern A -> [2:]B -> C
select asymbol: A.symbol, aprice: A.price,
    bAvgPrice: B.avg(price), cAvgPrice: C.avg(price);
```

Listing 4.3: Pattern matching query example with the *prev* and pattern element operators

Finally, there are a few checks performed when analyzing this clause in the *Type Checking* phase. The first one is the need to use the `define` clause immediatelly before the `pattern`, since we must declare the `pattern elements` before using them in the pattern. We must also check whether we are using `pattern elements` which have been declared in `define`, otherwise an exception will be raised.

4.5 Our Algorithm

After analyzing the pattern matching clauses of several CEP engines (refer to Chapter 2), as well as the implementation details of some engines and languages, we designed our own algorithm for doing pattern matching queries. The decision for doing our own algorithm was due, on the one hand, to the complexity of some implementations (for instance RETE) and on the other hand, to the PKernel query plan design. Nevertheless, the algorithm has some similarities with both the approaches analyzed earlier.

In PKernel a query is ultimately composed of *operators* which will be connected in a dependency graph, such that the operations defined in the *operators* are executed in the correct order (refer to Chapter 3). Then, when we have the dependency graph created, each *operator* will generate the code according to its template, which will all be in the class generated for the query they belong to.

For the pattern matching queries, we have created two new operators - *OpDefinePatternElements* and *OpPatternMatch*. The former will be responsible for generating the code for the boolean expression of each pattern element, as well as the code for testing if the event(s) received match each pattern element. The latter operator will be responsible for generating the graph with the nodes corresponding to each operation. It also receives from *OpDefinePatternElements* a *List* data structure, where for every event there will be a *Pair*[12] containing the event itself, and another list containing the pattern elements which the event matched. Before analyzing all the node types, it is important to note that apart from the *Root* node, every node will *extend* an AbstractNode class that provides the abstract method *receives*[13]. This method will be used to pass the events between the different nodes. Furthermore, we have an *Enum* data structure containing the different return types of the *receive* method:

MATCH Returned when the event matched in the node it was sent to.

NO_MATCH Returned when the event does not match in the node it was sent to. If the node pertains to a *strict* sequence and if no other node returned a *MATCH*, it will make the pattern break and restart the evaluation (if a restart has not occurred already).

BREAK Returned when the event makes the pattern break. If it is the first time the event was evaluated by the network then the pattern will re-evaluate from the beginning, otherwise we will discard the event.

The available nodes for building the graph are:

Root This is the entry point for all the events coming from the stream. When an event arrives, it will be forwarded to the corresponding *Type* nodes by the order in which they are in the list. This behavior is similar to what RETE does with its *Root* node. The *Root* node is the most important node in the network, and it holds a lot of the logic of the algorithm.

Type These nodes represent each pattern element in the `define` clause, and will forward the events to every child nodes. The order in which they are

[12] A Pair is a data structure available in PKernel which holds two values (*first* and *second*) having any type, since it is also a *generic* class.

[13] A class diagram for all the nodes is available in Appendix D.

forwarded is from the last to the first sequences in the pattern. For example, if we have a pattern `A -> B -> A`, then the *Type* node corresponding to *A* will have two output nodes, one for each appearance in the pattern, which are stored in a list data structure in order to maintain the order defined in the pattern. In this example, illustrated in Figure 4.5, the first output node will be the node *SEQ #1* and the second output node will be *SEQ #2*, but the order in which the event is sent is first *SEQ #2* and, if there is no *MATCH* or *BREAK*, *SEQ #1*.

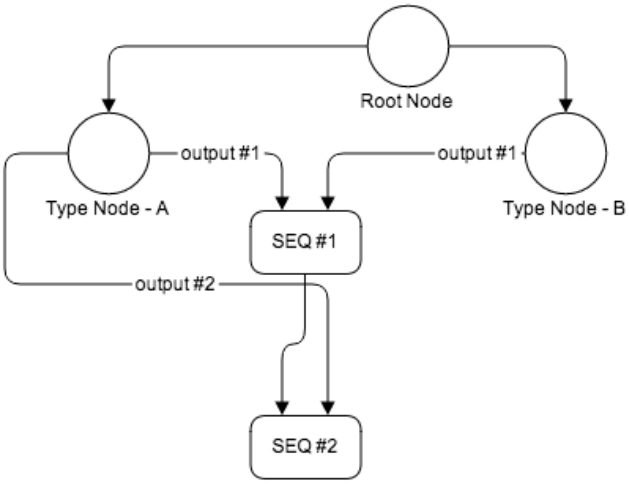


Figure 4.5: Type nodes' output node order example.

And This node represents the *And* operator described earlier. For every pattern element in the operator, there will be an input in this node - `A and B and C`, will have three inputs (the *Type* nodes *A*, *B*, *C*). Do note that a pattern such as `A and A` will only have a single *A type* node, if we want two events matching *A* in the *And* operator we need the to use cardinalities. Moreover, when there is a *Negation* operator in the sequence defined, e.g. `A and !B`, we have an additional node list for the nodes which represent this operator (*Not* nodes). When an event arrives to this node, we check if it came from a *Not* node, and if so we return a *BREAK*. Otherwise, we will check if we have already received from the node, store the event and return *MATCH* if we had not received, or return *NO_MATCH* otherwise.

Or This node represents the *Or* operator described earlier. Just like it happens in the *And* node, for every pattern element used with this operator there will be a corresponding input.

Not When we have a *negation* operator, we will create this node type. A *negation* operation can only be applied to a single pattern element, so we will always have the *Type* node corresponding to the pattern element as its input. Moreover, since this operator is always used within the *And* operator, we will always link this node to an *And* node.

Cardinalities As the name suggests, this node represents the *Cardinalities* operator. Since a cardinality operator is always related with a single

pattern element, this node has always a *Type* node as its single input. In cardinalities we have three types: exact (`[2]A`), interval (`[2:3]A`), and unbounded (`[2:]A`); nevertheless, all of these types will be represented by this node type, with all the different implementations being contained within it.

Seq This node always has two inputs (left and right), where the events received from the left must occur before the ones on the right. In a pattern with N sequences (`pattern` $\texttt{seq}_1$ `->` $\texttt{seq}_2$ `->` `...->` $\texttt{seq}_N$), there will exist $N-1$ *Seq* nodes. The first one will have $\texttt{seq}_1$ and $\texttt{seq}_2$ as its input, but the second one will have the first *Seq* node as its left input and $\texttt{seq}_3$ as its right input.

Single Seq This node exists when we have a pattern with only one sequence, e.g., `pattern A and B`. Since we have a single sequence, we do not need all the logic already implemented in *Seq* node, so we created this newer node for this specific case.

For the queries defined in listings 4.4 and 4.5, the graphs built will be figures 4.6a and 4.6b, respectively.

```
from stocks
define A: price > 200 and price <= 300;
       B: price > 300 and price <= 400;
       C: price > 400;
pattern A -> [2:]B -> [1:3]C
select asymbol: A.symbol, aprice: A.price, bAvgPrice: B.avg(price),
    cAvgPrice: C.avg(price);
```

Listing 4.4: Pattern matching query example

```
from stocks
define A: price > 200 and price <= 300;
       B: price > 300 and price <= 400;
       C: price > 400 and price <= 500;
       D: price > 500;
pattern A and B or C -> D
select dSymbol: D.symbol, dPrice: D.price;
```

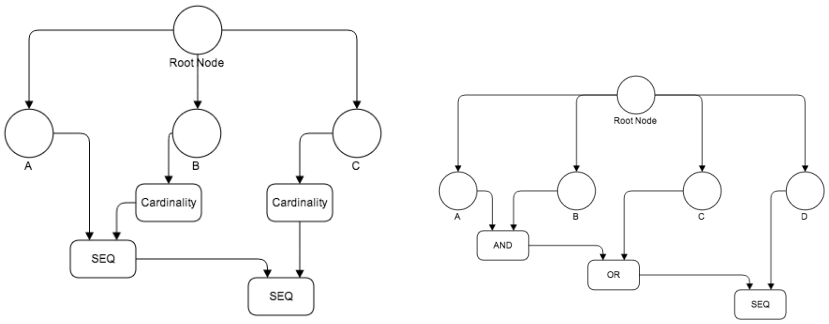
Listing 4.5: Pattern matching query example

The workflow of every node type is available in Appendix E.

4.6 Use-Cases

One of the most commonly known use-cases for Pattern Matching queries are stock patterns. There are a significant amount of such patterns, therefore we will only focus on two - *Double Bottom* and *Head and Shoulders*. Another important use-case is fraud detection, which is one of Feedzai's core businesses.

There is however one caveat. Since the stock prices are volatile and eratic, it is very difficult to make a pattern sufficiently permissive to accomodate the



(a) (b)

Figure 4.6: Generated graphs for two pattern matching queries.

changes in the stock prices. In our approach, we must assume there is some kind of sampling of the events, in order not to have all the changes being analyzed by the pattern. In [13][33], all the examples containing stock patterns also have identical approaches for the pattern definitions.

4.6.1 Stock Patterns

4.6.1.1 Double Bottom

A *Double Bottom* pattern occurs when we have a pattern resembling the shape of a 'W' - refer to Figure 4.7. When this pattern arises, it means that the stock will start an uptrend. Let us translate this pattern into a PQL query, which

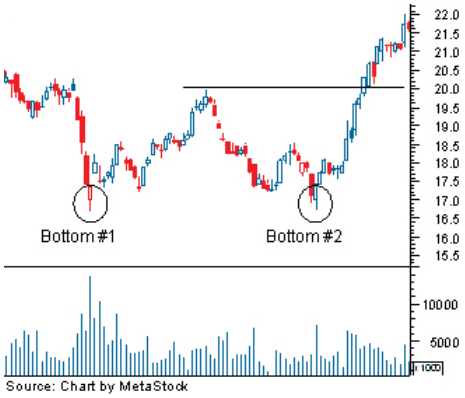


Figure 4.7: Double Bottom example. Source: `http://www.investopedia.com/university/charts/charts4.asp`

will output a result when the pattern occurs. The final query can be found in Listing 4.6. The idea behind the final query is to constantly compare the current stock price with the previous one in order to get the expected 'W' shape. So, we needed four conditions, each representing a trend - upward or downward. When we want an upward trend that means the previous price is lower than the current, this is translated by the following condition of a `pattern element`:

`price >= prev.price;`. For a downward trend the condition is the opposite one, i.e., the previous price must be greater than the current one. Additionally, to signal the *buy* event, which is the objective of this pattern, we needed a new condition to know when we have reached a price greater than the previous maximum value - this is depicted in Figure 4.7 by the horizontal black line. This condition is translated in the query by the *buy* `pattern element`, where we compare the current price with the last event of the first upward trend which occurs after the *Bottom #1*.

In the `pattern` clause we define all the upward and downward trends with a cardinality of at least one event - `[1:]`. The `first_downward pattern element` needs two events because the first event will always match the condition - its `prev` event is always `null`. So, as long as an event keeps passing the condition of the trend we will keep appending it to the list data structure of that `pattern element`. When the pattern finally matches, we can get the event corresponding to the *Bottom #1* by getting the last event of the `first_downward pattern element` - this corresponds to `bottom_1` in the `select` clause. Since it is a list, we can use the methods available in PQL to query it: `get, avg, sum, stddev, max, min, maxBy, minBy, map` and `contains`.

```
double_bottom =
    from stocks
    define
        first_downward: price <= prev.price;
        first_upward: price >= prev.price;
        second_downward: price <= prev.price;
        buy: price >= first_upward.price;
        -- we need to give priority to the previous pattern element
        second_upward: price >= prev.price;
    partition by symbol
    pattern [2:]first_downward -> [1:]first_upward ->
        [1:]second_downward -> [1:]second_upward -> buy
    select bottom_1: first_downward.last().price,
        peak: first_upward.last().price,
        bottom_2: second_downward.last().price, buy_price: buy.price;
```

Listing 4.6: Double Bottom query

4.6.1.2 Head and Shoulders

The *Head and Shoulders* stock pattern is used to signal a sell order. The pattern finds three peaks, where the first and third have similar values, and the middle one has a value greater than the first peak - refer to Figure 4.8. Listing 4.7 contains the PQL query which translates this pattern. Like the previous stock pattern, we use the `prev` and the last event of a `pattern element` quite extensively, the main reason being the need to compare the current values with the ones obtained earlier. For example, in the upward trend where we might reach the `head`, we must make sure the current price is greater than the price of the first shoulder - translated by this condition: `upward_head: price >= prev.price and price >= first_upward.price;`

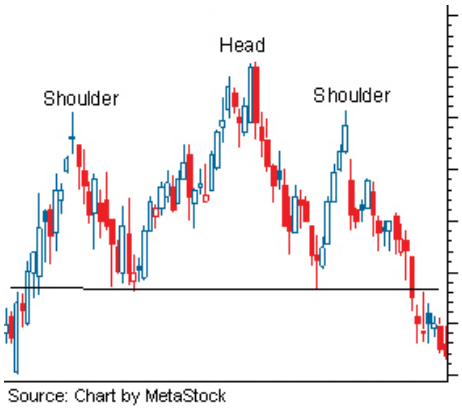


Figure 4.8: Head and Shoulders example. Source: `http://www.investopedia.com/university/charts/charts2.asp`

```
head_shoulders =
    from stocks
    define
        first_upward: price >= prev.price;
        first_downward: price <= prev.price;
        upward_head: price >= prev.price and price >= first_upward.price;
        -- we need to give priority to the previous pattern element
        upward_shoulder: price >= prev.price;
        downward_shoulder: price <= first_upward.price;
        -- we need to give priority to the previous pattern element
        downward_head: price <= prev.price;
        upward_shoulder_2: price >= prev.price and price <= upward_head.price;
        sell: price <= downward_head.price;
        -- we need to give priority to the previous pattern element
        downward_shoulder_2: price <= prev.price
    partition by symbol
    pattern [2:]first_upward -> [1:]first_downward ->
        [1:]upward_shoulder -> [1:]upward_head -> [1:]downward_head ->
        [1:]downward_shoulder -> [1:]upward_shoulder_2 ->
        [1:]downward_shoulder_2 -> sell
    select first_shoulder: first_upward.last().price,
             head: upward_head.last().price,
             second_shoulder: upward_shoulder_2.last().price
             sell: sell.price;
```

Listing 4.7: Head and Shoulders query

In the `pattern` clause, we also have all the trend `pattern elements` with an unbounded cardinality of at least one. Similar to the previous query analyzed, the first trend `first_upward` needs two events because the first event reaching the pattern will always match the condition - `prev` event is `null`. As stated in the previous query, since these trends have cardinalities when we access one of these `pattern elements` it will be a list, and we can apply the available methods for this data structure.

Finally, in the condition for the `sell pattern element` we are only comparing its price with the lowest price reached between the `head` and the second `shoulder` - which corresponds to the last event of the `downward_head pattern element`. So, we are assuming the black line portrayed in Figure 4.8 - usually

called *neckline* - in our case is only equal to that last minimal price, and is not created also based on the lowest price between the first `shoulder` and the `head`.

4.6.2 Fraud Detection

One of the most commonly known rules for fraud detection in card transactions is the check for two transactions in a relatively short period of time, and in quite distant locations. This rule can easily be translated into a PQL query - refer to Listing 4.8.

```
fraud = from transactions
    define
        first: true; -- any transaction
        fraud: distance(location, first.location) > LIMIT;
    partition by card_number
    pattern first -> fraud within 1 hour
    select first_loc: first.location, fraud_loc: fraud.location,
        time_span: fraud.timestamp - first.timestamp;
```

Listing 4.8: Fraud detection query example

The `distance` is an UDF[14] to calculate the distance between the two locations provided, and the `LIMIT` is a PQL immutable variable that is defined outside the query.

The result of this query will be a stream of tuples containing: the location of the first purchase, the location of the fraudulent purchase and the time interval between these two purchases.

4.7 Current Limitations

As of this writing, there is still no support for multi-stream pattern matching, since a query can only receive events from one stream, or receive from more than one stream joining them. It is still unclear if we will really need this feature, however an elaborate way to do it would be to forward the events from the different stream into a single one with the use of the clause `insert into`. This is just a workaround, and if this operation proves to be a requirement, then a possible syntax would be to declare all the streams in the `from` clause with a mandatory alias in order to refer to that event in the `define` clause, e.g. `from s in stocks, n in news`.

Another limitation is that the sequence operator "`->`" is not a freely used operator, i.e., it is not at same level as an *and* or an *or* operators, and thus we cannot create a pattern such as: `A -> (B -> C) or (C -> A)`. This was not elicited as a requirement, and there are still cases where we can convert to a similar pattern with the current available operators. However, to surpass this limitation we will have to change not only the grammar, but the algorithm itself.

[14] *User Defined Function.*

4.8 Benchmark

With the implementation finalized, we began idealizing and implementing a benchmark for the new capabilities. In order to have a better grasp of the actual performance, it was decided that we needed to run the same benchmark on other engines. The engines evaluated were: Esper and Siddhi. The Streambase engine was not comprised in this benchmark since its pattern matching capabilities are not equivalent to ours, nor Esper's and Siddhi's. In Streambase, when we specify a pattern, for every event reaching its inputs we may start a new pattern execution, as well as match the already started ones - refer to 2.2.

For this benchmark, we created sixteen queries for testing each operation - `and`, `cardinalities`, `pattern conditions`, and so forth. Furthermore, an additional test was made where all the queries were registered in the engine. The basis for this test was to investigate the scalability of each engine, and it also better represents a real-world environment where we are receiving a large number of events and they are evaluated by different queries.

Finally, each of these seventeen tests were repeated thirty times in order to get a better statistical significance.

4.8.1 Setup

- Hardware
  - 8x Intel Core i7-3770K CPU @ 3.50GHz
  - 15.4 GiB RAM
- Software
  - Ubuntu 13.04
  - Java HotSpot VM - 1.7.0_45
  - Scala - 2.10.3
  - PKernel - 13.1.0
  - Siddhi - 2.0.0
  - Esper - 4.11.0
- Stream Definition

```
stocks = Stream(
    timestamp:  long,
    symbol:     string,
    index:      string,
    price:      double
);
```

- Dataset
  - 1 million events - 25.2 MB
  - 5 million events - 130.6 MB

  - 10 million events - 261.9 MB
  - 20 million events - 534.4 MB

- Measures
  - Latency
  - Time
  - Throughtput

4.8.2 Result Analysis

The result of the test with all the queries registered is shown in Figures 4.9 and 4.10. Besides the total time spent processing the events and the throughput, another important measure is latency, i.e., the time spent processing each individual event. For calculating the latency each test was run five times with the 20 million event dataset. The latency results for the test with all the queries registered are shown in Figure 4.11.

In both the Figures, we clearly observe that PKernel has the best performance out of the three engines being benchmarked. Moreover, if we analyze the PKernel throughput from Figure 4.10, we easily notice that it is actually increasing the more events we inject, meaning we did not reach its saturation point. This contrasts with the two other engines, where their throughput remains almost constant, especially Siddhi. From Figure 4.11, we also notice PKernel gets the lowest latency value for all percentiles except 50^{th} and 99.9^{th}, where Esper has the best values. Furthermore, Siddhi consistently achieves the highest latency values for all the percentiles analyzed.

In table 4.2, we observe that the standard deviation obtained for the Esper engine is very high when we injected the datasets with 10 and 20 million events, reaching 38 and 84 seconds, respectively.

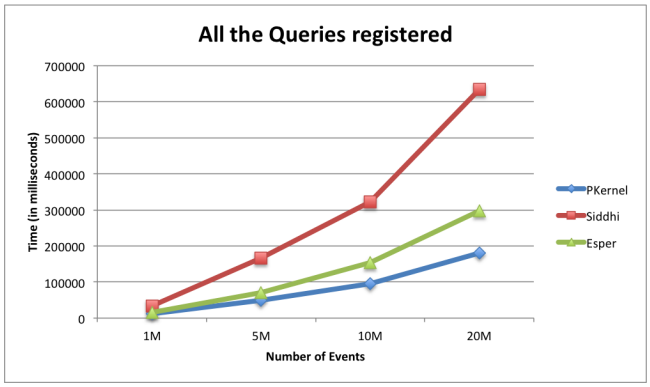


Figure 4.9: Time spent processing the events.

The results for each individual query can be found in Appendix G.

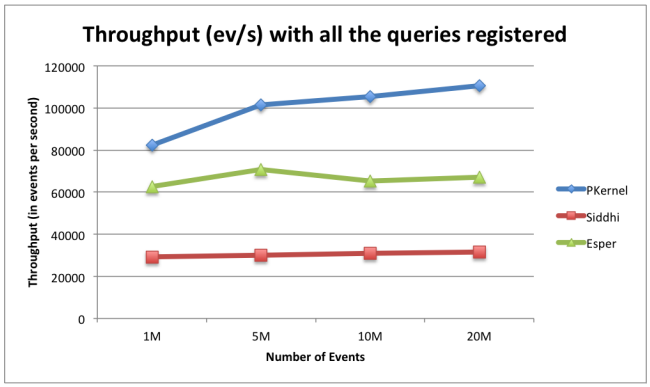


Figure 4.10: Throughput achieved.

Engine	50th	75th	95th	99th	99.5th	99.6th	99.7th	99.8th	99.9th	99.99th
PKernel	8009.400	9434.400	11498.600	12955.000	13863.001	14621.001	16689.600	20754.001	60503.001	138474.602
Siddhi	33323.600	41040.200	75033.000	118232.000	161945.802	170986.802	181246.006	193976.001	212729.801	311434.400
Esper	7562.200	11787.600	17075.000	20419.800	21939.801	22610.000	23617.201	25365.600	30813.001	180705.612

Figure 4.11: Latencies for each of the specified percentiles, in nanoseconds.

Engine	Events	Throughput (ev/s)	$\overline{x}_{milliseconds}$	$\sigma_{milliseconds}$
PKernel	1M	82361	12141,63333	133,8116235
	5M	101584	49220,03333	915,0722784
	10M	105469	94814,16667	1901,574395
	20M	110587	180852,6	3358,939114
Siddhi	1M	29272	34161,93333	772,8226336
	5M	30119	166006,4333	5162,180151
	10M	30943	323170,3333	6177,103329
	20M	31506	634780,7	4225,995326
Esper	1M	62689	15951,63333	325,2181743
	5M	70858	70563,6	2904,600757
	10M	65171	153440,7333	38297,01129
	20M	67072	298185,8667	84786,37186

Table 4.2: Throughput, mean time and standard deviation of time spent.

4.9 Tests

In this project a Test-Driven Development (TDD) approach was used (refer to 6.3.2). For validating the pattern matching clauses and operations, we implemented several unit tests for each of these features using the module *test-framework* of PKernel. In total, the number of system tests implemented was 8, with a total of 35 PQL queries. In all the 35 queries, we provide the events constituting the input for the *stream(s)*, and for each of the queries we declare the expected output events at each timestamp.

Chapter 5

State-Based PQL

State-Based programming is a new capability of PQL, where the focus is given to instances of entities, and their states. The basic unit in Event Processing is, of course, the events themselves. However, this approach does have its limitations, perhaps the most flagrant one being event correlation. Over the years, new clauses were added to resolve this issue, the most well known of them being the `group by` clause. *Pattern Matching* and *joins* also play a major role in the event correlation field.

The main objective of *State-Based* programming is to take event correlation even further, and give the users the ability of treating events as instances of a higher well defined structure, which we will call *entities*. These *entities* are similar to classes in the OOP programming paradigm, and in a way, they can be thought of as an enhanced `group by`. In a `group by` clause, we aggregate events with the same values for the key fields, but there is no way of specifying a life-cycle for each of these events. In *State-Based* programming we will accomplish this by specifying the states an instance might be in, as well as provide ways of getting more information from these states, such as: *timers*, in order to know how much time an instance is in a given state; *counters*, which will correspond to the number of times an instance reached a state; and finally *members* which will be new fields - similar to instance variables.

In the next sections, we will give a more detailed description of *State-Based* programming, also providing an overview of its implementation in the overall architecture of PKernel. Finally, in the last section, we will present some use-cases.

5.1 Drafts

We started the development of this new capability by doing a thorough research into this field. However, there is very little developments in this area - as can be depicted by the lack of information we presented in chapter 2. The main source of contributions to the features and requirements were the TIBCO Business Events Data Modeling module (refer to 2.6.2), and a use-case provided by Prof. Paulo Marques for a telecommunication company network management system. In this use-case, there were several networks and routers, and we wanted to know the state of a connection in the whole network (idle, connected, discon-

nected). An important feature was the possibility of knowing how many times a connection remained in a given state, as well as how much time it stood in that state.

While doing our research, we also started developing drafts for the definition of the entities and their states. For the basis of the language draft, we took a look at some Domain-Specific Languages (DSL) literature - mainly by the well-known computer scientist Martin Fowler[43][4]. Furthermore, we also tried to make the new additions to the language coherent with the clauses that already existed in PQL.

All the drafts produced can be found in Appendix H.

5.2 Requirements

The functional and non-functional requirements for the *State-Based Programming* are available in Appendixes B and C, respectively. They were defined in the same manner as the ones for *Pattern Matching* (refer to 4.2), i.e., using *user stories* and the *MoSCoW* method [10][11] for classifying the priorities of each requirement.

5.3 Entity Definition

Based on the drafts developed and the requirements elicited, a final formal entity definition was decided. The complete grammar definition is available in Appendix I.

There are nine possible constructs to define an entity, and we must respect their order. The nine constructs are:

create from This is the first construct and it is here we specify the input stream, as well as the fields to aggregate the events over - translated by the `groupKeys` element in the grammar definition shown below. This construct is mandatory. Do note that only *streams* are allowed to be the input for an entity - *windows* and *dictionaries* are not permitted at the moment.

```
createFrom
    : CREATE FROM id SEMICOLON
    | CREATE FROM id ON groupKeys SEMICOLON
    ;
```

If we have a stream named stocks, this construct could be defined as: `create from stocks on symbol;`. Notice in the grammar definition that the originating stream is a single `id`, meaning that at this moment we can only create instances from a single stream.

states After the previous construct we must provide a list of states for the instances of the entity. The grammar definition follows:

```
states:
    STATES '{' state (',' state)* '}'
    ;
state
    : id
    | id TIMER
    | id COUNTER
    | id TIMER COUNTER
    | id COUNTER TIMER
    ;
```

We must specify at least one state, and this state can contain a timer, a counter, or both. As a convention, if we want to access the timer of a state the identifier is created as follows: *state*`_timer`. The same applies to a counter.

If we specify repeated states, then an exception will be raised at the time of the compilation. Furthermore, we must make sure the name of the state does equal one of the fields of the *stream*.

start at After declaring the possible states, we can specify the state in which all the instances start - e.g. `start at first_state;`. This clause is optional, so if we do not use it, a default state - `START` - is chosen as the first one. In the *Type Checking* phase, we will check if the provided state does belong to the states declared before.

```
startAt
    : START ATT id SEMICOLON
    ;
```

end at Similar to the previous construct, here we specify the last state of the instances. This is also an optional construct, and the default state is `END`. Furthermore, we also check if the state provided is a valid one.

```
endAt
    : ENDACTION ATT id SEMICOLON
    ;
```

timers and counters The timers and counters specified in the `states` construct only apply to a single state. In this construct, we can compose more complex timers and counters by defining a path, i.e., a sequence of (valid) states the instance must go through - separated by the characters '`=>`', as shown in the grammar rule below. Although not perceptible in the grammar, we have the possibility of using a wildcard - using the underscore character '_' - when we are not interested in the state the instance passes through. Furthermore, we can mark them as being `global`, which indicates that there will be a single timer or counter for the whole entity. This behavior is similar to the static variables in OOP.

At the moment, the non-global timers are not cumulative, i.e., each time an event occurs the value of the timer contains its start and end timestamps. Notwithstanding, the counters, both global and local, are cumulative.

```
timerOrCounter
    : TIMER name=id id ('=>' id )+ SEMICOLON
    | GLOBAL TIMER name=id id ('=>' id )+ SEMICOLON
    | COUNTER name=id id ('=>' id )+ SEMICOLON
    | GLOBAL COUNTER name=id id ('=>' id )+ SEMICOLON
    ;
```

members Members can be thought of as additional instance variables. Just like the previous construct, we can mark a member as `global`. Do note that once a member is defined, we cannot change its type in the actions of the constructs *transition* and *expires*. This check is done at compile time, so a `Runtime Exception` will be avoided due to incompatible types. Also, we must check whether there are no duplicate members, or its name is already associated with a *stream* field.

```
member
    : GLOBAL MEMBER ID '=' expr SEMICOLON
    | LOCAL? MEMBER ID '=' expr SEMICOLON
    ;
```

define This construct is similar to the `define` clause added in the *Pattern Matching* (refer to 4.4.1), but without the `partition by` clause since we already have that information in the `create from`. It will be responsible for creating the `pattern elements` which will be used in the next construct - *transitions*.

transitions To express a transition between one state to another we will use this construct. One of the key features in the transition is the ability to express its condition using the pattern matching capabilities implemented earlier. The condition for the transition is stipulated after the `when` token. We can also have *wildcard* transitions, i.e., when we specify a *from state* named '_', we will create transitions from each individual state to the one specified after the `to` token. Both the *from* and the *to* states must have been defined in the `states` construct.

```
transition
    : TRANSITION FROM from=id TO to=id
    WHEN patternMatchExpr
    (DO transAction* ENDACTION)?
    ;
```

If we look closer at the grammar definition above, we can see the rule `transAction*`. This is used to define the actions we will execute upon the transition, and currently must be present inside a `do ...end` block[15].

Let us look at an example to get a better idea of the syntax:

```
transition from buy to sell
when good_news -> bad_news
do
    post to alerts (index, stock, "SELL!")
    alert_type = 4; -- member assign
end
```

The `post` action was one of the *actions* we established as possible in this context. What it does is send an event to the stream indicated and the fields or literals chosen - in the example above we have a string literal that will be included as a field of the event. Under the hood, there are a few checks being made, mainly related with the types of the fields provided which must match the ones from the receiving stream. If the type expected is not equal to one given but they are *promotable*, then we will change the

[15] If you are familiar with the Ruby programming language, you may be familiar with this syntax

expression to a *cast expression*, thus changing the type to the expected one.

The other action implemented was the `assign`, which enables us to change the value associated with a `member`. However, the new value type must match the type associated with the `member` upon its creation. We also check if the types are *promotable* if they do not match, i.e., we can promote an *integer* value to a *long* value, however, we cannot promote a *string* into an *integer*, nor the other way around.

Finally, we must keep in mind that the order in which we specify the transitions matters, since that will be the order in which we will evaluate the patterns. So, if we have two transitions *t1* and *t2*, and we want to give priority to the pattern in *t2*, then we must write the transition *t2* before *t1*.

expires This construct is a special type of a transition, where we do not specify any pattern, we only provide a *timeout* after which the instance changes its state. An example of its usage follows: `expire state1 after 10 min to state2`.

```
expire
    : EXPIRE from=id AFTER x=INT_LIT units=id TO to=id
        (DO transAction* ENDACTION)?
```

Similar to the *transitions* construct, we can declare actions to be performed when the `expire` occurs. The actions available are also the same - `post` and `assign`.

One should be mindful that we can only have one expire per state, the main reason being that since we are registering a timeout, the only one which may occur will be the one with the lowest timespan.

5.4 Implementation

We will now take a closer look at the implementation of the *State-Based Programming* into PQL. This new capability involved more changes to PKernel than the implementation of *Pattern Matching*. The main reason was the creation of new types and new clauses, contrasting with *Pattern Matching* where its scope was limited to queries.

The first change performed was the addition of a new high level definition: the entity. Previously, there were only two of these: the stream definition, and the query definition. In an entity definition parsing, we only have the overall type after the `member` statements, since we inherit the fields from the originating stream, and we also add a field for every *timer*, *counter* and *member* defined. This contrasts with the previous two definitions, where we have the type immediately: in the stream definition we provide the schema for every event in that stream which includes the name of the field and its type; and in the query definition, we always have an input which must be already registered in the *environment*:

```
-- stream definition example
stocks = Stream(timestamp: long, symbol: string, price: int);
-- query definition example
query = from stocks select symbol;
```

As stated in Chapter 3, a query is ultimately composed of *operators* to build a dependency graph. When defining a stream, for example, we will create an *operator* called *OpInputStream* containing the stream name and the type of its events - which will always be a *tuple*. In the entity case, we needed a new *operator* which would be used as input in queries. The new *operator* created was the *OpEntity*.

The *OpEntity* operator has all the information about an entity. At first, we thought about creating different operators for every construct (`create from`, `states`, etc) and *link* them sequentially, but decided it would be much simpler to have all the components in one place rather than evaluating every operator one by one. Furthermore, it will make it easier to generate the resulting Java code, since we have all the information needed in one place and there are no complex mechanisms to deal with possible dependencies between each construct. The *OpEntity* contains the following components:

source This will be the source operator for the entity. If the `create from` specifies the stream `stocks` defined in an example above, then it will be the *OpInputStream* of that stream.

tyTuple This will be the type which will represent an instance of an entity, in this case it will always a tuple type. As stated earlier, it will contain all the fields from the source stream plus all the *timers*, *counters* and *members* defined.

entity Contains the entity name.

states Contains a list of all the states which were defined for this entity.

startAt The starting state.

endAt The final state.

timers This is a list of all the timers for this entity. If none were defined, then this will be empty.

counters This is a list of all the counters for this entity. If none were defined, then this will be empty.

members This is a list of all the members for this entity. If none were defined, then this will be empty.

patternElements Contains all the `pattern elements` defined, as well as the *expressions* for the conditions.

transitions A list containing all the transitions, by the order in which they were defined. In the code generation phase, each transition will have its own method and its own *RootNode* (refer to section 4.5).

expires A list containing all the expire transitions, by the order in which they were defined. In the code generation phase, each expire will have its own method which may be called if the timeout does occur.

deps The operator dependencies for the entity definition. If we have a condition of a `pattern elements` which requires the average of a field in a stream, then this average will be an *operator* and we must be informed of changes when we check the condition.

depsTransitions This is similar to the previous component, but we are only interested in the *operators* *OpPatternSelectList* (refer to section 4.5). So, for each transition we will have a list of these *operators*. The motivation for this component was the need to keep these *OpPatternSelectList* operators out of the dependency graph, because they will have as their source operator the same *OpEntity* which also depends on them, creating a circular-dependency.

```
stocks = Stream(timestamp:long, symbol: string,
    index: string, price: int, opinion: string,);

entity StockItem {
    create from stocks on symbol;

    states {
        buy timer,
        hold timer,
        sell timer
    }

    define
        opinion_buy: opinion == "buy";
        opinion_hold: opinion == "hold";
        opinion_sell: opinion == "sell";

    transition from _ to buy
    when opinion_buy

    transition from _ to hold
    when opinion_hold

    transition from _ to sell
    when opinion_sell
};
```

Listing 5.1: Example of an entity definition

Listing 5.1 is an example of an entity definition for a stock symbol. We have three states which depict an analyst's opinion on that specific stock: `buy`, `hold` and `sell`. Also, each of these states has a timer associated with it, so we can keep track of how long it stays on the given state.

5.4.1 Code Generation and Instance Workflow

Upon filling the information for the *OpEntity* operator in the *Plan Builder* phase (refer to Chapter 3), we will generate the code for receiving an event from the source stream, and output the entity instance related with the event. This behavior is depicted in the activity diagram in Figure 5.1. We start by receiving the event from the originating stream. With the fields provided in the `create from` we build a *tuple* containing the values of those fields, which will be the *key*. Every entity has a *map* data structure which holds, for every *key*, the last *tuple* for that instance. After we build the *key*, we query the previously mentioned *map* to get the instance. If none is found, then this is the first event for that instance and we must initialize all the timers, counters and members, set the current state to be state provided in `start at`, and register

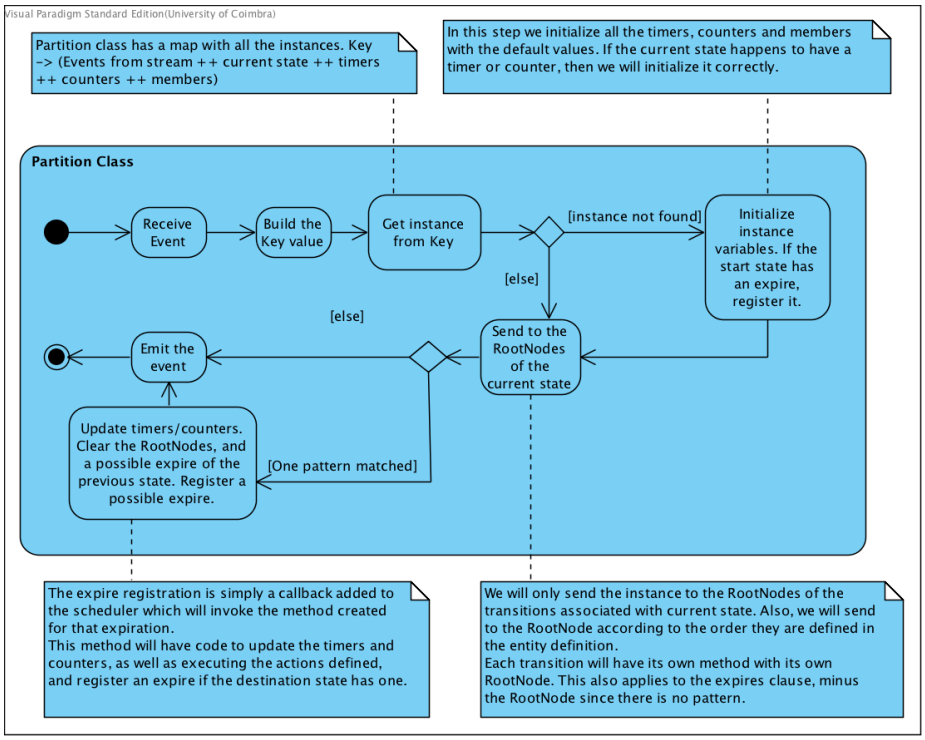


Figure 5.1: Activity diagram representing the behavior in an entity partition.

the expire *callback*. If the instance already existed, then we do not need to initialize anything. The next step is sending the instance to every *RootNode* of transitions that start at the current state, while keeping the order in which they were defined. If one of these *RootNodes* returns a value, then it means we have matched the whole pattern and the transition must be performed, i.e., we update the current state, the timers and counters, and execute the actions stipulated, cancel any expire transition registered, and finally register a expire transition if one was defined, in this order.

5.4.2 Query Interaction

With the entity definition dealt with, we must now give the ability of interacting with it in the queries. The type of an entity is a `TyEntity`. This is a type created in the compiler (it will not be translated to any type in the code generation) which contains:

- A `TyDict`, which represents a dictionary, where the keys are the fields provided in the `create from`, and the value is the type of the instance.
- A `TyTuple`, which represents a *tuple*, containing the names and types of the global timers, counters and members, if any.

To make it easier, all the operations available for the dictionaries (`TyDict`) can also be applied to the entities: we can use it in the `from` clause, we can access a specific instance by its key (`entity[key]`), use the clause `select` and

`where` over it, and so forth. The entity also inherits its shortcomings. With a dictionary we do not have any historical track, we only have access to the current instances. To overcome this, we create a new operator for dictionaries called *OpDictUpdated*. What this does is it creates a *stream* of the updates being made to the dictionary - inserts, deletes and updates. Since it is a *stream*, we can create *windows* over it, e.g. `entity.updated()[10 min]` creates a window of the changes made in the dictionary in the last 10 minutes.

Additionally, we can access the *global* variables of an entity similar to static variables in the Java programming language - `entity.global_variable`. To achieve this we created a new operator - *OpEntityGlobalGet* - which contains three fields:

query The name of the query where we call the global variable.

source The *OpEntity* operator corresponding to the entity for which the global variable belongs.

field The name of the global variable.

We will now present some query examples with some common operations. They assume we are using the entity defined in listing 5.1.

- Current number of instances in each state

```
instance_count_by_state =
    from StockItem
    group by state
    select count: count();
```

- Time spent at state *buy* by all instances currently in that state

```
total_time_by_state =
    from StockItem
    where state == "buy"
    select total_time: sum(buy_timer);
```

- Time spent at state *buy* by all instances in the last 24 hours

```
total_time_by_state =
    from StockItem.updated()[1 day]
    where state == "buy"
    select total_time: sum(buy_timer);
```

- A *continuous value* of the current state for stock `AAPL`

```
current_state_aapl = StockItem["AAPL"].state;
```

In the case of a timer, we have additional operations available. In PQL, we defined the timer with the type *TyTimer*, which can be thought of as a *tuple* with two elements, or a *pair*. The two elements constituting a *timer* are the `start` and `end`. There are three operations which can be called on a *timer*:

start Returns the start timestamp. If the *timer* has not started yet, then it will return the default value `0`.

interval Returns the difference between the `end` and the `start`. If the *timer* has not ended yet, then it will return the default value `0`.

end Returns the end timestamp. If the *timer* has not ended yet, then it will return the default value `0`.

As an example:

```
aapl_buy_timestamp = StockItem["AAPL"].buy_timer.start();
```

The previous query `aapl_buy_timestamp` would give us the timestamp of when the `AAPL` stock entered the `buy` state.

5.5 Use-Case

5.5.1 Shipping Company

In this use-case, we will model the process of an order in a shipping company. We have defined six states: *make_order, payment, shipped, arrived_destination, order_cancelled* and *lost*. All these states, except *lost*, have transitions defined. In the case of the *lost* state we will transition to it through the `expires` timeout, where we will consider a package as being lost if the shipping is taking more than two weeks.

Moreover, we have a stream `orders_received` which will receive events when an order has been successfully received by the customer. Besides the mandatory timestamp, order id and client id, we also have a field called `number_warehouses` which represents the number of warehouses where the package stayed until it reached the customer's destination. The posts to this stream are being made in the transition `shipped` to `arrived_destination`.

Since we are interested in receiving alerts of lost packages, we created a new *stream* `lost_alerts` where we receive information on the time at which we reported the order as being missing, as well as information on the order and respective client.

```
orders = Stream(
    timestamp:  long,
    order_id:   long,
    client_id:  long,
    type:       string,
    success:    boolean
);

orders_received = Stream(
    timestamp:  long,
    order_id:   long,
    client_id:  long,
    number_warehouses: int
);

lost_alerts = Stream(
    timestamp:  long,
    order_id:   long,
    client_id:  long,
);

entity Order {
    create from orders on order_id;

    states {
        make_order,
        payment timer, -- time waiting for shipping
```

```
    shipped timer,
    arrived_destination,
    order_cancelled,
    lost
}

global counter shipments_lost shipped => lost;

define
    order: type == "make";
    payment: type == "payment" and success == true;
    shipment: type == "shipped";
    warehouse: type == "warehouse";
    arrived: type == "arrived";
    cancelled: type == "cancelled" or (type == "payment" and success == false);

transition from START to make_order
when order

transition from make_order to payment
when payment

transition from payment to shipped
when shipment

transition from shipped to arrived_destination
when warehouse[1:] -> arrived
do
    post to orders_received (timestamp, order_id, client_id, warehouse.count(), shipped_timer);
end

transition from _ to order_cancelled
when cancelled

expire shipped after 2 weeks to lost
do
    post to lost_alerts (timestamp, order_id, client_id);
end
}
```

Chapter 6

Work Plan and Methodology

This chapter explains the planning made for the first semester and its changes and delays, as well as planning for the second semester. Furthermore, we introduce the methodologies used in the development of this project.

6.1 First Semester

In Figure A.1 of Appendix A.1 we can see the initial high-level plan for the work to be done during the first semester. However, during the actual work there were a few changes to this planning, as shown in Figure A.2. There are a couple of reasons for the delays. First, the PKernel engine is a complex system with a lot of different modules. It is also written primarily in the Scala programming language[16] and in a functional programming way. We already had some experience with Scala, but not so much on its functional programming side. The MOOC[17] *Functional Programming Principles in Scala*[18] hosted by the creator of Scala, Prof. Martin Odersky, was a great help in overcoming these hurdles.

6.2 Second Semester

The plan stipulated for the second semester is available in Appendix A.2. There were no major changes in the planning made. The only significant one was the *Benchmarking Entities* task which was not performed at this time.

[16] `http://www.scala-lang.org/`
[17] Massive open online course
[18] `https://www.coursera.org/course/progfun`

6.3 Methodology

6.3.1 Scrum

In both semesters an agile software development methodology was adopted - more specifically Scrum. Scrum is "framework within which people can address complex adaptive problems, while productively and creatively delivering products of the highest possible value. (...) The Scrum framework consists of Scrum Teams and their associated roles, events, artifacts, and rules." [59, p. 3].

- Roles
  - Product Owner: Paulo Marques (CTO)
  - Scrum Master: Ricardo Ferreira
- Events
  - The *sprints*.
- Artifacts
  - The monthly reports sent to the supervisors.

At Feedzai there is a weekly status meeting, deviating from Scrum principles where there exists a *Daily Scrum* - a daily 15 minute meeting to synchronize the work done in the previous day, as well as the plan for the next 24 hours [59].

6.3.2 Test-Driven Development

Test-Driven Development (TDD) is a software development process in which the developers first write a test case for a given feature or improvement, and only after this step start its implementation, writing the minimum amount of code in order to pass the test. Once the developed code is accepted the code, the developer may *refactor* the code. All these steps are illustrated in Figure 6.1.

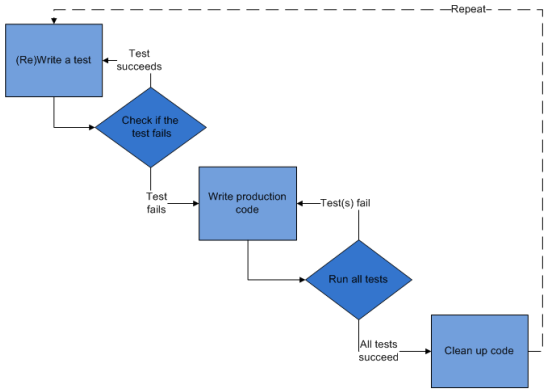


Figure 6.1: Test-driven development workflow - taken from `https://en.wikipedia.org/wiki/File:Test-driven_development.PNG`.

In this work, for the pattern matching capabilities we started by creating tests with some simple pattern queries, and their corresponding expected results. At the end of the implementation phase we created more complex tests.

6.3.3 Git

The version control system used throughout this work was Git[19], using the branching model already adopted at Feedzai (refer to Figure 6.2)[20]. In this

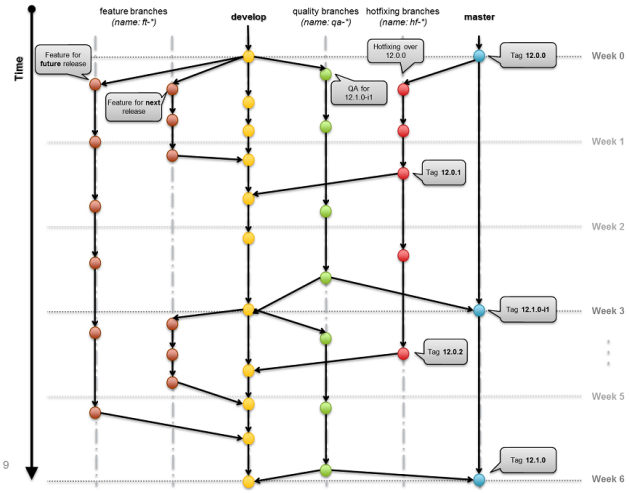


Figure 6.2: Git branching model used at Feedzai.

branching model, there exist five main branch types. The *develop* branch is the main branch for development, and contains code which may not be ready for production. The *feature* branch is used to implement new features isolated from the other branches, and once the feature is complete the branch will merge back into the *develop* branch. The *quality* branch is only only for quality assurance related code. *Hot Fixing* branches are created for bug fixing existing releases, and for each Hot Fix release there will be a corresponding branch. Finally, we have the *master* branch which contains the latest stable version of the product.

6.3.4 Code Review

At Feedzai, we use Phabricator[21] as our code review platform. With this platform we get useful features like a view of the changes performed in each file, the possibility of commenting the overall work or just a specific line in the code. Additionally, for a *commit* to be accepted, we must provide at least two developers to do the review, and at least one of them must do the review.

During this thesis, we got involved in some code reviews, which helped in getting more familiarized with the overall product. Plus, our own developed code was also reviewed by other software engineers at Feedzai.

[19] `http://git-scm.com/`

[20] The branching model adopted is a variation of the one presented at the following link `http://nvie.com/posts/a-successful-git-branching-model/`

[21] http://phabricator.org/

Chapter 7

Conclusion and Future Work

In this chapter we will present a summary of the conclusion and findings of our work. In the final section, we will give possible future paths for the work done.

7.1 Summary

In this work, we started by implementing the Pattern Matching capabilities into PQL. This is a major feature addition since previously to achieve similar results we would need to use the rules engine embedded in Feedzai Pulse, which is a separate module from PKernel. However, this *workaround* did not have the same level of abstraction as Pattern Matching, e.g. we do not have time policies clauses (*within, after*) nor their consumption policies (*strict, relaxed*); in order to achieve this in the rules engine we would have to utilize broader operations, thus being more verbose and more prune to errors and *bugs*. Furthermore, we proved that this implementation can be applied in a real-time scenario, as demonstrated by the results of the benchmark performed. Finally, this feature has been included in the latest release of Feedzai Pulse - 14.0.

The State-Based programming was always a much talked about and idealized feature amongst Feedzai's engineers, since it would provide them a much higher level language to model a common use-case amongst the Pulse's client deployments. This feature will be included in the upcoming release of Feedzai Pulse - 14.1.

On a more personal note, this work was an amazing journey where I had the chance of getting in touch with new technologies and concepts, as well as improve my knowledge on subjects I already had contact with. Overall, this was a truly rewarding experience.

7.2 Future Work

Concerning the pattern matching there are a few possible future paths which can be taken:

- Evaluate the need for more operators;

- Evaluate pattern matching in field-programmable gate array (FPGA). Some work with regard to this approach can already be seen in [66].

Furthermore, we may also choose to mitigate the limitations currently present in the implementation - refer to section 4.7.

The *State-Based Programming* implementation still lacks a performance benchmark, however, we are confident the performance will be adequate in a real-time environment, especially since the operations being made have already proven to be effective in this environment. As possible future paths, we have the creation of a Graphical User Interface (GUI) for the *entity* definition to be more user-friendly, and consequently be used by a much broader audience, not just people who are familiar with PQL.

Finally, we will also submit a paper to conferences on the Event Processing field.

References

[1] 10. The Rete Algorithm. http://herzberg.ca.sandia.gov/docs/52/rete.html. Last accessed: 30/06/2014.

[2] Advantages of the "As a user, I want" user story template. http://www.mountaingoatsoftware.com/blog/advantages-of-the-as-a-user-i-want-user-story-template. Last accessed: 30/06/2014.

[3] CEP market players end 2012. http://buki79.tumblr.com/post/41611299435/cep-market-players-end-2012. Last accessed: 30/06/2014.

[4] Domain-Specific Languages: An Introductory Example. http://www.informit.com/articles/article.aspx?p=1592379&seqNum=2. Last accessed: 30/06/2014.

[5] Drools - ReteOO. http://legacy.drools.codehaus.org/ReteOO. Last accessed: 30/06/2014.

[6] EPL Reference: Match Recognize. http://esper.codehaus.org/esper-4.10.0/doc/reference/en-US/html/match-recognize.html. Last accessed: 30/06/2014.

[7] EPL Reference: Patterns. http://esper.codehaus.org/esper-4.10.0/doc/reference/en-US/html/event_patterns.html. Last accessed: 30/06/2014.

[8] EsperTech Technical Datasheet. http://www.espertech.com/download/public/EsperTech%20technical%20datasheet.pdf. Last accessed: 30/06/2014.

[9] LINQ (Language-Interpreted Query). http://msdn.microsoft.com/en-us/library/bb397926.aspx. Last accessed: 30/06/2014.

[10] MoSCoW : Requirements Prioritization Technique. http://businessanalystlearnings.com/ba-techniques/2013/3/5/moscow-technique-requirements-prioritization. Last accessed: 30/06/2014.

[11] MoSCoW Method for Requirements Prioritization. http://www.businessanalysis.in/2013/06/moscow-method-for-requirements.html. Last accessed: 30/06/2014.

[12] Pattern Matching in sequences of rows. http://dist.codehaus.org/esper//row-pattern-recogniton-11-public.pdf. Last accessed: 30/06/2014.

[13] Pattern Recognition With MATCH_RECOGNIZE. http://docs.oracle.com/cd/E14571_01/apirefs.1111/e12048/pattern_recog.htm.

[14] Pulse Query Language. https://docs.feedzai.com/display/pulse/Pulse+Query+Language. Last accessed: 30/06/2014.

[15] Rete Algorithm Demystified. http://techondec.wordpress.com/2011/03/14/rete-algorithm-demystified-part-2/. Last accessed: 30/06/2014.

[16] Rete NT - 10x faster than Rete 2. http://blog.athico.com/2010/08/rete-nt-10-x-faster-than-rete-2.html. Last accessed: 30/06/2014.

[17] Rete, Rete II, Rete III & Rete in TIBCO Business Events. http://www.slideshare.net/TimBassCEP/ss-presentation-716373. Last accessed: 30/06/2014.

[18] RETE2. http://www.pst.com/rete2.htm. Last accessed: 30/06/2014.

[19] SASE website. http://avid.cs.umass.edu/sase. Last accessed: 30/06/2014.

[20] Siddhi website. http://siddhi.sourceforge.net/. Last accessed: 30/06/2014.

[21] State Machine Diagrams. http://www.uml-diagrams.org/state-machine-diagrams.html. Last accessed: 30/06/2014.

[22] TIBCO Business Events website. http://www.tibco.com/products/event-processing/complex-event-processing/businessevents/default.jsp. Last accessed: 30/06/2014.

[23] User stories: a beginners guide. http://www.boost.co.nz/blog/2010/09/user-stories/. Last accessed: 30/06/2014.

[24] What algorithms does Esper use? Is it based on research? http://esper.codehaus.org/tutorials/faq_esper/faq.html#what-algorithms. Last accessed: 30/06/2014.

[25] What is Rete III? http://dmblog.fico.com/2005/09/what_is_rete_ii.html. Last accessed: 30/06/2014.

[26] WSO2 Complex Event Processor Documentation - Patterns. http://docs.wso2.org/display/CEP300/Patterns. Last accessed: 30/06/2014.

[27] WSO2 Complex Event Processor website. http://wso2.com/products/complex-event-processor/. Last accessed: 30/06/2014.

[28] Event Processing Glossary - Version 2.0. www.complexevents.com/wp-content/uploads/2011/08/EPTS_Event_Processing_Glossary_v2.pdf, 2011. Last accessed: 30/06/2014.

[29] TIBCO Business Events Getting Started. https://docs.tibco.com/pub/businessevents/5.1.1_dec_2012/pdf/tib_be_getting_started.pdf, 2012. Last accessed: 30/06/2014.

[30] TIBCO Business Events Pattern Matching Developer's Guide. https://docs.tibco.com/pub/businessevents_event_stream_processing/5.1.1_dec_2012/pdf/tib_be_event_stream_processing_pattern_matcher_developers_guide.pdf, 2012. Last accessed: 30/06/2014.

[31] Adaikkalavan, R. SNOOP Event Specification: Formalization Algorithms, and Implementation Using Interval-Based Semantics. Master's thesis, The University of Texas at Arlington, 2002.

[32] Agrawal, J., Diao, Y., Gyllstrom, D., and Immerman, N. Efficient pattern matching over event streams. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data - SIGMOD '08* (2008), p. 147.

[33] Balkesen, C., Dindar, N., Wetter, M., and Tatbul, N. Rip: Run-based intra-query parallelism for scalable complex event processing. In *Proceedings of the 7th ACM International Conference on Distributed Event-based Systems* (New York, NY, USA, 2013), DEBS '13, ACM, pp. 3–14.

[34] Batory, D. The LEAPS Algorithm. Tech. rep., University of Texas at Austin, 1994.

[35] Bizarro, P. Use Case Tutorial - Lessons Learned (7/7). http://www.slideshare.net/pedrobizarro/use-case-tutorial-lessons-learned-77. Last accessed: 30/06/2014.

[36] Chakravarthy, S., Krishnaprasad, V., Anwar, E., Kim, S.-K., Gehani, N. H., Jagadish, H. V., and Shmueli, O. Composite Events for Active Databases : Contexts and Detection Semantics. In *Proceedings of the 20th International Conference on Very Large Data Bases, (VLDB'94), September 12-15, 1994, Santiago de Chile, Chile.* (1994), pp. 606–617.

[37] Chakravarthy, S., and Mishra, D. Snoop: An expressive event specification language for active databases, 1993.

[38] Chakravarthy, S., and Mishra, D. Snoop: An expressive event specification language for active databases, 1994.

[39] Challand, Y. A Real Time Expert System - Adapting Match Algorithms and Implementing a Tailored Rule Language. Tech. rep., Aalborg University, Aalborg, 2011.

[40] Doorenbos, R. B. *Production Matching for Large Learning Systems.* PhD thesis, Carnegie Mellon University, 1995.

[41] Etzion, O., and Niblett, P. *Event Processing in Action.* Manning Publications, 2010.

[42] Forgy, C. Rete: A Fast Algorithm for the Many Patterns/Many Objects Match Problem. *Artificial Intelligence 19* (1982), 17–37.

[43] Fowler, M. *Domain Specific Languages*, 1st ed. Addison-Wesley Professional, 2010.

[44] Gatziu, S., and Dittrich, K. Detecting composite events in active database systems using Petri nets. *Proceedings of IEEE International Workshop on Research Issues in Data Engineering: Active Databases Systems* (1994).

[45] Gehani, N., Jagadish, H., and Shmueli, O. Composite event specification in active databases: Model & implementation. In *Proceedings of the International Conference on Very Large Data Bases* (1992), pp. 327–337.

[46] Gyllstrom, D., Agrawal, J., Diao, Y. D. Y., and Immerman, N. On Supporting Kleene Closure over Event Streams. *2008 IEEE 24th International Conference on Data Engineering* (2008).

[47] Hanson, E. Gator: A discrimination network structure for active database rule condition matching. Tech. rep., University of Florida, 1993.

[48] Hanson, E., and Hasan, M. S. Gator: An optimized discrimination network for active database rule condition testing. Tech. rep., University of Florida, 1993.

[49] Hasan, M. Z. The management of data, events, and information presentation for network management. Tech. rep., in Computer Science, 1996.

[50] Luckham, D., Kenney, J., Augustin, L., Vera, J., Bryan, D., and Mann, W. Specification and analysis of system architecture using Rapide. *IEEE Transactions on Software Engineering 21* (1995).

[51] Luckham, D. C. Rapide: A language and toolset for simulation of distributed systems by partial orderings of events. Tech. rep., Stanford University, Stanford, CA, USA, 1996.

[52] Luckham, D. C. *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems*, vol. 2003. Addison-Wesley Professional, 2002.

[53] Madden, N. Optimising rete for low-memory multiagent systems. In *GAME-ON* (2003), Q. H. Mehdi, N. E. Gough, and S. Natkin, Eds., EUROSIS, pp. 77–.

[54] Miranker, D. P. TREAT: A better match algorithm for AI production systems. *Elements 1* (1987), 42–47.

[55] Motakis, I., and Zaniolo, C. Composite temporal events in active database rules: A logic-oriented approach. In *Deductive and Object-Oriented Databases*, vol. 1013. Springer Berlin Heidelberg, 1995, pp. 19–37.

[56] Müller, A. Event Correlation Engine. Master's thesis, ETH Zürich, 2009.

[57] Nayak, P., Gupta, A., and Rosenbloom, P. Comparison of the Rete and Treat production matchers for Soar (A summary). In *Proceedings of the Seventh National Conference on Artificial Intelligence* (1988), pp. 693–698.

[58] Schmidt, K.-U., Stühmer, R., and Stojanovic, L. Blending Complex Event Processing with the RETE Algorithm. *iCEP2008 1st International workshop on Complex Event Processing for the Future Internet colocated with the Future Internet Symposium FIS2008 Vol-412* (2008), 1–10.

[59] Schwaber, K., and Sutherlan, J. The Scrum Guide - The Definitive Guide to Scrum: The Rules of the game. `https://www.scrum.org/Portals/0/Documents/ScrumGuides/2013/Scrum-Guide.pdf#zoom=100`, 2013.

[60] Sriskandarajah, S., Gajasinghe, K., Loku Narangoda, I., and Chaturanga, S. Siddhi-CEP - High Performance Complex Event Processing Engine. Master's thesis, University of Moratuwa, Sri Lanka, 2011.

[61] Stellman, A. Understanding Nonfunctional Requirements. `http://broadcast.oreilly.com/2010/02/nonfunctional-requirements-how.html`. Last accessed: 30/06/2014.

[62] Suhothayan, S., Gajasinghe, K., Loku Narangoda, I., Chaturanga, S., Perera, S., Nanayakkara, V., and Narangoda, I. Siddhi: a second look at complex event processing architectures. In *Proceedings of the 2011 ACM workshop on Gateway computing environments - GCE '11* (2011), p. 43.

[63] Walzer, K., Breddin, T., and Groch, M. Relative temporal constraints in the Rete algorithm for complex event detection. In *Proceedings of the second ...* (2008), pp. 147–155.

[64] Wang, Y.-W. W. Y.-W., and Hanson, E. A performance comparison of the Rete and TREAT algorithms for testing database rule conditions. *[1992] Eighth International Conference on Data Engineering* (1992).

[65] Widder, A., von Ammon, R., Schaeffer, P., and Wolff, C. Combining discriminant analysis and neural networks for fraud detection on the base of complex event processing. In *Fast Abstract, Second International Conference on Distributed Event-Based Systems, DEBS 2008, Rom, Juli 2008* (2008).

[66] Woods, L., Teubner, J., and Alonso, G. Real-time pattern matching with fpgas. *2013 IEEE 29th International Conference on Data Engineering (ICDE) 0* (2011), 1292–1295.