Scalable Data Analytics,
Scalable Algorithms, Software Frameworks
and Visualization ICT-2013 4.2.a

Project **FP7-619435/SPEEDD**
Deliverable **D3.1**
Distribution **Public**

http://speedd-project.eu

First version of event recognition and forecasting technology (Part 1)

Fabiana Fournier (IBM) and Inna Skarbovsky (IBM)

Status: Final (Version 1.0)

December 2014

Project

Project Ref. no	FP7-619435
Project acronym	SPEEDD
Project full title	Scalable ProactivE Event-Driven Decision Making
Project site	http://speedd-project.eu/
Project start	February 2014
Project duration	3 years
EC Project Officer	Aleksandra Wesolowska

Deliverable

Deliverable type	Prototype
Distribution level	Restricted
Deliverable Number	D3.1
Deliverable Title	First version of event recognition and forecasting technology
Contractual date of delivery	M11 (December 2014)
Actual date of delivery	December 2014
Relevant Task(s)	WP3/Tasks 3.2 and 3.3
Partner Responsible	IBM
Other contributors	NCSR
Number of pages	50
Author(s)	Fabiana Fournier (IBM) and Inna Skarbovsky (IBM)
Internal Reviewers	Pedro Bizarro and Ivo Correia (Feedzai)
Status & version	Final
Keywords	Complex event processing, forecasted/predicted event, event recognition, probabilistic event

Executive Summary

At the heart of the SPEEDD prototype relies the event processing component. Its role is to detect events and derive situations to feed the decision module so proactive actions can be taken. To this end, the complex event processing component needs to deal with uncertainty in the input as well as the output events. This document is the first part of the Deliverable 3.1 “First version of event recognition and forecasting technology” and its purpose is to present the advancements made in the complex event processing tooling of SPEEDD to cope with uncertainty.

The inclusion of uncertainty aspects, mainly manifested in the run-time module, impacts all levels of the architecture and logic of an event processing engine. Even at this stage, the current implemented applications possess a high level of sophistication and complexity. The extensions made in the complex event processing engine include the addition of new built-in attributes and functions, the support of new types of operands, and the support of the event processing patterns to cope with all these. Although these extensions are driven by the use cases requirements, these have not been implemented ad-hoc, but as generic building blocks in the complex event processing programmatic language making it a first-of-a-kind event processing engine capable to deal with uncertainty and to derive forecasted events.

Document History

Version	Date	Author	Change Description
0.1	15/11/2014	Fabiana Fournier (IBM)	First draft
0.2	1/12/2014	Fabiana Fournier (IBM)	Second draft for internal review
0.3	15/12/2014	Fabiana Fournier (IBM)	Updates per internal review
1.0	30/12/2014	Fabiana Fournier (IBM)	Final fixes and cleanup

Table of Contents

List of Tables

List of Figures

Acronyms

CDF	Cumulative Distribution Function
CEP	Complex Event Processing
CNP	Card Not Present
CP	Card Present
CVV	Card Verification Value
EEP	Extendable Expression Parser
EPA	Event Processing Agent
EPN	Event Processing Network
JSON	JavaScript Object Notation
SPEEDD	Scalable ProactivE Event-Driven Decision making
WP	Work Package

1 Introduction

1.1 Purpose and scope of the document

Work Package 3 (WP3) "Real-Time Event Recognition and Forecasting under Uncertainty" deals with all the developments around event processing technologies under uncertainty. This report is the first version of SPEEDD (Scalable ProactivE Event-Driven Decision) event recognition and forecasting technology and it includes first results for T3.2 (event recognition under uncertainty) and T3.3 (event forecasting under uncertainty), and the first version of the event recognition and forecasting component (software). This report presents the extensions and developments made with relation to the project two use cases: the traffic management use case (see D7.1 "User Requirements and Scenario Definitions") and the credit card fraud use case (see D8.1 "User Requirements and Scenario Definitions"). Two updated versions of this report will be submitted at M22 and M32 of the project to describe further developments achieved.

This report covers all aspects of the event driven run-time module in SPEEDD. Basically, the report describes the extensions to the engine (both in the user interface and in the run-time) and the implementation for first year of the CEP (Complex Event Processing) component applications. A complementary report led by partner NCSR describes the first results for T3.1 (machine learning for event definition construction) which covers the off-line aspects of the event-driven under uncertainty.

The main role of the CEP component is to feed the decision making component with meaningful events for its decision making process (see D6.1 "The Architecture Design of the SPEEDD prototype"). The developments made in the CEP engine are driven by requirements imposed by both the use cases and the decision making component. In other words, the use cases and decision making process requirements dictate the extensions to be made in the run-time engine.

This report is structured as follows: Section 2 provides some background on CEP and terminology used throughout this report. Section 3 describes the CEP component in SPEEDD, including extensions made to cope with uncertainty both in the user interface and run-time engine, as well as the implementation of the two use cases. Section 4 presents initial performance outcomes of our implementations. We conclude the report with summary and future steps.

1.2 Relationship with other documents

At the heart of the SPEEDD prototype resides the complex event processing component, therefore, this report is strongly related to D6.1 "The Architecture Design of the SPEEDD prototype". The requirements for the CEP engine are dictated from the use cases in the project, thus, this report is also strongly related to system requirements for the Proactive Traffic Management use case described in D8.1 and for the Proactive Credit Card Fraud Management described in D7.1. The main goal of the CEP component is to derive forecasted events that feed the decision making component so actions can be taken before an

undesired event (such as a congestion situation in the high way) takes place. Therefore our work is also related to D4.1 “First version of real-time decision-making technology”.

2 Complex event processing background

Each complex event processing engine uses its own terminology and semantics. We follow the semantics presented in Etzion’s and Niblet’s book [1]. We describe below some main terms used in our work for the sake of clarity. We use the IBM Proactive Technology Online (PROTON) research asset as the complex event processing engine in SPEEDD (see D6.1). This engine has been released as open source as an outcome of the FI-WARE project (being Proton the CEP Generic Enabler in the FI-WARE platform[1]) and it is extended to cope with predictive capabilities in the scope of the SPEEDD project.

2.1 Terminology

Henceforth we briefly present main concepts and building blocks in our terminology. For further details refer to [1].

2.1.1 Event types

Generally speaking, an **event** is an occurrence within a particular system or domain; it is something that has happened, or is contemplated as having happened in that domain. The word “event” is also used to mean a programming entity that represents such an occurrence in a computing system. In the latter definition, an event is an object of an event type. Events are actual instances of the event types and have specific values. For example, the event "*today at 10 PM a customer named John Doe made a new deposit to his bank account”* is an instance of the *Transaction event type*. An event type specifies the information that is contained in its event instances by defining a set of **attributes**. The event attributes are grouped into the header or metadata (e.g., the occurrence time of the event instance) and the payload (specific information about the event, e.g., *customer name*).

We relate to the following event types:

A **raw event** is an event that is introduced into an event processing system by an event **producer** (an entity at the edge of an event processing system that introduces events to the system). An example of a raw event is a *Cash deposit* into a bank account.

A **derived event** is an event that is generated as a result of event processing that takes place inside the event processing system. An example is that *a Large cash deposit has been made into a bank account*.

A **situation** is a derived event that is emitted outside the event processing system and consumed by at least one **consumer** (an entity at the edge of an event processing system that receives events from the system). An example is a *Suspicious bank account*.

[1] https://forge.fi-ware.org/plugins/mediawiki/wiki/fiware/index.php/FI-WARE_Architecture

2.1.2 Context

Context is a named specification of conditions that groups event instances so they can be processed in a related way. While there exist several context dimensions, in this report we employ the two most commonly used dimensions (in the future we might enlarge the set of context types, depending on the scenarios requirements): temporal and segmentation-oriented. A **temporal context** consists of one or more time intervals, possibly overlapping. Each time interval corresponds to a context partition, which contains events that occur during that interval. A **segmentation-oriented context** is used to group event instances into context partitions based on the value of an attribute or collection of attributes in the instances themselves. As a simple example, consider a single stream of input events, in which each event contains a customer identifier attribute. The value of this attribute can be used to group events so there is a separate context partition for each customer. Each context partition contains only events related to that customer, so the behaviour of each customer can be tracked independently of the other customers. A **composite context** is a context that is composed from two or more contexts, known as its members. The set of context partitions for the composite context is the Cartesian product of the partition sets of the member contexts

2.1.3 Event Processing Network (EPN)

An **Event Processing Network (EPN)** is a conceptual model, describing the event processing flow execution. An EPN comprises a collection of Event processing Agents (EPAs), event producers, events and consumers (Figure 1). The network describes the flow of events originating at event producers and flowing through various event processing agents to eventually reach event consumers. For example, in Figure 1, events from Producer 1 are processed by Agent 1. Events derived by Agent 1 are of interest to Consumer 1 but are also processed by Agent 3 together with events derived from Agent 2. Note that the intermediary processing between producers and consumers in every installation is made up of several functions and often the same function is applied to different events for different purposes at different stages of the processing.

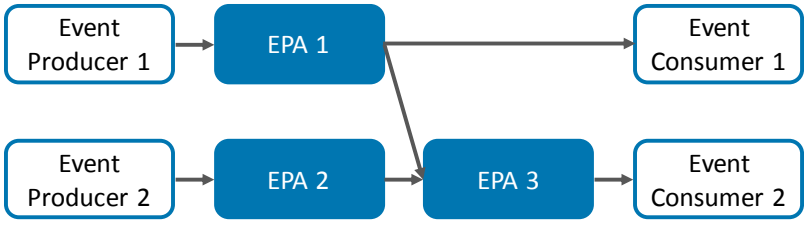

Figure 1: Illustration of an event processing network

2.1.3.1 Event Processing Agent (EPA)

An **Event Processing Agent (EPA)** is a component that, given a set of input/incoming events within a context, applies some logic for generating a set of output/derived events. An EPA can apply different event patterns to detect specific relations among the input events.

An EPA performs three logical steps, a.k.a **pattern matching process** or **event recognition** (see Figure 2). Please note that all three steps are optional but at least one must be done inside an EPA.

- The **filtering step**, in which relevant events from the input events are selected for processing according to the filter conditions. The output of this step is a set of **participant events**.
- The **matching step** that takes all events that passed the filtering and looks for matches between these events, using an event processing pattern or some other kind of matching criterion. The output of this step is the **matching set**.
- The **derivation step** that takes the output from the matching step and uses it to derive the output events by applying derivation formulae.

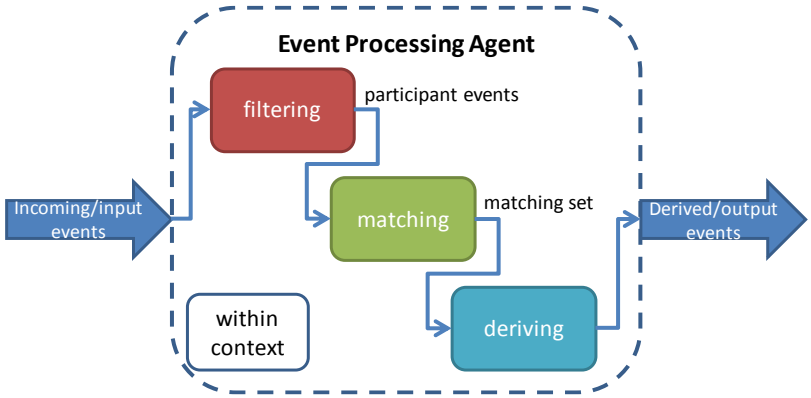

Figure 2: Event recognition process in an EPA

An **event pattern** is a template specifying one or more combinations of events. Given any collection of events, if it's possible to find one or more subsets of those events that match a particular pattern, it can be said that such a subset satisfies the pattern. Some common examples of patterns applied in our scenarios:

- **Sequence**, means that at least one instance of all participating event types must arrive in a specified order for the pattern to be matched.
- **Count**, means that the number of instances in the participant event set satisfies the pattern's number assertion.
- **All**, means that at least one instance of all participating event types must arrive for the pattern to be matched; the arrival order in this case is immaterial.
- **Trend**, events need to satisfy a specific change (increasing or decreasing) over time of some observed value; this refers to the value of a specific attribute or attributes.
- **Absence**, a specified event(s) must not occur within a predefined time window. The matching set in this case is empty.
- **Average (AVG)**, means that the value of a specific attribute, averaged over all participant events, satisfies the average threshold assertion.

2.1.4 Pattern policies

A **pattern policy** is a named parameter that disambiguates the semantics of the pattern and the pattern matching process. Pattern policies fine-tune the way the pattern detection process works. Proton supports five types of policies:

Evaluation policy – when the matching sets are produced? The EPA can either generate output incrementally (in this case the evaluation policy is called *Immediate*) or at the end of the temporal context (called *Deferred*).

Cardinality policy – how many matching sets are produced within a single context partition? Cardinality policy helps limiting the number of matching sets generated, and thus the number of derived events produced. The policy type can be *single*, meaning only one matching set is generated; or *unrestricted*, meaning there are no restrictions on the number of matching sets generated.

Repeated/Instance Selection type policy – what happens if the matching step encounters multiple events of the same type? The *override* repeated policy means that whenever a new event instance is encountered and the participant set already contains the required number of instances of that type, the new instance replaces the oldest previous instance of that type. The *every* repeated policy means that every instance is kept, meaning all possible matching sets can be produced. *First* means that every instance is kept, but only the earliest instance of each type is used for matching. *Last* is the same as first, but the latest instance of each type is used for matching.

Consumption policy – what happens to a particular event after it has been included in the matching set? Possible consumption policies are *consume*, meaning each event instance can be used in only one matching set; and *reuse*, meaning an event instance can participate in an unrestricted number of matching sets.

Policy relevance can be dictated by the event pattern. For example, the evaluation policy for an absence pattern is always deferred (as we are testing the existence of an event instance for a specified temporal context). Also, not all possible policies combinations are meaningful. For example, the choice of consumption policy is irrelevant if the cardinality policy is single, because that means that the matching step runs only once.

2.1.5 Context initiator policies

A temporal context starts with an **initiator** and ends with a **terminator**. An initiator can be an event, system startup, or absolute time. A terminator ends the temporal context. The terminator can be an event, relative expiration time, an absolute expiration time, or “never ends”, i.e. the temporal context remains open until engine shutdown.

A **context initiator policy** tunes up the semantics for temporal contexts in which the context initiator is determined by an event. A context initiator policy defines the behaviours required when a window has been opened and a subsequent initiator event is detected. The options are: *add*, a new window is opened alongside the existing one; or *ignore*, the original window is preserved.

2.2 Complex event processing tooling

In the SPEEDD project the complex event processing component is built on and extends the IBM Proactive Technology Online (PROTON) research asset. This asset has become open-source[2] in the scope of the FI-WARE FI-PPP project[3]. Documentation regarding the CEP open source asset can be found in [2], [3], and [4].

PROTON comprises a run-time engine, producers, and consumers with the characteristics and capabilities described in the Background Section (Section 2). Specifically, it includes an integrated run-time platform to develop, deploy, and maintain event-driven applications using a single programming model.

2.2.1 Event attributes

Every event instance has a set of built-in attributes (metadata). PROTON employs the following attributes in the event type's metadata:

- *Name* – of the event type.
- *OccurenceTime* – a timestamp attribute, which we expect the event source to fill in as the occurrence time of the event. If left empty, this equals the *detectionTime* attribute value.
- *DetectionTime* – a timestamp attribute that records the time the CEP engine detected the event. The time is measured in milliseconds, specifying the time difference between the current machine time at the moment of event detection and midnight, January 1, 1970 UTC.
- *EventId* – a unique string identification of the event, which can be set by the event source to match the asynchronous output for the event.
- *EventSource* – holds the source of the event (usually the name of event producer).

The above built-in attributes can be used in an expression in the same manner as user-defined attributes. User defined attributes can be added to the event class by defining their types. If the attribute is an array, its dimension should be specified.

2.2.2 PROTON interfaces

PROTON standalone runtime engine has three main interfaces with its environment as depicted in Figure 3.

1. Input adapters for getting incoming events
2. Output adapters for sending derived events
3. CEP application definition (build time or authoring tool)

[2] Link to the open source: https://github.com/ishkin/Proton
[3] http://www.fi-ware.org/

The application definitions, i.e. the EPN, are written by the application developer during the build-time. The definitions output in JSON (JavaScript Object Notation) format, is provided as configuration to the CEP run-time engine. At run-time, the standalone CEP engine receives incoming events through the input adapters, processes these incoming events according to the definitions, and sends derived events through the output adapters (see Figure 3).

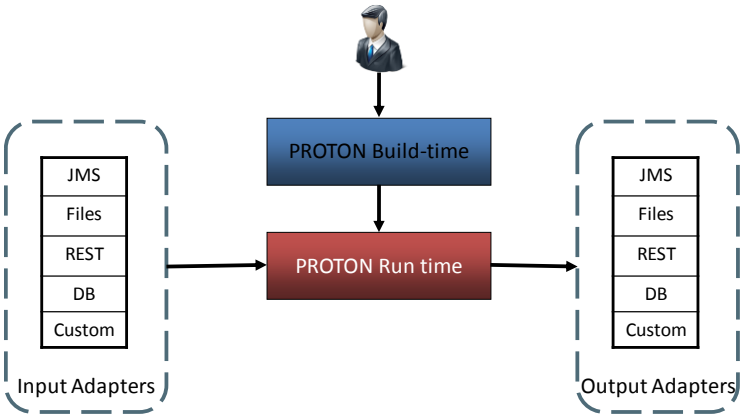

Figure 3: PROTON interfaces

2.2.3 Input and output adapters

As aforementioned, the definitions of the producers and consumers are specified during the application build-time and are translated into input and output adapters during execution time in the standalone CEP engine. The physical entities representing the logical entities of producers and consumers in PROTON are adapter instances. For each producer an input adapter is defined, which defines how to pull the data from the source resource and how to format the data into PROTON's object format before delivering it to the run-time engine. The adapter is environment-agnostic, but uses the environment-specific connector object, injected into the adapter during its creation, to connect to PROTON runtime.

In the distributed implementation (on top of STORM [5]) where PROTON runtime is just one part of the general architecture, the communication with the CEP engine is done via STORM communication channels. Therefore, Proton receives STORM tuples as input and emits STORM tuples as output. Each tuple consists of the name of the event type, and a Map of the event attributes. Therefore, in this case the adapters are not employed. For more elaborated specification of this mechanism see D6.1.

In the trials carried out to test the use cases implementation (see Sections 3.4 and 3.5) we use a CSV file for input and output. The input file contains simulated data using the same schema that real-data in order to test the correctness of the EPN defined for the use cases. In the prototype demo, we use Kafka [6] as the event bus as described in the architecture deliverable (D6.1).

2.2.4 PROTON definitions

The CEP application definitions file can be created in three ways:

1. Build-time user interface – By this, the application developer creates the building blocks of the application definitions. This is done by filling up forms without the need to write any code. The file that is generated is exported in a JSON format to the CEP run-time engine.

2. Programming – The JSON definitions file can alternatively be generated programmatically by an external application and fed into the CEP run-time engine.

3. Manually – The JSON file is created manually and fed into the CEP run-time engine.

The created JSON file comprises the following definitions:

- Event types – the events that are expected to be received as input or to be sent as output. An event type definition includes the event name and a list of its attributes.
- Producers – the event sources and the way PROTON gets events from those sources.
- Consumers – the event consumers and the way they get derived events from PROTON.
- Temporal contexts – time window contexts in which event processing agents are active.
- Segmentation contexts – semantic contexts that are used to group several events to be used by the EPAs.
- Composite contexts – grouping together several different contexts.
- Event processing agents – patterns of incoming events in specific context that detect situations and generate derived events. An EPA includes most of the following general characteristics:
    - Unique name
    - EPA type (operator). For each operator, different sets of properties and operands are applicable.
    - Context
    - Other properties such as condition
    - Participating events
    - Segmentation contexts
    - Derived events

The JSON file that is created at build-time contains all EPN definitions, including definitions for event types, EPAs, contexts, producers, and consumers. At execution, the standalone run-time engine accesses the metadata file, loads and parses all the definitions, creates a thread per each input and output adapter, and starts listening for events incoming from the input adapters (producers) and forwards events to output adapters (consumers).

For the distributed implementation on top of STORM, an input Bolt serves the same function as input adapter, and the derived events are passed as STORM tuples to the next stage in the SPEEDD topology processing (see D6.1).

2.2.5 Expressions in PROTON

When building an event processing application, we sometimes need to set values to attributes or properties. We do so by writing expressions. These expressions are tested at build-time and evaluated at runtime by the **PROTON EEP** (Expandable Expression Parser)

An expression can be any combination of these:

- Constant (5, true, false, "silver", …)
- Field (<EventName>.<EventAttribute>)
- Built-in attribute (detectionTime, count, …) and built-in aggregation attributes (sum, max, …)
- Operator (+, -, =, …)
- Segmentation context (segmentationContext.CustomerKey)
- Built-in function (arrayContains(a,v), distance(x1,y1,x2,y2), …)

Examples:

Max(DayStart.InitialStockLevel,0)

if CustomerRating="gold" then "approve" else "reject" endif

Examples of built-in functions:

- **Max** – Max(a,b,c) returns the maximum number among the arguments.
- **Min** – Min(x,100) returns the minimum number among the arguments.
- **Average** – Average(x,y,z,t) returns the average number of the arguments.
- **Modulo** – Mod(x,y) returns the remainder when dividing x by y.
- **Round** – Round(x) returns the closest integer value to x.
- **Absolute** – Abs(x) returns the absolute value of x.
- **CompareTo** – CompareTo(str1,str2) compares two strings lexicographically. The result is a negative integer if str1 lexicographically precedes str2. The result is a positive integer if str1 lexicographically follows the str2. The result is zero if the strings are equal
- **Distance** – Distance(x1,y1,x2,y2) returns the distance between (x1,y1) and (x2,y2).
- **Angle** – Angle(x,y,z,w) calculates the angle generated between (x1,y1),(0,0),(x2,y2).
- **IsNull** – IsNull(val) checks whether the given val equals null. Returns a Boolean value.

PROTON EEP uses any of the following operators (Table 1).

Table 1: Operators in PROTON EEP

Type	Operator	Example
Mathematical	+ - / *	customerBuy.quantity + 5
Comparison	= == != > < <= >=	customerRating != "gold"
Boolean	and or not xor & && | || ! ^ true false	customerOrigin = "USA" or customerLanguage = "English"
If-then-else	if <cond1> then Exp1 elseif <cond2> then Exp2 else exp3 endif	If customerRating = "gold" then customerRequest else 0 endif
Lexical	++ (concatenation)	"Name: " ++ Trans.customerName

EEP expressions can include operands of types Boolean, Datetime, Double, Integer, Numeric, String, or array of each of these simple types.

3 Real-time event recognition and forecasting under uncertainty

Proactive event driven computing deals with the inherent uncertainty in the event inputs, in the output events, or in both ([7][8],[9], [10], and [11]).

In order to cope with uncertainty, PROTON has been extended both in the authoring tool and in the run-time engine as described in the following Sections. Requirements are driven by the use cases as well as by the Decision Module, as the latter apply situations detected by the CEP module to conduct real-time decision making.

3.1 Event definitions

In general, there exist two methods to define the rule patterns for a CEP application: machine learning and experts. In the first, the patterns are learnt automatically by a computer program, while in the second, they are given by an external entity; usually a subject expert matter specialized in the domain. It is also possible to combine between these two methods. Currently, the event patterns for both use cases as described in this document and implemented in PROTON, have been given by the domain partners. It might be that the hybrid approach will be used in the scope of SPEEDD at a later phase,

when definitions automatically learnt will go through a manual refinement process in order to be compatible to PROTON's definition file.

3.2 Extending PROTON's run-time engine

We say that if an event is predicted or forecasted to occur in the future, it can be expressed by setting the event a future *occurrence time* with an optional supported distribution and a certainty value. This imposes fundamental extensions in PROTON's Extendable Expression Parser (EEP) as described henceforth.

3.2.1 New built-in attributes

The event metadata in PROTON has been extended as follows:

- Addition of the built-in *double Certainty* attribute that stores the certainty of this event. An event has a default certainty value equal to 1, while it can have any value between (0-1].
- Support for distribution values (see next Section) of *Occurrence time* built-in attribute.

3.2.2 New operands types

The operands types have been extended to cope with distributions. Two types of distributions are supported: Continuous distribution and discrete distribution. Canonic forms of distribution have been implemented for each of these types. In the continuous case, there is a continuous function which its integral equals to 1. In the discrete case, it is a set of values with their associated probabilities where the sum of all probabilities is equal to 1.

Furthermore for continuous distributions, we support the Sigmoid(a,b,x) function which returns 1/ (1 + e^(-a (x - b))) (see Section 3.4.3.)

For discrete distributions, we currently support the following:

- Bernoulli (p) – where p is the probability of success
- Binomial (n, p) – where n is the number of trials and p is the probability of success
- Uniform (list of numbers) – each number is associated with a probability equals to 1/number of numbers

3.2.3 New built-in functions

- **CDF** – CDF(d, alpha) – returns the cumulative distribution function of d (which is of type distribution) at point alpha, which is the probability that d is smaller or equal to alpha.
- **Mean** – Mean(d) – returns the expectation of the distribution d
- **PDF** – PDF(d, x) – returns the probability density function of the distribution d at point x
- **Percentile** – Percentile(d, alpha) – returns the smallest value x, for which CDF(d, x) is larger or equal to alpha
- **Var** – Var(d) – returns the variance of the distribution d

We also added the two following built-in functions for the sake of the Sigmoid function

- **Power(a,x) -** returns a^x for two doubles
- **Exp(x) -** returns e^x

As a result, the EPAs' operators have been adjusted to deal with uncertain or probabilistic operands and expressions. Our two outstanding examples in our first implementations include TREND, COUNT, and ALL (see Sections 3.4 and 3.5).

3.3 Extending PROTON's authoring tool

The authoring tool forms have been accommodated to support all above extensions. Figure 4 shows a screenshot from PROTON's authoring tool showing the addition of the *Certainty* built-in attribute.

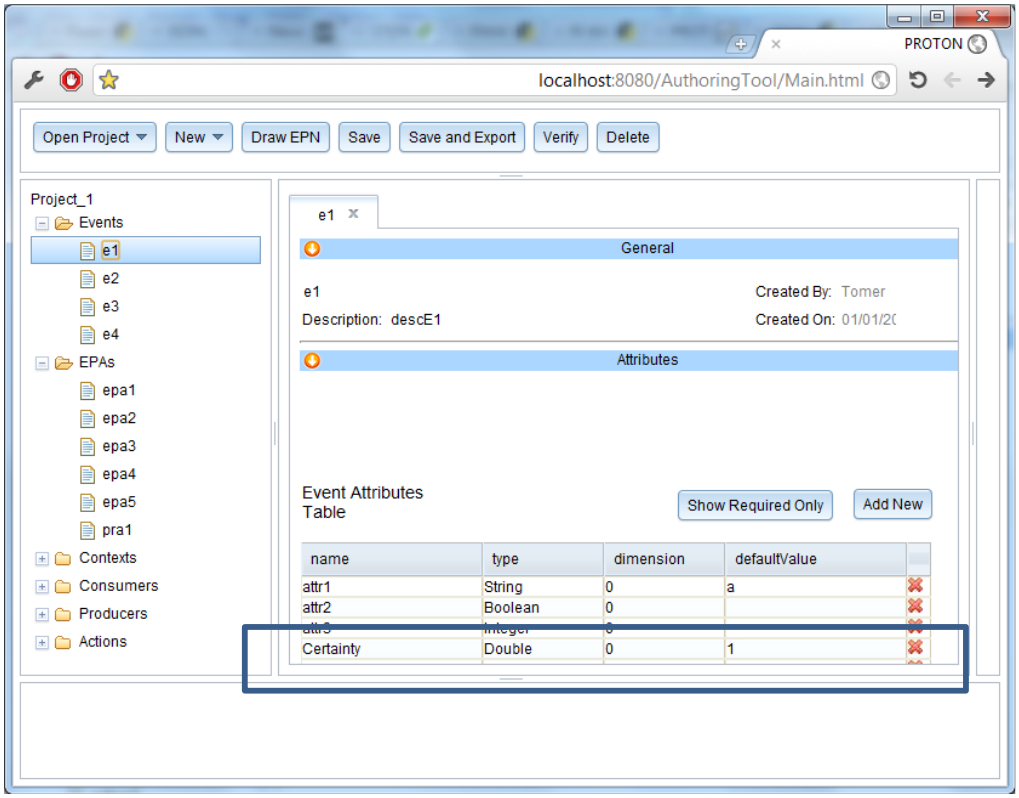

Figure 4: PROTON 's screenshot showing the new certainty attribute

3.4 Implementation of first EPN for the fraud detection use case

The overarching aim of the CEP module in this use case is to detect a potential fraud incident. To this end, a first EPN has been created with the collaboration of the use case owner partner Feedzai with the goal of having something meaningful and representative, yet doable to be achieved in the first year of the project. The outcome is an EPN consisting of seven EPAs shown in Figure 5 and detailed in the following Sections. For the sake of simplicity we only show the EPAs and the events flow in the network. Dotted lines represent events, other than input events, that are either initiators (in yellow) or terminators (in red) of a context. The PROTON JSON definitions file that comprises this EPN is provided as part of the software deliverable that accompanies this report.

In the current EPN we want to fire situations in the following cases (for detailed descriptions of each EPA see Sections 3.4.2.1-3.4.2.7):

- An attempt of withdrawing/paying of increasing amounts is carried out for a single card (EPA2, *IncreasingAmounts*).
- Several attempts of using a wrong CVV (Card Verification Value) for the same card are made (EPA5 or *CVVatack*).
- Increasing amounts of withdrawals/payments are carried out after a CVV attack occurs (EPA1 and EPA6).
- Multiple occurrences of a suspicious fraudulent card happen at the same ATM (EPA3 and EPA4)
- There are two consecutive attempts of using the same card in differ locations (EPA7 or *ClonedCard*).

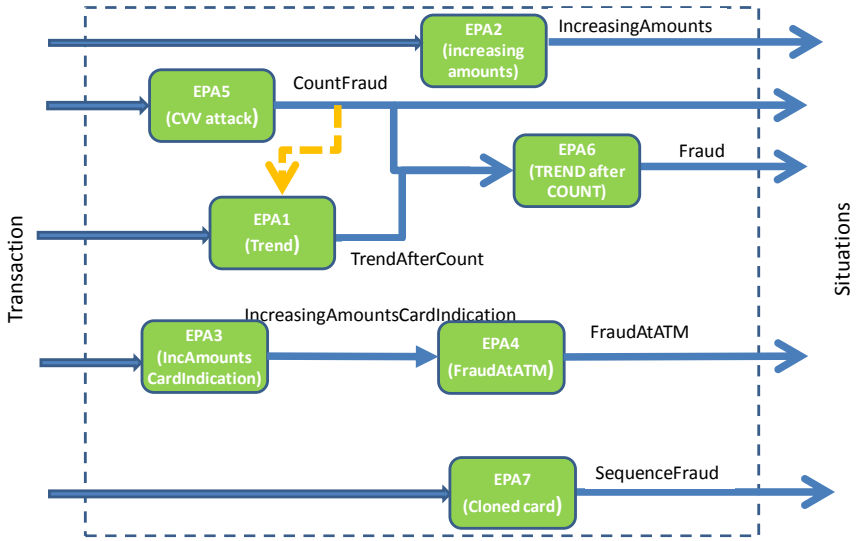

Figure 5: Fraud use case initial EPN

In the fraud use case we distinguish between two types of transactions: CP (Card is Present) and CNP (Card Not Present). For further information on the use cases please refer to D7.1 "User Requirements and Scenario Definitions". The logic of the patterns is the same for the two types, unless explicitly mentioned (see EPA7 description in Section 3.4.2.7). What differs is the temporal time windows length (CNP is much faster so the temporal windows will be shorter). In the current EPN we implicitly apply the CP case, having relatively long temporal windows (a few minutes) and omitting the filtering or opening of contexts just for the CP case. Obviously, the full implementation should cover both cases with explicit filter expressions for each EPA.

3.4.1 Event types

Eight event types have been defined that comprise the event inputs, outputs/derived, and situations as shown in Figure 5. For the sake of simplicity we only show the user-defined attributes or the event payload and not the metadata (refer to Section 2.2.1.).

Although the names of concepts in can be determined freely by the application designer in PROTON, we use some naming conventions for the sake of clarity. We denote event types with capital letters. Built-in/metadata attributes start with a capital letter, as well as payload attributes that hold operators values, while payload attributes start with a lower letter. Table 2 shows the event definitions for the fraud EPN.

Note that the *Transaction* raw event includes more fields or attributes. We defined only the ones required for pattern detection in the current EPN implementation. When running in SPEEDD architecture, PROTON will ignore event attributes not specified in its JSON.

Table 2: Initial EPN for the fraud use case

Event name	Transaction
Payload	card_pan; terminal_id; cvv_validation; amount_eur; acquirer_country; is cnp
Event name	IncreasingAmounts
Payload	card_pan; terminal_id; TrendCount
Event name	TrendAfterCount
Payload	card_pan; terminal_id; TrendCount
Event name	IncreasingAmountsFirst
Payload	card_pan; terminal_id; TrendCount
Event name	FraudAtATM
Payload	terminal_id
Event name	CountFraud
Payload	card_pan; TransactionsCount
Event name	Fraud
Payload	card_pan
Event name	SequenceFraud
Payload	card_pan

3.4.2 Event processing agents

Henceforth, we describe the EPAs in the following order: Event name; motivation; event recognition process (following Figure 2); contexts along with temporal context policy; and pattern policies.

In the event recognition process we only show the steps that take place in the specific EPA, while the others are greyed. For the *filtering step* we show the filtering expression; for the *matching step* we denote the pattern variables; and for the *derivation step* we denote the values assignment and calculations. Please note that for the sake of simplicity we only show the assignments that are not copy of values (all other derived event attributes values are copied from the input events). For attributes, we just denote their names without the prefix of 'attribute_name.'

In our initial implementation we use the *Sigmoid* probabilistic function to calculate the probability of the derived event. The Sigmoid function has been selected since it fits to situations that exhibit a progression from small beginnings that accelerate over time. A sigmoid curve is produced by a mathematical function having an "S" shape [12]. Of course, other parameters and functions might be applicable as well and are one of the topics for further testing in year two and three of the project. A Sigmoid function receives three parameters (a,b,x) and returns 1/ (1 + e^(-a (x - b))). The patterns have been tested with several parameters and the ones shown in the figures, have been chosen to run the input events set.

3.4.2.1 EPA1: Trend

Motivation: Check for consecutive transactions (at least two) with increasing amounts. The monitoring of the pattern starts only after a situation of CountFraud is detected. The same EPA as EPA2, except that the derived event serves as input for EPA6 and its context is initialized by the *CountFraud* event and not *Transaction*.

Event recognition process:

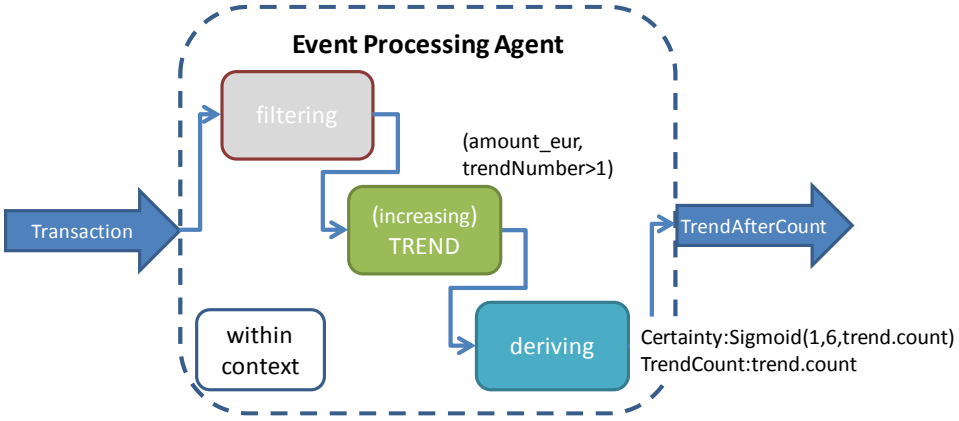

Figure 6: Event recognition process for Trend EPA

trendNumber and *trend.count* are built-in TREND variables that denote the minimal number of input events required in order to satisfy the pattern and the actual number correspondingly.

Pattern policies:

Evaluation	Cardinality	Repeated	Consumption
IMMEDIATE	UNRESTRICTED	FIRST	REUSE

Context:

Segmentation: by card_pan

Initiator policy: IGNORE

Meaning: A temporal window of 5 min is opened, once a *CountFraud* event is derived (see EPA5). In this elapsed time we check for a TREND pattern over the amounts in the transactions per a single card.

According to the pattern policies (see table above), we can derive more than one event (with larger values in the *certainty* attribute) if the TREND pattern is satisfies. After 5 min the temporal window closes.

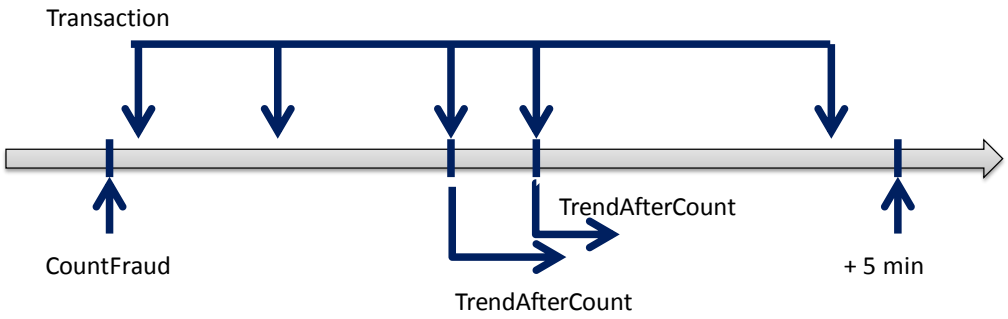

Figure 7: Context for Trend EPA

3.4.2.2 EPA2: IncreasingAmounts

Motivation: Check for consecutive transactions (at least two) with increasing amounts that can hint to a possible fraud attempt.

Event recognition process:

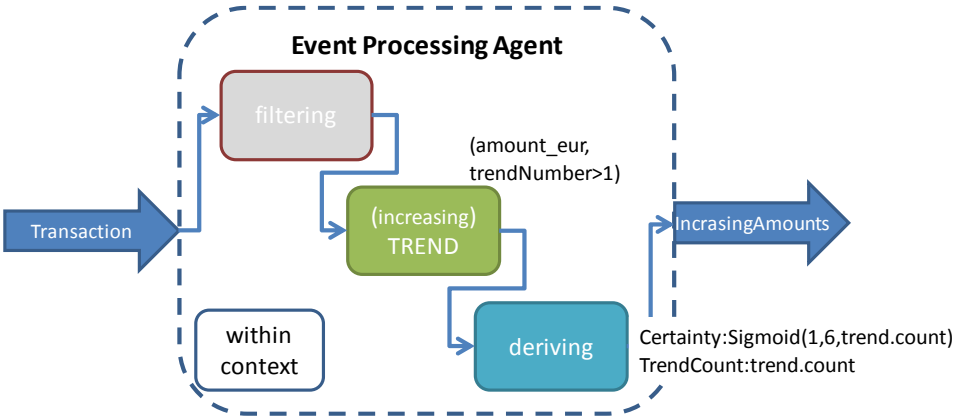

Figure 8: Event recognition process for IncreasingAmounts EPA

Pattern policies:

Evaluation	Cardinality	Repeated	Consumption
IMMEDIATE	UNRESTRICTED	FIRST	REUSE

Context:

Segmentation: by card_pan AND terminal_id

Initiator policy: IGNORE

Meaning: A temporal window of 5 min is opened with the arrival of a first *Transaction* event. In this elapsed time we check for a TREND pattern over the amounts in the transactions per a single card and a single ATM. As in the previous case, we derive an event as it happens during the time window.

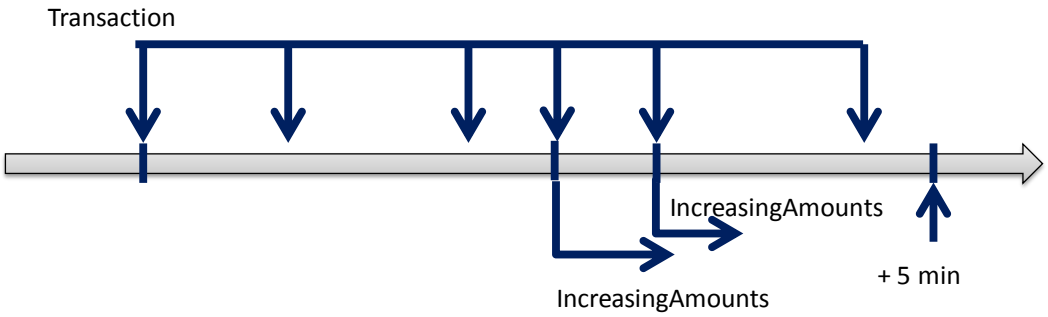

Figure 9: Context for IncreasingAmounts EPA

3.4.2.3 EPA3: IncreasingAmountsCardIndication

Motivation: The same as before, but this EPA's derived event is input to EPA4 and serves to eliminate multiple occurrences of a single card number and send a suspicious fraudulent card only once (the last one) to EPA4 which tests whether there are multiple credit cards frauds at the same ATM.

Event recognition process:

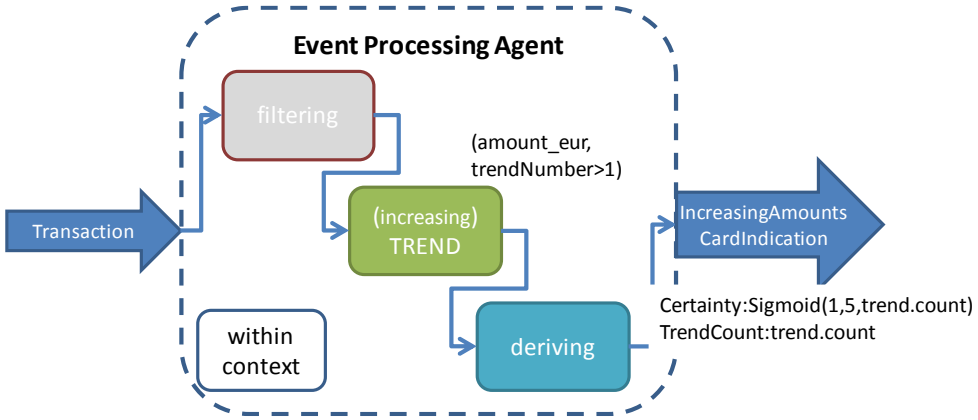

Figure 10: Event recognition process for IncreasingAmountsCardIndication EPA

Pattern policies:

Evaluation	Cardinality	Repeated	Consumption
DEFERRED	UNRESTRICTED	FIRST	REUSE

Context:

Segmentation: by card_pan AND terminal_id

Initiator policy: IGNORE

Meaning: A temporal window of 5 min is opened with the arrival of a first *Transaction* event. In this elapsed time we check for a TREND pattern over the amounts in the transactions per a single card and a single ATM. In this case, we derive a single event at the end of the window if the TREND pattern is satisfied (as per the *deferred* policy) and with the most updated *trend.count* at the moment of derivation

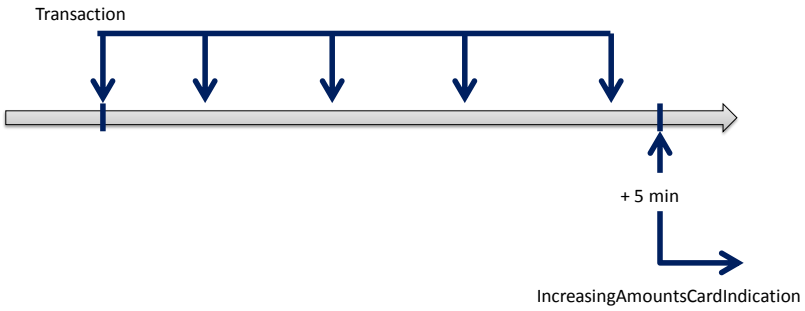

Figure 11: Context for IncreasingAmountsCardIndication EPA

3.4.2.4 EPA4: FraudAtATM

Motivation: Checking for suspicious ATMs. We are looking for at least two different cards with increasing amounts (a suspicious card) in a single ATM.

Event recognition process:

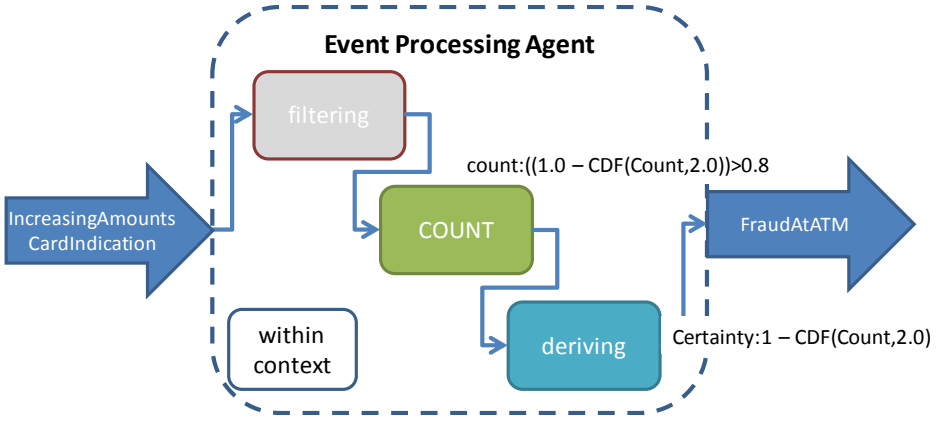

Figure 12: Event recognition process for FraudAtATM EPA

Count sums the number of the input event occurrences, while *count* is the assertion value for the COUNT pattern. We are detecting the pattern once the probability that we have at least two instances of *IncreasingAmounts* attacks on this terminal is more than 0.8. Since the *IncreasingAmounts* events are probabilistic, the assertion should take this into account in the counting calculation. The more input events we have (more *IncreasingAmounts* with different cards at this terminal), the higher the *Certainty* value of *FraudAtATM* derived event.

Pattern policies:

Evaluation	Cardinality	Repeated	Consumption
IMMEDIATE	UNRESTRICTED	FIRST	REUSE

Context:

Segmentation: by terminal_id

Initiator policy: IGNORE

Meaning: A temporal window of 5 min is opened with the arrival of a first *IncreasingAmountsCardIndication* event. In this elapsed time we check for a COUNT pattern per a single ATM. As in the previous case, we derive an event as it happens during the time window. Note, that since the COUNT pattern is probabilistic we might encounter more than two input events before deriving an output event.

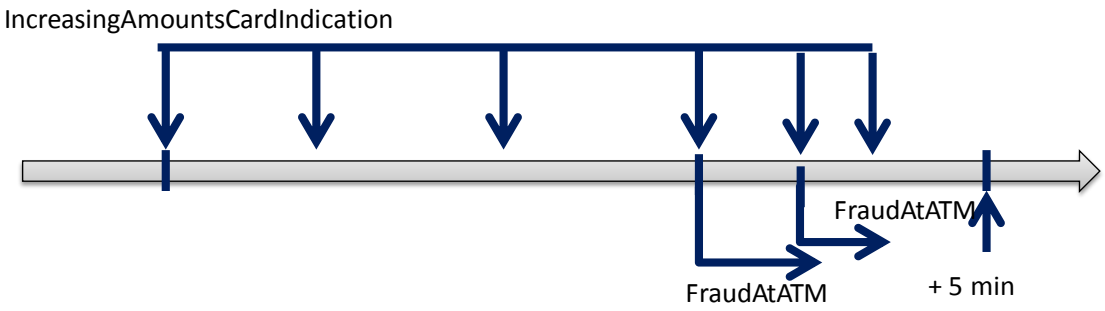

Figure 13: Context for FraudAtATM EPA

3.4.2.5 EPA5: Count (CVV attack)

Motivation: CVV attack case, a fraud is suspected whenever a large number of attempts (>3) using a card with wrong CVVs are made.

Event recognition process:

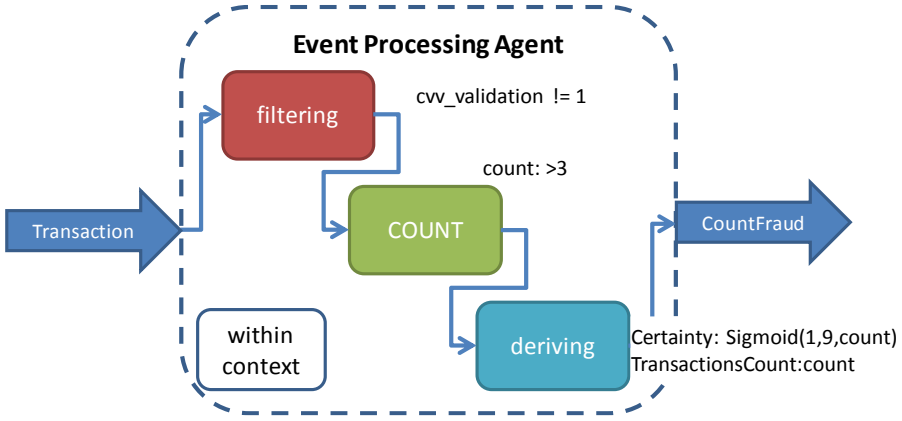

Figure 14: Event recognition process for Count (CVV attack) EPA

Pattern policies:

Evaluation	Cardinality	Repeated	Consumption
DEFERRED	UNRESTRICTED	FIRST	REUSE

Context:

Segmentation: by card_pan

Initiator policy: IGNORE

Meaning: A short temporal window of 2 min is opened with the arrival of a first *Transaction* event per card. At the end of the window the COUNT evaluation is made and a derived event is emitted if the pattern is satisfied.

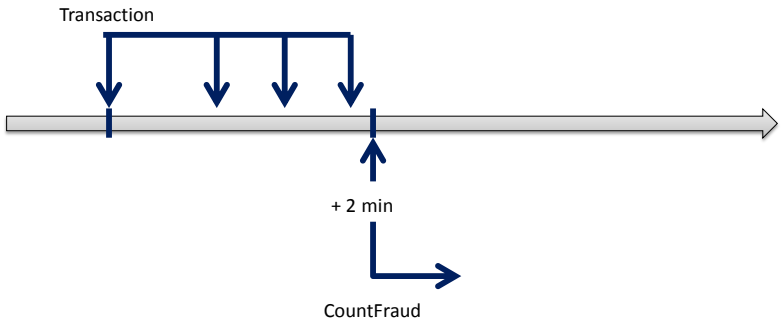

Figure 15: Context for Count (CVV attack) EPA

3.4.2.6 EPA6: CombinedCountTrendFraud (The TREND after COUNT case)

Motivation: The TREND after COUNT case, that is, we look for a case that an attempt for a CVV attack has been made preceding increasing amounts (TREND pattern). In other words, a "correct CVV" was found after several attempts that led to consecutive increasing amounts in the transactions. We have this EPA in addition to EPA1 so that we can combine different policies and derive events with payload

attributes generated from events in the matching set (EPA1 is a TREND operator with no access to the matching set events, see Section 3.5.53.4.4).

Event recognition process:

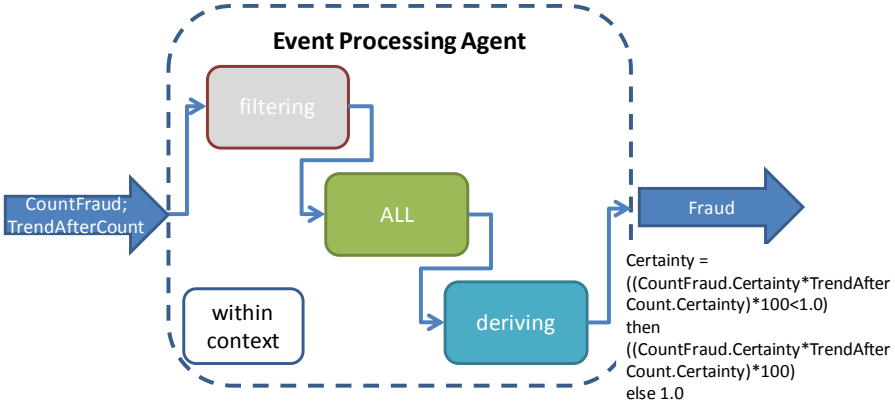

Figure 16: Event recognition process for CombinedCountTrendFraud (the TREND after COUNT case) EPA

Pattern policies:

Evaluation	Cardinality	Repeated	Consumption
IMMEDIATE	UNRESTRICTED	LAST	REUSE

Context:

Segmentation: by card_pan

Initiator policy: IGNORE

Meaning: A temporal window of 5 min is opened with the arrival of a first *CountFraud* event per card. During the time window, we look for pairs of a *CountFraud* and *TrendAfterCount* and emit a *Fraud* event whenever the pattern is satisfied.

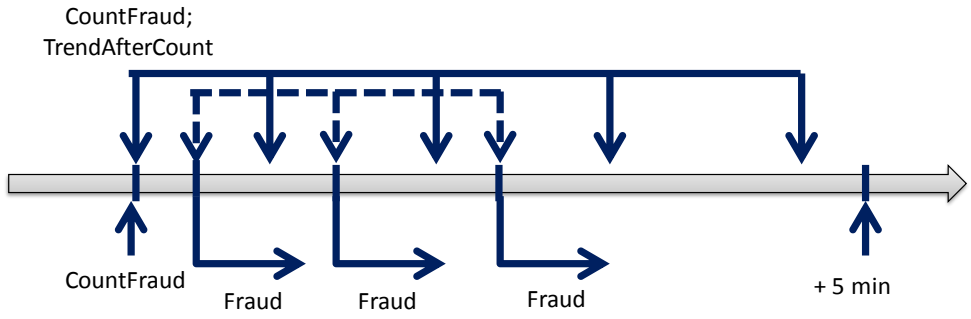

Figure 17: Context for CombinedCountTrendFraud (the TREND after COUNT case) EPA

3.4.2.7 EPA7: Sequence (Cloned Card)

Motivation: The cloned card case (for CP only), we check that two "close transactions" (a few minutes apart) cannot take place at two different physical locations (a location is represented by a country code, therefore, whenever there are two different locations, these are physically distant). T1 and T2 are aliases of the event type *Transaction*.

Event recognition process:

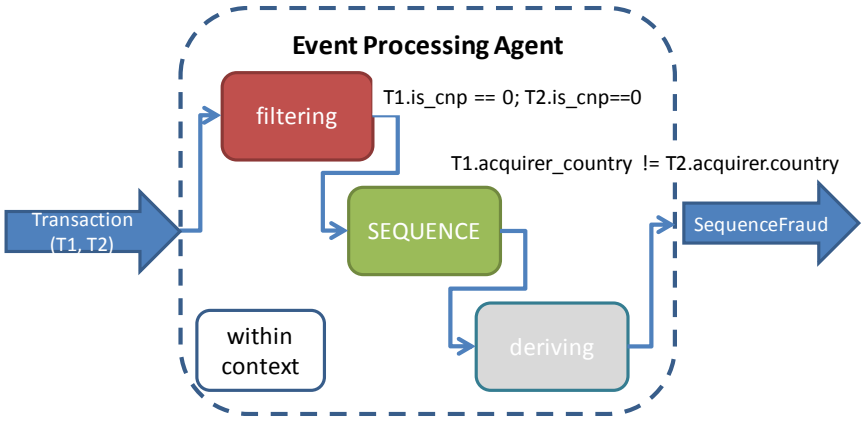

Figure 18: Event recognition process for Sequence (Cloned Card) EPA

Note that the *SequenceFraud* derived event has a certainty of 1 (in this case the fraud indication is of 100%) and therefore the *Certainty* attribute is not shown in the derivation step (the default is "1").

Pattern policies:

Evaluation	Cardinality	Repeated	Consumption
IMMEDIATE	UNRESTRICTED	T1=FIRST; T2=OVERRIDE	CONSUME

Context:

Segmentation: by card_pan

Initiator policy: IGNORE

Meaning: A temporal window of 5 min is opened with the arrival of a first *Transaction* event per card. During the time window, we look for pairs such as the location of the first transaction in the pair differs from the second transaction in the pair. In these cases, a *SequenceFraud* event is emitted. The *Transaction* events that participate in the pattern matching are those that belong to the CP case. The policies defined assure that every two consecutive events with different locations are checked in the pattern.

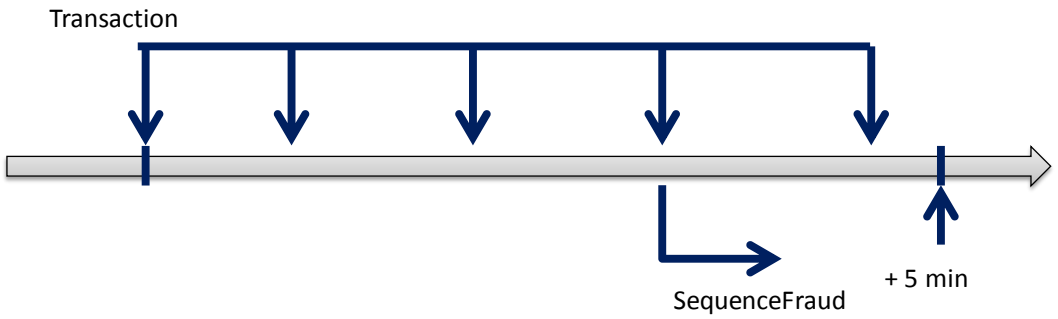

Figure 19: Context for Sequence (Cloned Card) EPA

3.4.3 Uncertainty

In our implemented EPN and first fraud use case event driven application we deal with two types of uncertainties: uncertainty in the derived event since the fact that a pattern is satisfied doesn't implicate a fraudulent activity with 100% assurance; and uncertainty in the input events. The latter stems from the fact that the input events are themselves derived event from the first type, thus the uncertainty is propagated forward.

3.4.3.1 Uncertainty in the derived event

In this case, the input events are certain (a Transaction event happens) but the derived event is not certain (e.g., the fact that we have 4 Transactions with increasing amount of money doesn't necessarily imply a fraud). EPA1, EPA2, EPA3, and EPA5 belong to this category of cases.

In this case we applied the Sigmoid function as aforementioned to derive the certainty of the derived event. Different values have been tested and the ones showed in the EPAs figures have been selected for the specific EPAs. For example, for EPA5 the values chosen are a:1 and b:9, for EPA2 a:1 and b:6 (Figure 20).

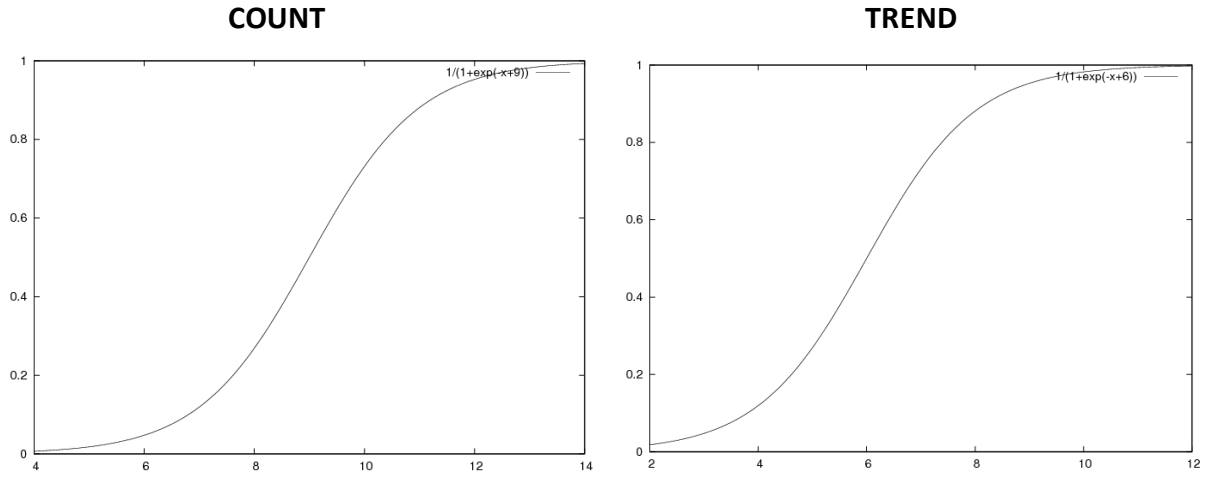

Figure 20: Sigmoid function results for EPA4 (COUNT) and EPA1 (TREND)

Henceforth we show the probability calculations for EPA6, the Trend after Count case (Equation 1):

The probability of a fraudulent event (F) given that a COUNT pattern (C) is satisfied (EPA5) is: P (F | C=c)

The probability of a fraudulent event given a TREND pattern (T) is satisfied (EPA1) is: P (F | T=tr)

The probability of a fraudulent event given first a COUNT and then a TREND (these are two independent variables) using Bayes formula:

$$P(F \mid C{=}c \cap T{=}tr) = P\,(F \cap C{=}c \cap T{=}tr) \,/\, P\,(C{=}c \cap T{=}tr) =$$

$$P(C{=}c \mid F)\cdot P(T{=}tr \mid F)\cdot P(F) \,/\, P(C{=}c \cap T{=}tr) = P\,(F|C{=}c)\cdot P(F|T{=}tr) \,/\, P(F)$$

Equation 1: Calculation of the probability for EPA6 (Trend after Count)

whereas we assume P(F) is given (in our tests, P(F) = 0.01)

3.4.3.2 Uncertainty in the input event(s)

When the input events contain uncertainty, this is propagated to the pattern matching step which becomes uncertain. EPAs 4 and 6 belong to this category.

For EPA4 we use a Cumulative Distribution Function (CDF) to calculate the certainty that a COUNT is detected. In EPA6 the ALL certainty is the multiplication of the certainty of the input events, since they are independent.

3.4.4 Summary

The first EPN for the fraud detection use case (see Figure 5) includes seven EPAs, one raw event, five situations (four probabilistic and one deterministic). Our implementation relies on PROTON's building blocks and capabilities and it might be possible that the same application will look differently when implemented in another CEP engine that uses different building blocks. For example, we might sometimes have same patterns that differ in context or policies and therefore we duplicate them as in the case of EPA1 and EPA2 that the pattern is the same (TREND) but their context is different. Another example is the TREND operator or pattern, which might not exist in another CEP engine. In this case, a workaround using existing building blocks or primitives needs to be established. Still, there are some constraints and limitations in PROTON's current language and implementation that dictated some of our design and implementation choices. The most noticeable example is the implementation of EPA6 which seems redundant. As a matter of fact, the most intuitive way to implement the "TREND after COUNT" EPA was just having EPA1 coming after EPA5. However, when the TREND operator (as in all operators of type aggregation) is satisfied, a derived event is emitted without the possibility to access the matching set events.

3.5 Implementation of first EPN for the traffic management use case

The overarching aim of the CEP module in this use case is to detect congestions or potential congestion situations in the Grenoble highway. To this end, a first EPN has been created with the collaboration of the use case owner partner CNRS with the goal of having something meaningful and representative, yet

doable to be achieved in the first year of the project. The outcome is an EPN consisting of seven EPAs shown in Figure 21 and detailed in the following Sections. For the sake of simplicity we only show the EPAs and the events flow in the network. Dotted lines represent events, other than input events, that are either initiators (in yellow) or terminators (in red) of a context. The PROTON JSON definitions file that comprises this EPN is provided as part of the software deliverable that accompanies this report. For further information on the use cases please refer to D7.1 "User Requirements and Scenario Definitions".

In the current EPN we want to fire situations in the following cases (for detailed descriptions of each EPA see Sections 3.5.3.1-3.5.3.7)

- A *Congestion* (EPA2) in a specific location is building-up.
- A *ClearCongestion* (EPA3) at a specific location is identified.
- A *PredictedCongestion*, that is, a forecasted congestion is identified at a specific location (EPA4 and EPA5).
- Calculations on sensor readings are emitted to be consumed by the decision making module (EPA6 and EPA7)

Note that for all the detections apart of EPA6, average measurements are taken (EPA1).

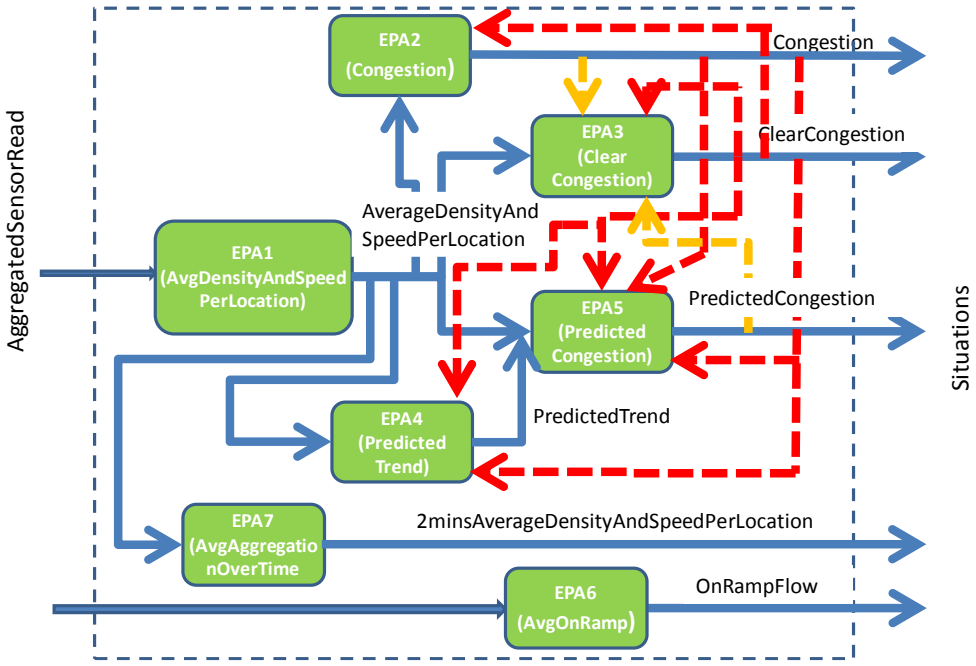

Figure 21: Traffic management use case initial EPN

3.5.1 Calculations of congestion, clear congestion, and "almost congestion" situations

We differentiate among three situations at a specific location using two parameters: density and speed. A *Congestion* (shown in red in Figure 22) exists if the density in a specific location is above a certain given value (*density_threshold1*) and the speed is below a certain given value (*speed_threshold1*). On the other hand, a congestion is over (*ClearCongestion*, shown in green), whenever the density is below a

certain given value (*density_threshold2*) and the speed is above a certain given value (*speed_threshold2*). We emit a probabilistic *PredictedCongestion* situation (in yellow) in between the *Congestion* and the *ClearCongestion* thresholds. Note that for the speed we added a new threshold (*speed_threshold3*) in order to narrow the limits for a *PredictedSituation*, but of course, any value between *speed_threshold1* and *speed_threshold2*, can be selected.

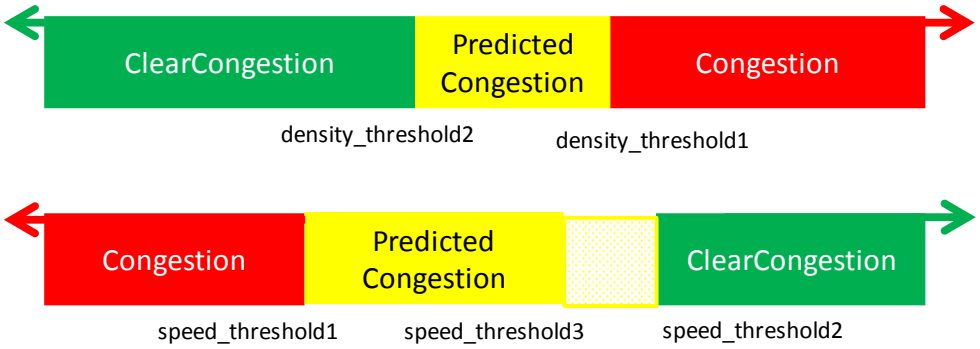

Figure 22: Illustrative diagram of the different traffic situations

Two main assumptions have been made:

- Density is naively calculated as flow divided by speed.
- Density values are computed first per lane and then they are aggregated to compute the average density value per location.
- Calculations are made for a single location or sensor and not for a segment or cell (distance between two consecutive locations).

Future versions of this application will take into account more sophisticated density calculations as well as segments.

3.5.2 Event types

Eight event types have been defined that comprise the event inputs, outputs/derived, and situations as shown in Figure 21. For the sake of simplicity we only show the user-defined attributes or the event payload and not the metadata (refer to Section 2.2.1.).

Although the names of concepts in can be determined freely by the application designer in PROTON, we use some naming conventions for the sake of clarity. We denote event types with capital letters. Built-in/metadata attributes start with a capital letter, as well as payload attributes that hold operators values, while payload attributes start with a lower letter. Table 3 shows the event definitions for the traffic management EPN. Note that the *problem_id* attribute is not part of the raw event payload and is intended for monitoring reasons by the decision module of the SPEEDD prototype. This module correlates decisions made and their effect by this attribute. At this stage, we assign the *location_id* value to the *problem_id* at the derivation step, but more complex expressions can be applied.

Note that the *AggregatedSensorRead* raw event includes more fields or attributes. We defined only the ones required for pattern detection in the current EPN implementation. When running in SPEEDD architecture, PROTON will ignore event attributes not specified in its JSON.

Table 3: Initial EPN for the traffic management use case

Event name	AggregatedSensorRead
Payload	location, lane, occupancy, vehicles, average_speed
Event name	Congestion
Payload	location, average_density, problem_id
Event name	PredictedCongestion
Payload	location, average_density, problem_id
Event name	ClearCongestion
Payload	location, problem_id
Event name	OnRampFlow
Payload	location, average_flow, average_speed, average_density
Event name	AverageDensityAndSpeedPerLocation
Payload	location, average_flow, average_density, average_ speed
Event name	2minsAverageDensityAndSpeedPerLocation
Payload	location, average_flow, average_speed, average_density
Event name	PredictedTrend
Payload	location, problem_id

3.5.3 Event processing agents

Henceforth, we describe the EPAs in the following order: Event name; motivation; event recognition process (following Figure 2); contexts along with temporal context policy; and pattern policies.

In the event recognition process we only show the steps that take place in the specific EPA, while the others are greyed. For the *filtering step* we show the filtering expression; for the *matching step* we denote the pattern variables; and for the *derivation step* we denote the values assignment and calculations. Please note that for the sake of simplicity we only show the assignments that are not copy of values (all other derived event attributes values are copied from the input events). For attributes, we just denote their names without the prefix of 'attribute_name.'

3.5.3.1 EPA1: AvgDensityAndSpeedPerLocation

Motivation: This EPA calculates averages of speed and vehicles to derive an average density over all the lanes in a certain location except for on-ramp lanes, which are treated differently (see D8.1 and D5.1 for further details).

Event recognition process:

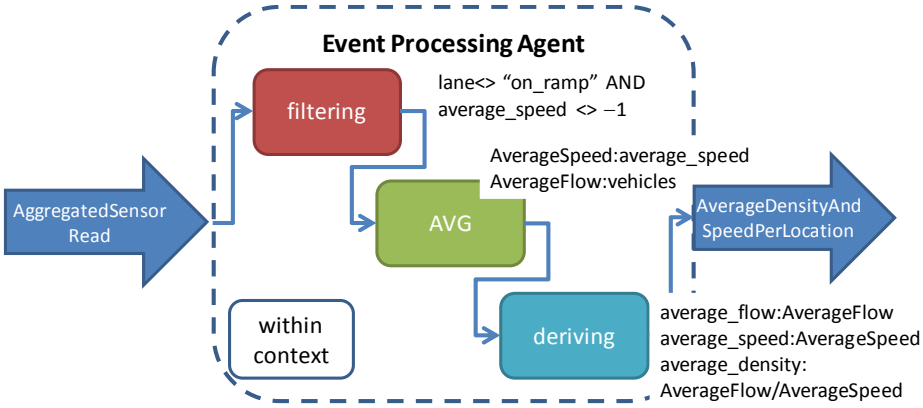

Figure 23: Event recognition process for AvgDensityAndSpeedPerLocation EPA

AverageSpeed and *AverageFlow* are computed variables of the AVG pattern.

Pattern policies:

Evaluation	Cardinality	Repeated	Consumption
DEFERRED	SINGLE	FIRST	CONSUME

Context:

Segmentation: by location_id

Initiator policy: IGNORE

Meaning: For each batch of raw events (AggregatedSensorRead) at each location, there is a derived event for the average density and speed. As there is one batch every 15 sec, the initiator policy doesn't play a role in this case.

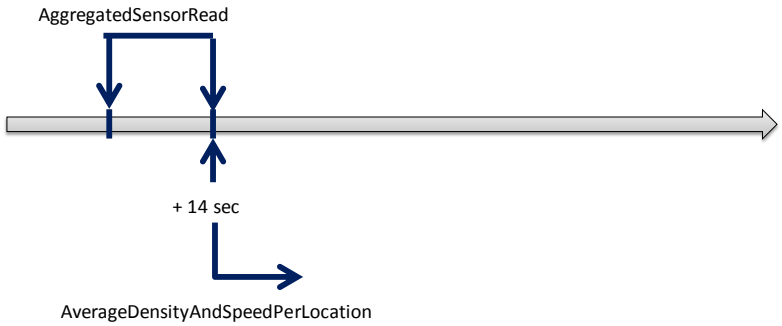

Figure 24: Context for AvgDensityAndSpeedPerLocation EPA

3.5.3.2 EPA2: Congestion

Motivation: To derive a congestion alert whenever the average density and speed are above and under specific given thresholds (labeled by 1). In this case the derived event has a probabilistic value of 1, since the congestion detected is already taking place. In our initial tests we used density_threshold1:0.95 and speed_threshold1:45.

Event recognition process:

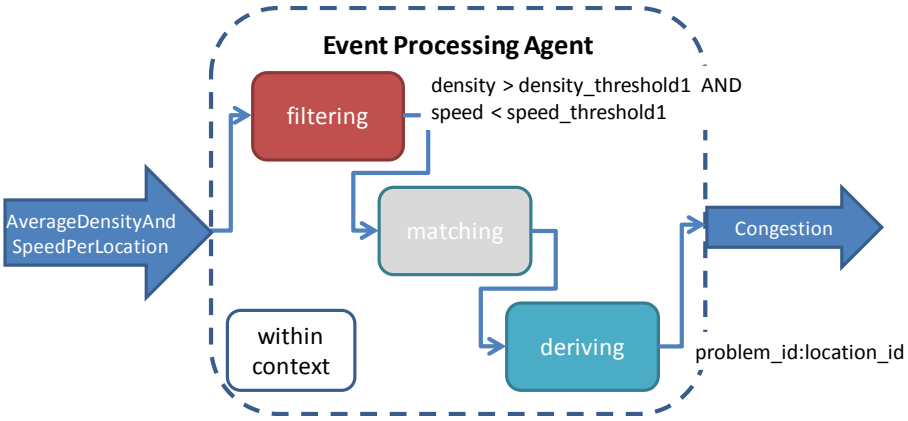

Figure 25: Event recognition process for Congestion EPA

Pattern policies:

Evaluation	Cardinality	Repeated	Consumption
IMMEDIATE	SINGLE	FIRST	CONSUME

Context:

Segmentation: by location_id

Initiator policy: IGNORE

Meaning: We open a single context for a location in order to detect an actual congestion. The context is closed with the *ClearCongestion* event, i.e., the congestion passed.

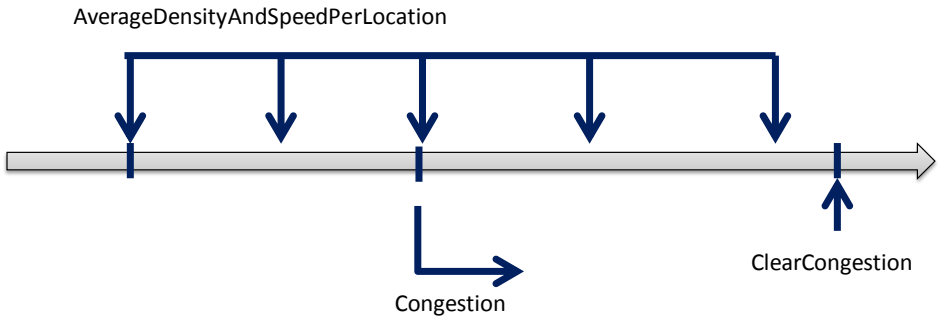

Figure 26: Event recognition process for Congestion EPA

3.5.3.3 EPA3: ClearCongestion

Motivation: A derived event is emitted whenever the flow is perceived as "normal", meaning the density and speed thresholds are in the "normal range". In our initial tests we used density_threshold2:0.80 and speed_threshold2:70.

Event recognition process:

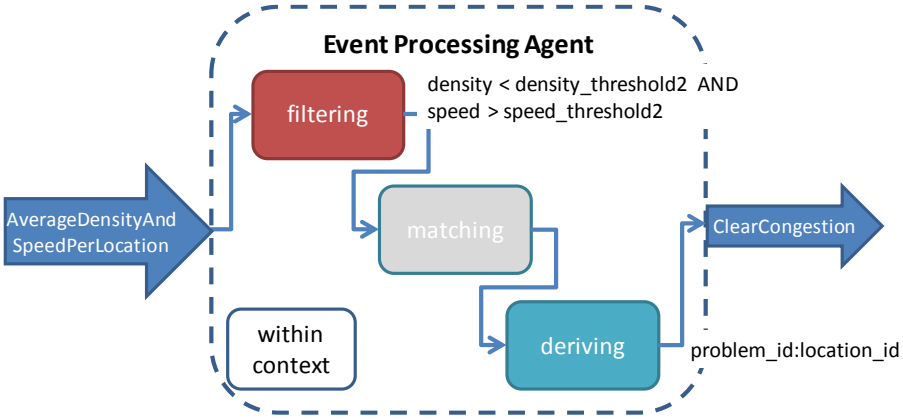

Figure 27: Event recognition process for ClearCongestion EPA

Pattern policies:

Evaluation	Cardinality	Repeated	Consumption
IMMEDIATE	SINGLE	FIRST	CONSUME

Context:

Segmentation: by location_id

Initiator policy: IGNORE

Meaning: We open a single context for a location in order to detect when an occurring congestion or predicted congestion goes away. To this end, the context is opened with either the *Congestion* or *PredictedCongestion* events (the first that comes) and is closed when a *ClearCongestion* is detected (the derived event also closes the open context).

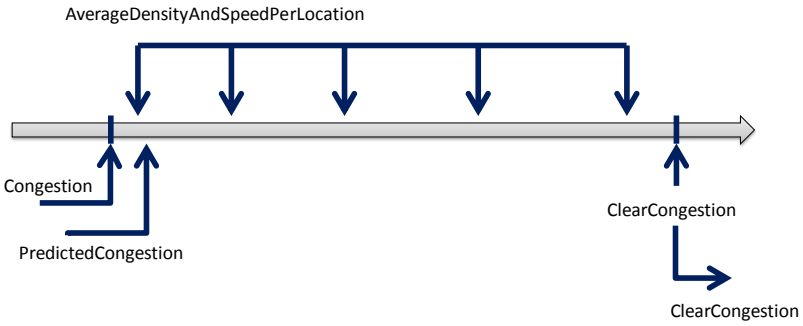

Figure 28: Context for ClearCongestion EPA

3.5.3.4 EPA4: PredictedTrend

Motivation: The CEP run-time engine will derive a *PredictedTrend* event (input to the *PredictedCongestion* EPA) whenever it detects an increase in the density values of at least 6 consecutive input events. The density values are still in the "normal range" so that neither a *Congestion* nor a *ClearCongestion* are detected. We used the value of 51 for our tests for speed_threshold3.

Event recognition process:

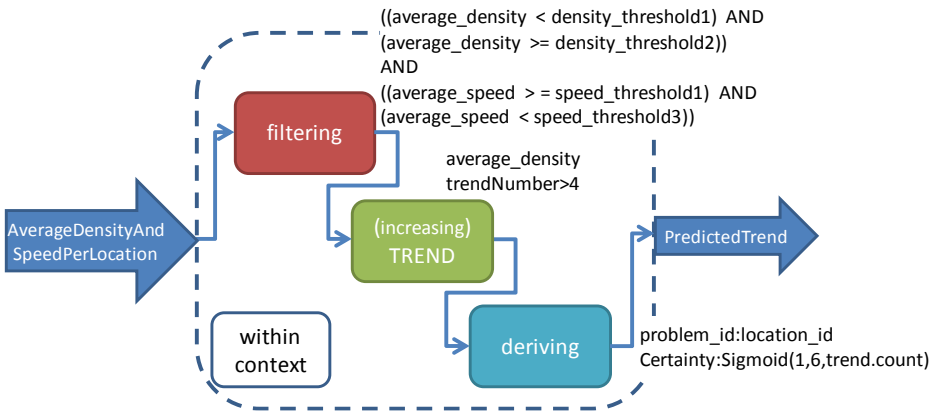

Figure 29: Event recognition process for PredictedTrend EPA

Pattern policies:

Evaluation	Cardinality	Repeated	Consumption
IMMEDIATE	UNRESTRICTED	FIRST	REUSE

Context:

Segmentation: by location_id

Initiator policy: IGNORE

Meaning: We open a single context for a location in order to detect an increasing TREND pattern. To this end, the context is opened with the first input event that comes and is closed when either a *Congestion* or *ClearCongestion* is detected for the same location. As we use the IMMEDIATE and UNRESTRICTED

policies, we derive a *PredictedTrend* event for each TREND encountered; with increasing *Certainty* value if the buildup continues (density continues to rise and speed continues to decrease).

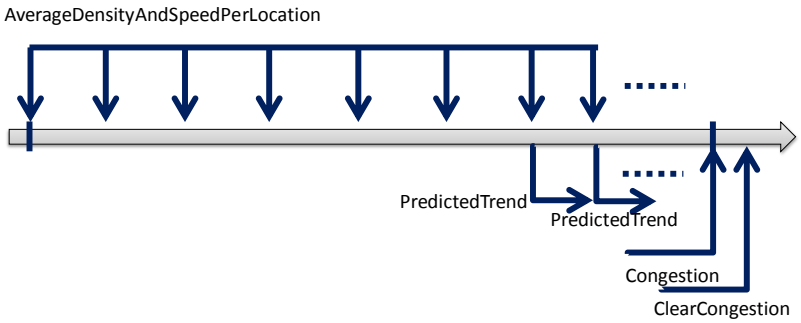

Figure 30: Context for PredictedTrend EPA

3.5.3.5 EPA5: PredictedCongestion

Motivation: The goal is to derive a *PredictedCongestion* event, that is, we believe there is a chance that a congestion will take place in the near future. We use the TREND pattern from EPA4 as input and derive an event that gets the uncertainty of the last *PredictedTrend* in the pair of input events (see repeated policy below). We have this EPA in addition to EPA4 so that we can combine different policies and derive events with payload attributes generated from events in the matching set (EPA4 is a TREND operator with no access to the matching set events, see 3.5.5).

Event recognition process:

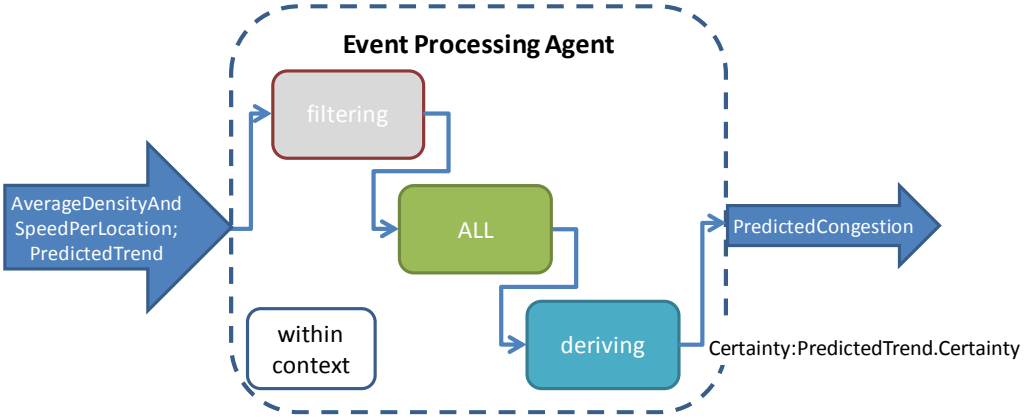

Figure 31: Event recognition process for PredictedCongestion EPA

Pattern policies:

We denote *AverageDensityAndSpeedPerLocation* as e1 and *PredictedTrend* as e2.

Evaluation	Cardinality	Repeated	Consumption
IMMEDIATE	UNRESTRICTED	e1: OVERRIDE e2: LAST	e1: REUSE e2: CONSUME

Context:

Segmentation: by location_id

Initiator policy: IGNORE

Meaning: The context is opened with the first *AverageDensityAndSpeedPerLocation* input event that comes and is closed when either a *Congestion* or *ClearCongestion* is detected for the same location. Whenever a *PredictedTrend* input event comes, a *PredictedCongestion* derived event is emitted. As we use the IMMEDIATE and UNRESTRICTED policies, we can derive more than one event. We use the LAST policy, so that the last *PredictedTrend* event is used for the ALL pattern (assuming that the probabilities are going up as we have more increasing densities).

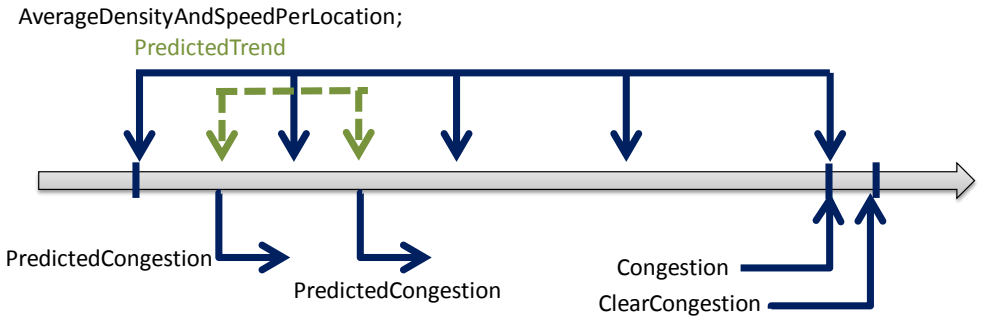

Figure 32: Context for PredictedCongestion EPA

3.5.3.6 EPA6: AvgOnRamp

Motivation: To derive average values of on-ramp lanes every two minutes for the decision module.

Event recognition process:

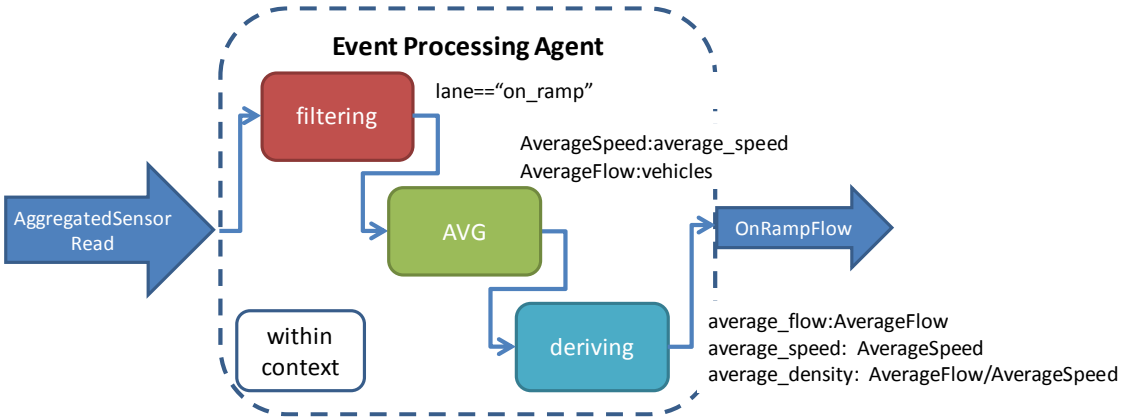

Figure 33: Event recognition process for AvgOnRamp EPA

Pattern policies:

Evaluation	Cardinality	Repeated	Consumption
DEFERRED	SINGLE	FIRST	CONSUME

Context:

Segmentation: by location_id

Initiator policy: ADD

Meaning: In each location, for each input event (sliding/overlapping temporal windows) we open a temporal context and perform average calculations. The temporal contexts are closed after two minutes.

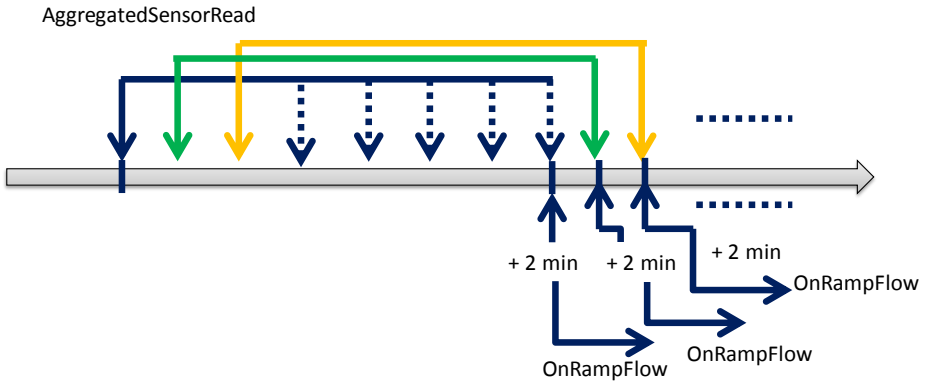

Figure 34: Event recognition process for AvgOnRamp EPA

3.5.3.7 EPA7: AvgAggregationOverTime

Motivation: To derive average values for all lanes every two minutes for the decision module.

Event recognition process:

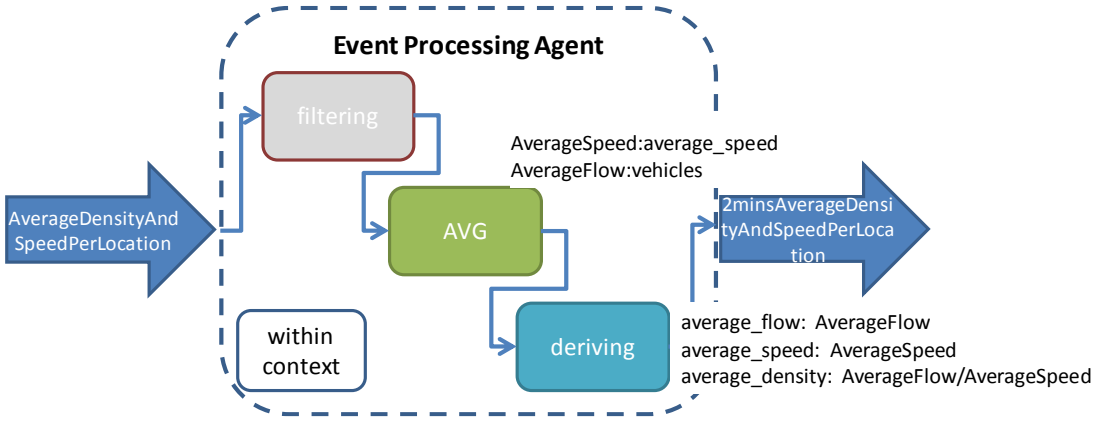

Figure 35: Event recognition process for AvgAggregationOverTime EPA

Pattern policies:

Evaluation	Cardinality	Repeated	Consumption
DEFERRED	SINGLE	FIRST	CONSUME

Context:

Segmentation: by location_id

Initiator policy: ADD

Meaning: In each location, for each input event (sliding/overlapping temporal windows) we open a temporal context and perform average calculations. The temporal contexts are closed after two minutes.

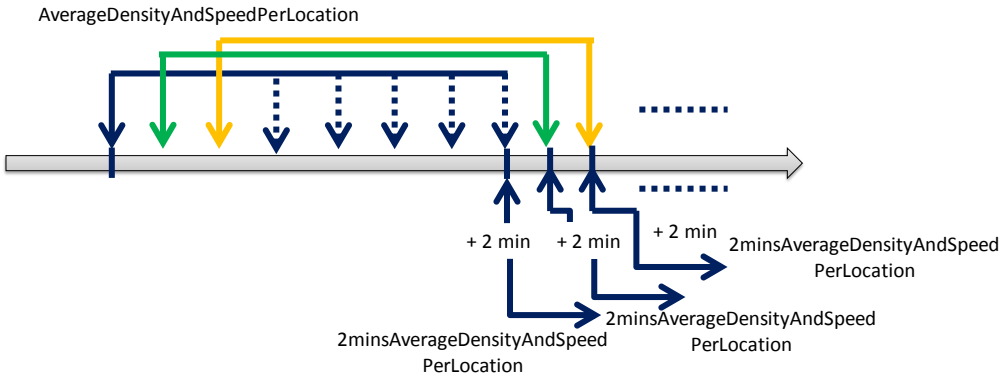

Figure 36: Event recognition process for AvgAggregationOverTime EPA

3.5.4 Uncertainty

Similar to the fraud use case, our implemented EPN and first traffic management use case event driven application copes with uncertainty that stems from the input events and from the derived events.

Note that the derived events in EPAs 1, 2, 3, 6, and 7 have a certainty value equals to 1. EPAs 1, 6, and 7 are calculations on input events required by the decision module. EPA2 measures a congestion situation while is already taking place, while EPA3 detects that the congestion is over, again when it already takes place.

3.5.4.1 Uncertainty in the derived event

In this case, the input events are certain (a sensor reading event happens) but the derived event is not certain (e.g., the fact that we have 5 sensor readings that show an increase in the density, doesn't necessarily imply there will be a traffic congestion for sure). EPA4 belongs to this category of cases. As in the fraud use case, we apply the Sigmoid function to calculate the probability of the occurrence of a derived event.

3.5.4.2 Uncertainty in the input event(s)

When the input events contain uncertainty then the pattern becomes uncertain. EPA5 belongs to this category. In EPA5 the uncertainty value in the input event is propagated to the derived event in the ALL pattern.

3.5.5 Summary

The first EPN for the traffic management use case (see Figure 21) includes seven EPAs, one raw event, five situations (one probabilistic and four deterministic). As the same as for the fraud detection, our implementation relies on PROTON's building blocks and capabilities and it might be possible that the same application will look differently when implemented in another CEP engine that uses different building blocks. There are also constraints that PROTON programmatic language and implementation imposes when designing and building a new event-driven application. As in the case of the fraud detection EPN, we need EPA5 so we can derive an event that includes payload values of the TREND operator matching set (EPA4).

4 Performance evaluation

In addition to functional tests that assured the results received are in accordance to the expected outputs in terms of correctness and functionality, tests have been run to assess the throughput and latency of the two EPNs for our two use cases applications.

The machine used for the tests purposes was a Lenovo Thinkpad with the following characteristics:

Intel(R) Core(TM) i7-2720QM CPU @ 2.20GHz 2.20 GHz
8.00 GB
64-bit Operating System

The tests were performed on Oracle JRE v 1.6.0_39, with the heap space allocated as 40MB initially and allowed to grow to max 1 GB, with a maximum PermGen size of 128MB.

The tests were carried out under the following constraints which influenced (and eventually deteriorated) the performance:

- The Thinkpad that served for the tests also run additional heavy load programs in parallel to the tests (e.g., IBM Lotus Notes), due to constraints of the infrastructure for performance measurement.
- The raw events and derived events were read from an input file and were written to an output file on the same machine, while the event processing application was running, and at the same JVM process Massive I/O and use of process memory can significantly affect the performance, especially during throughput testing.
- Maximum CPU load reached during testing was 40%. We believe that tuning the configuration so that CPU reaches the max load, while playing with CPU affinity to allow priority for the event processing and to decrease amount of preempts, can significantly improve performance.
- Lack of system warm-up. Performing warm-up for JVM compiler optimization procedures will give results closer to real-life performance.

Considering all of the above, we treat the current performance results as an initial indication of a lowest bound for performance metrics for the CEP component. We believe we will get more accurate metrics using suitable infrastructure and system tuning, and running the CEP tool in a distributed manner on top of STORM.

The following Sections describe our experiments and results.

4.1 Throughput

4.1.1 Datasets

The throughput was calculated using a dataset consisting of 10,000 real data raw events. For the traffic management use case we used the September 2014 input and for the fraud use case the anonymized data, both provided by the use cases partner.

4.1.2 Throughput results for the fraud detection use case

In order to test throughput the entire EPN has been applied (see Figure 5) with maximum load. We denote by *start_time* the detection time of the first input event and by *end_time* the detection time of the last derived event. We calculated the time between the injection of the first raw event and the derivation of the last raw event. Therefore, the throughput measures the amount of events processed through the entire EPN divided by (*start_time* – *end_time*).

Ten runs of 10,000 raw events each were carried out. The mean throughput over these runs was 1.948 milliseconds per event, i.e., 513 events per second.

4.1.3 Throughput results for traffic management use case

In order to test throughput the entire EPN has been applied (see Figure 21) with maximum load. We denote by *start_time* the detection time of the first input event and by *end_time* the detection time of the last derived event. We calculated the time between the injection of the first raw event and the derivation of the last raw event. Therefore, the throughput measures the amount of events processed through the entire EPN divided by (*start_time* – *end_time*).

Ten runs of 10,000 raw events each were carried out. The mean throughput over these runs was 1.73 milliseconds per event, i.e., 575 events per second.

4.2 Latency

4.2.1 Datasets

The latency was measured with simulated datasets, each 10,000 raw events. We haven't used real-data since we tested specific patterns and needed to be sure that these are represented enough in the datasets. For each use case, we selected one "predictive pattern" (i.e., the derived event is probabilistic or has a certainty value < 1) and one "detection pattern", i.e., the derived event has a certainty value = 1. When discussing predictive patterns, we can also relate to the predictive horizon, meaning, to "how far" in the future the predictive event can hold.

4.2.2 Latency results for the fraud detection use case

4.2.2.1 First case: IncreasingAmounts

In this scenario, EPA2 (Increasing Amounts) latency has been tested (see Figure 5). This EPA tests for a TREND pattern of at least two consecutive *Transactions* with increasing amounts. The latency was calculated as the elapsed time between the detection time of the last *Transaction* event in the matching set (at the moment the input event is injected into CEP engine) and the detection time of the *IncreasingAmounts* derived event.

An overall of 10,000 events have been processed that included around the 70% of the pattern satisfaction.

The forecasting horizon in this case depends mainly on the fraud given probability (assumed to be 0.01) and the TREND threshold (amount of events in the matching set). These can be configured to better fit the real data.

The latency mean found was: 110.8 milliseconds and the 90% percentile 118 milliseconds.

4.2.2.2 Second case: SequenceFraud

In this scenario, EPA7 (Cloned Card) latency has been tested (see Figure 5). This EPA detects two consecutive *Transactions* with the same credit card (present) at different locations. The latency is calculated from the raw event injection until the sequence pattern detection.

Note that the performance of a SEQUENCE pattern is heavily influenced by the policies undertaken, specially the *repeated* policy (see Section 2.1.4), with worst performance for the *every* policy which tests all possible combinations of the participating events. In our case, we apply the *first* and *override* options meaning we have a 1:1 match and no iteration over possible other combinations.

The input file comprises 10,000 raw events, with around 10% of events that satisfy the SEQUENCE pattern.

The latency mean found was: 112 milliseconds and the 90% percentile 125 milliseconds.

4.2.3 Latency results for traffic management use case

4.2.3.1 First case: PredictedTrend

In this scenario, two EPAs have been tested (see Figure 21), EPA1 (AvgDensityAndSpeedPerLocation) and EPA4 (Predicted Trend), that is, we derive the *PredictedTrend* event with a probability for a congestion. EPA1 receives the raw events (*AggregatedSensorRead*) as input events and emit the average measures (*AverageDensityAndSpeedPerLocation*) used then by EPA4 for the TREND pattern (at least five instances with increasing *density* values). The context window was shortened to 3 min for practicality. Our data set included around a 1% of TREND detection.

Latency was measured from the moment the trend situation happens (5th occurrence of *AverageDensityAndSpeedPerLocation* with increasing density enters the system) until *PredictedTrend* is

derived. Therefore, the detection time of the *AverageDensityAndSpeedPerLocation* event was subtracted from the detection time of the *PredictedTrend* event.

An overall of 10,000 raw events have been processed.

Regarding the forecasting horizon, the *PredictedCongestion* event is fired after 5 successive average calculations with increasing *density*. Each average calculation is based on input readings every 15 seconds. We consider a traffic buildup from the moment events start entering the matching set. In our EPN, it takes 75 seconds to detect and forecast a building congestion (from traffic buildup until congestion). The amount of time between a prediction and an actual congestion heavily depends on the thresholds values for the buildup. Note that these are configurable.

The latency mean found was: 110.5 milliseconds and the 90% percentile 110 milliseconds.

4.2.3.2 Second case: Congestion and ClearCongestion

In the second scenario, two EPAs have been tested (see Figure 21), EPA1 (AvgDensityAndSpeedPerLocation), EPA2 (Congestion) and EPA3 (ClearCongestion). In this scenario, the *AggregatedSensorRead* raw events pass through EPA1 in which averages are calculated and from which *AverageDensityAndSpeedPerLocation* derived events serve as input for detect either a clearance or an appearance of congestion. The dataset contained 10,000 raw events.

The latency is measured as the elapsed time between the detection time of *AverageDensityAndSpeedPerLocation* to the detection time of *Congestion* and *ClearCongestion* events. The percentage of detection within the dataset was around 5%.

The latency mean found was 101.253 milliseconds and the 90% percentile 102 milliseconds.

4.3 Performance evaluation results summary

Performance evaluation has been made on two criteria: throughput and latency. These have been measured on partial and complete current EPNs of the two use cases with simulated as well as with real raw events.

Our main objective in the project is to improve the current situation with the use of SPEEDD technology. In the scope of this first deliverable related to the CEP component, we have just made a first step towards this direction. Any attempt to generalize over the results so far will be somehow missing the main goal and impractical due to several reasons:

- The EPNs are still initial and partial, not covering all possible patterns. The reality that serves as benchmark is much more complex and contains more patterns that our component will simply miss and not detect at this stage.
- Improvements stem from the overall system, especially from the interaction and synergy between the CEP and decision making components. Evaluation in isolation might hurt the potential overall performance achieved.

- In the fraud detection use case, the Feedzai partner owns a dedicated operational system to detect fraud. Applying a CEP generic research prototype to "compete" with it in terms of accuracy of results (false positives and false negatives) seems impractical. The benefit should be associated with the fact that potential fraudulent transactions can be detected beforehand.
- On the other hand, in the traffic management use case, there is no real time operational system at all; therefore any improvement will be evident.

Still, we believe that the results so far can give an indication of what can be achieved and be a lower bound of performance. As aforementioned, the tests have been carried out on a single machine running other heavy processes in parallel. The results (both the throughput and the latency) can be significantly improved if running on a machine with a clean state, multiple cores, and when tuning for performance.

In addition, the throughput can be further beneficiary when running on a distributed environment like STORM with N nodes (refer to D3.1). Assuming the contexts distributions are more or less uniform, the throughput can be approximately the throughput on a single machine*N (minus communication and infrastructure overheads)

5 Summary and future steps

In this document, we present the first version of the complex event module under uncertainty in SPEEDD. The report comprises the extensions introduced so far in the event processing component tooling to cope with uncertainty driven by both the use cases and the decision module requirements. A detailed description of the first event driven applications for the two use cases in the project is given. We also show preliminary evaluation results of the proposed technologies.

The inclusion of uncertainty aspects, mainly manifested in the run-time module, impacts all levels of the architecture and logic of an event processing engine. Even at this stage, the current implemented EPNs possess a high level of sophistication and complexity. In order to cope with uncertainty, the following extensions have been implemented in PROTON:

- New built-in attributes have been added (*Certainty*)
- Operands types have been extended to support distributions (e.g., Sigmoid)
- Additional built-in functions have been added (e.g., CDF)
- Operators have been extended to support events with probabilistic attribute values (e.g., COUNT and TREND)

Important to note, these are not ad-hoc extensions to support specific use cases requirements, but generic building blocks that make PROTON first-of-a-kind CEP engine capable to deal with uncertainty and to derive forecasted events.

In the next months we plan to continue extending the support for uncertainty driven by the use cases requirements. This will be reflected in new patterns and the presence of uncertainty in the input or data sets (e.g., missing or erroneous attribute values). Advanced event processing networks will reflect:

- The addition of new patterns
- The fine tuning of the entire networks including different thresholds values
- Other probabilistic functions in addition to Sigmoid.
- In the traffic management use case, taking into account the following aspects:
  - Noise in the input data, e.g., taking into account the "-1" that are currently filtered out (the -1 indicates no speed value calculation is available).
  - Calculations per cell, not per sensor/location; on-ramp lanes;
  - Derivation of more fine-tuned events, such as "heavy/medium/low" congestion.
  - More sophisticated formulae for density.
  - More sophisticated values for problem_id instead of location_id

Our next reports at M22 and M32 will present the advances made in the coming months.

6 References

[1]. Etzion O. and Niblet P. *Event processing in action*. Manning, 2010

[2]. Proton user guide and programmer guide available at: https://forge.fi-ware.org/plugins/mediawiki/wiki/fiware/index.php/CEP_GE_-_IBM_Proactive_Technology_Online_User_and_Programmer_Guide

[3]. Open specification (REST api) available at: http://forge.fi-ware.org/plugins/mediawiki/wiki/fiware/index.php/Complex_Event_Processing_Open_RESTful_API_Specification

[4]. Installation and administration guide, available at: https://forge.fi-ware.org/plugins/mediawiki/wiki/fiware/index.php/CEP_GE_-_IBM_Proactive_Technology_Online_Installation_and_Administration_Guide

[5]. Toshniwal A., Taneja S., Shukla A., Ramasamy K., Patel J. M., Kulkarni S., Jackson J., Gade K., Fu M., Donham J., Bhagat N., Mittal S., Ryaboy D. *Storm@twitter*. SIGMOD Conference 2014, pp. 147-156.

[6]. Kreps J., Narkhede N., and Rao J. Kafka: A distributed messaging system for log processing. In Proceedings of 6th International Workshop on Networking Meets Databases (NetDB), Athens, Greece, 2011.

[7]. Artikis, A. Etzion, O. Feldman, Z. and Fournier, F. *Event Processing under Uncertainty*. DEBS 2012.

[8]. Wasserkrug, S. Gal, A. and Etzion, O., *A model for reasoning with uncertain rules in event composition*. Proceedings of the 21st Conference Annual Conference on Uncertainty in Artificial Intelligence (UAI). 2005.

[9]. Wasserkrug, S. Gal, A. Etzion, O. and Turchin, Y., *Efficient processing of uncertain events in rule-based systems*. IEEE Transactions on Knowledge and Data Engineering. 2012.

[10]. Engel Y. and Etzion. O. *Towards Proactive Event-Driven Computing*. DEBS 2011, 125-1136.

[11]. Engel Y., Etzion. O., and Feldman. Z., *A Basic Model for Proactive Event-Driven Computing*. DEBS 2012, 107-118.

[12]. Hastie T., Tibshirani R., and Friedman J., *The elements of statistical learning*. Vol. 2. No. 1. New York: Springer, 2009.