Industry Paper: The Uncertain Case of Credit Card Fraud Detection

Ivo Correia
Feedzai
Av. João II, Lote 1.06.2.2
1990-095 Lisboa, Portugal
+351 211 985635
ivo.correia@feedzai.com

Fabiana Fournier
IBM Research – Haifa
Haifa University Campus
Haifa 3498825, Israel
+972 4 8296489
fabiana@il.ibm.com

Inna Skarbovsky
IBM Research – Haifa
Haifa University Campus
Haifa 3498825, Israel
+972 4 8281330
inna@il.ibm.com

ABSTRACT

Uncertainty is inherent in many real-time event-driven applications. Credit card fraud detection is a typical uncertain domain, where potential fraud incidents must be detected in real time and tagged before the transaction has been accepted or denied. We present extensions to the IBM Proactive Technology Online (PROTON) open source tool to cope with uncertainty. The inclusion of uncertainty aspects impacts all levels of the architecture and logic of an event processing engine. The extensions implemented in PROTON include the addition of new built-in attributes and functions, support for new types of operands, and support for event processing patterns to cope with all these. The new capabilities were implemented as building blocks and basic primitives in the complex event processing programmatic language. This enables implementation of event-driven applications possessing uncertainty aspects from different domains in a generic manner. A first application was devised in the domain of credit card fraud detection. Our preliminary results are encouraging, showing potential benefits that stem from incorporating uncertainty aspects to the domain of credit card fraud detection.

Categories and Subject Descriptors

I.2.3 [**Artificial Intelligence**]: Deduction and Theorem Proving - *Uncertainty, fuzzy, and probabilistic reasoning*; K.4.4 [Computers and Society]: Electronic Commerce - *Payment schemes.*

General Terms

Design and Experimentation

Keywords

Complex event processing, pattern matching, uncertainty, credit card fraud detection

1. INTRODUCTION AND MOTIVATION

In most complex event processing (CEP) systems there is an underlying assumption that data is precise and certain, or that it has been cleansed before processing [2]. Another basic assumption is that the event rules or patterns are always deterministic. However, in many real-time, event-driven applications, these assumptions don't hold. Consider, for example, a (derived) event indicating credit card fraud based on a large number of withdrawals from a customer's account within a short time frame, in increasing amounts. This activity may indicate a fraud with some probability; however, it may also indicate a one-time behavior for this customer.

DEBS'15, June 29 - July 3, 2015, OSLO, Norway
Copyright 2015 ACM 978-1-4503-3286-6/15/06...$15.00.

DOI: http://dx.doi.org/10.1145/2675743.2771877

In specific applications, uncertainty can be found in the input events, in the output events, or in both [2]. In addition, there are cases in which the pattern itself can be uncertain.

We distinguish between three types of uncertainty in the input events:

1. Uncertainty in event content: one or more event attributes have probabilities attached to them.
2. Uncertain event occurrence: events handled as atomic units are represented along with their occurrence probability.
3. Uncertain rules: alternative events may be triggered based on rule elements, and are given probability values.

Consequently, complex event processing engines are required to accommodate and propagate the uncertainty from input events to the output (complex) events.

As credit card becomes the most popular mode of payment for both online and regular purchases, cases of credit card fraud are also on the rise. Financial fraud has increased significantly with the development of modern technology and the global superhighways of communication, resulting in the loss of billions of dollars worldwide each year. According to the BI Intelligence study from March 5, 2014[1], the cost of global payment card fraud grew by 19% in 2013, to$14 billion. The cost of U.S. payment card fraud grew by 29% to $7.1 billion, while in the rest of the world; card fraud grew by 11% to $6.8 billion. FICO published its latest map of card fraud in Europe, showing that card fraud losses in 2013 for the 19 European countries studied reached €1.55 billion[2].

Fraud detection, including the real-time detection of patterns across multiple locations or cards, was recently recognized as one of the main activities required for embedding real-time intelligence into bank operations [8][9]. Fraud detection in banking and credit card processing depends on correlating events across channels and accounts; this must be carried out in real time

[1] http://www.businessinsider.com/the-us-accounts-for-over-half-of-global-payment-card-fraud-sai-2014-3

[2] http://www.fico.com/en/newsroom/fico-infographic-european-card-fraud-losses-hit-new-high

to prevent losses before they occur. Therefore, fraud detection has been, and still is, one of the main classical applications for complex event processing ([11], [9][15][10][8]).

Fraud detection is a domain that possesses inherent uncertainty. Our previous example shows that certain "suspicious" behaviors don't necessarily indicate fraud.

We present a complex event processing tool that can efficiently handle the types of uncertainty found in credit card fraud, and show an implementation using real data. The tool is capable of handling uncertainty in a generic manner, and not ad-hoc.

The remainder of this paper is organized as follows: Section 2 introduces the semantics applied in the proposed tooling. Section 3 discusses the credit card fraud use case and event-driven application. In Section 4 we describe the actual event-driven application for the credit card fraud use case with the proposed tool, and Section 5 describes the results obtained by running the implementation over a real data set. We discuss related work in Section 6. Section 7 concludes the paper and identifies future research directions.

2. PRELIMINARIES

Each complex event processing engine uses its own terminology and semantics, and our work follows the semantics presented by Etzion and Niblet [6]. Below are some of the main terms used in our work and implemented in our complex event processing (Section 4.1).

2.1 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 p.m. a customer named John Doe withdrew 100 euros from 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 refer 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 *Transaction* 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. For example: *Increasing amounts have been withdrawn from 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). For example: a *Fraud transaction*.

2.2 Context

Context is a named specification of conditions that groups event instances so they can be processed in a related way. In this work, we employ the two most commonly used dimensions: temporal and segmentation. 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 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. For example, consider a single stream of input events, in which each event contains a credit card identifier attribute. The value of this attribute can be used to group events so that each credit card has a separate context partition. Each context partition contains only events related to that credit card, so the behavior of each card can be tracked independently of the other cards. A **composite context** is a context composed of 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.

2.3 Event Processing Network (EPN)

An **Event Processing Network (EPN)** is a conceptual model that describes 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, 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. The intermediary processing between producers and consumers in every installation is made up of several functions. Often, the same function is applied to different events for different purposes at different stages of the processing.

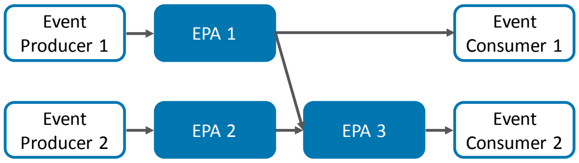


Figure 1. An event processing network

2.4 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, also known as the **pattern matching process** or **event recognition** (see Figure 2). All three steps are optional, but at least one must be performed inside an EPA.

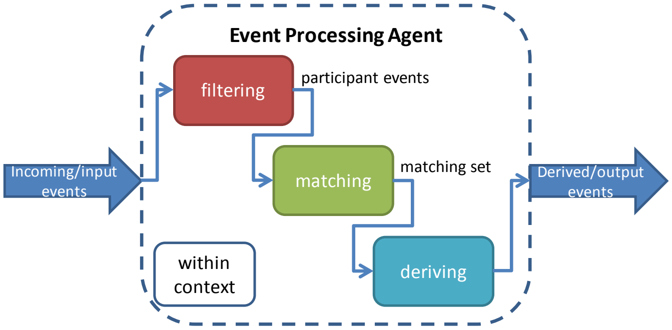


Figure 2. Event recognition process in an EPA

Pattern matching process steps:

- **Filtering:** 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**.
- **Matching:** takes all events that passed the filtering step 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**.
- **Deriving:** takes the output from the matching step and uses it to derive the output events by applying derivation formulas.

An **event pattern** is a template specifying one or more combinations of events. Given a collection of events, if one or more subsets of those events match a particular pattern, it can be said that such a subset satisfies the pattern. Some common examples of patterns:

- **Sequence**: at least one instance of all participating event types must arrive in a specified order for the pattern to be matched.
- **Count**: the number of instances in the participant event set satisfies the pattern's number assertion.
- **All**: 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 an observed specific attribute value.
- **Absence**: a specified event(s) must not occur within a predefined time window. The matching set in this case is empty.
- **Average**: the value of a specific attribute, averaged over all participant events, satisfies the average threshold assertion.

2.5 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. Our event processing engine supports five types of policies:

Evaluation policy – When are the matching sets 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? The cardinality policy helps limit 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 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 policy 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.

3. THE CREDIT CARD FRAUD USE CASE

3.1 Credit Card Fraud Domain

Credit card fraud can be divided into two types: offline fraud and online fraud.

- Offline or *card is present* (CP) fraud is committed by using a stolen physical card at a call center or anywhere else.
- Online fraud or *card is not present* (CNP) is committed via Internet, phone, shopping, web, or in the absence of a card holder.

In today's fraud detection systems, the "suspicious" transaction is marked and then manually inspected by human operators for a final verdict to disambiguate dubious cases. Therefore, the operators must be able to understand the motifs for the machine's reasoning, to make their final decision.

Another characteristic of this domain is that the dataset is very unbalanced, that is, the "not fraud" class is much more frequent (e.g., 1 to 2000 is common) than the "fraud" class.

Other important considerations include how fast the frauds can be detected (detection time/time to alarm), how many styles/types of fraud are detected, whether the detection was in online/real time (event-driven) or batch mode (time-driven). In real-time processing, transactions are analyzed as they come. Therefore, the process of tagging fraud happens before the transaction has been accepted or denied. If the transaction is marked as accepted, nothing can be done afterwards if the transaction happens to be fraudulent.

These considerations lead to a demanding environment, where speed and accuracy in the decision process are of the utmost importance.

Context is, in fact, one the most import aspects in fraud detection, as different fraud patterns will arise in different contexts. For example, in the CP scenario, two consecutive transactions with the same card made in different countries will be the first sign of fraud. However, in the CNP case, that situation happens quite often, as online purchases from merchants in different countries can easily be made in the space of a few minutes.

The full transaction flow is described in Figure 3. It starts at the cardholder, when making a purchase from a given merchant. The merchant's terminal will then send the transaction to the acquirer (acquiring bank), which relays the request to the issuer (issuing bank) through the network, also known as brand, or processor. The acquirer is the bank responsible for holding the merchants' accounts. The issuer is the bank that issued the card. The processor is the entity that serves as bridge between the acquirer and the issuer. Once the transaction is accepted by the issuer, it goes through the reverse path. Note that fraud can happen at any of the stakeholders, although it is more common at the cardholder or merchant. In other cases, a bank can be trying to avoid paying transaction fees, thereby committing fraud. We currently consider only cardholder and merchant detection, as the provided dataset does not have enough information to correctly verify the other cases.

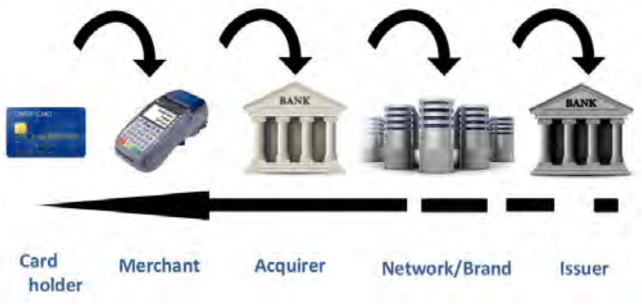


Figure 3. Fraud stakeholders

3.2 Credit Card Fraud Event Processing Network

The overarching aim of the CEP in this use case is to detect a potential fraud incident in real-time, so corrective actions can be taken. To this end, we have devised an EPN consisting of 13 EPAs (shown in Figure 4 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 derived events that are initiators of a context in other EPAs (in our EPN, EPA3 *CVVAttack* derived event initializes the contexts of EP9-EPA12; and EPA5 *SmallAmountFollowedByBigAmount* derived event initializes the temporal context of EPA13).

The two types of cases (CP and CNP) basically have the same pattern logic, except for EPA6, EPA8, and EPA12, which are only valid for the CP case, as explained in the following sections. For the common patterns, the difference resides in the length of the temporal windows, which can be either 2 or 5 minutes for the CP, and only 2 min (shorter) for the CNP case. These windows lengths have been chosen by fraud experts but other thresholds will be considered in future experiments (see Section 7).

In Figure 4 we only show the CP EPAs in the EPN, but their counterparts for the CNP case are also included in the complete EPN and have been implemented and tested. As shown, situations marked as potential (probabilistic) frauds are fired in the following cases:

- Consecutive withdrawals of increasing or decreasing amounts for a single card (EPA1 and EPA2).
- Several attempts to use a wrong CVV (Card Verification Value) for the same card are made (EPA3).
- A high number of transactions in a short time-period for a single card (EPA4).
- Small amount followed by big purchase for a single card (EPA5).
- Many large withdrawals from a single ATM (EPA6).
- Sudden card use near the expiration date (EPA7).
- Consecutive attempts to use the same card in different physical locations (EPA8).
- Combination of patterns, that is, a derived event of one pattern is the temporal context initiator of another EPA. In other words, two patterns that occur one after the other, and therefore derive a new event with higher probability of fraud than the original derived event (of the single EPA). These are EPAs 9-13, whose contexts are initiated by either EPA3 or EPA5 derived event (see Figure 4, second part). See Section 3.2.2.9 for details on these combined patterns.

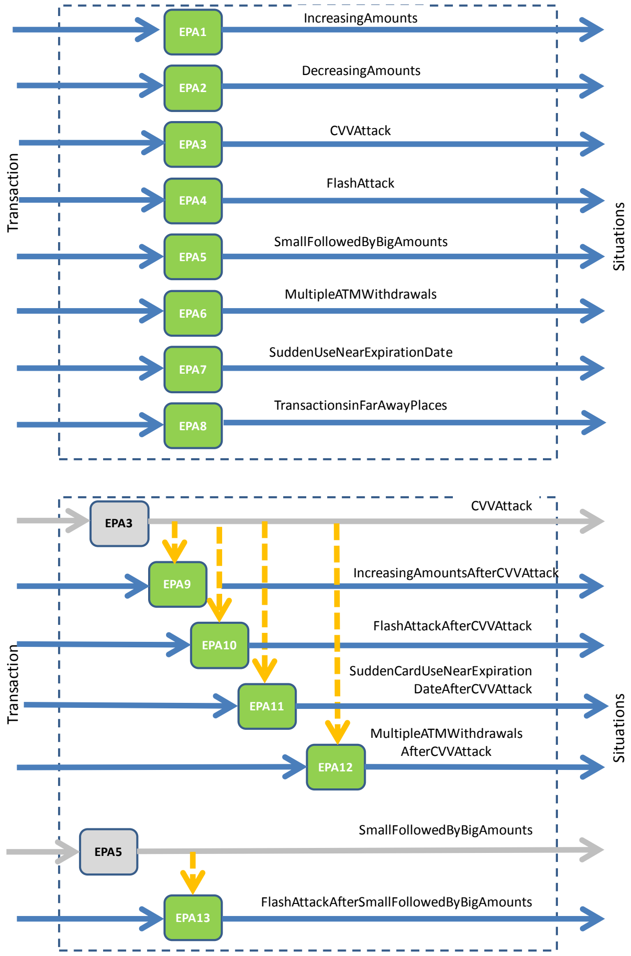

Transaction
EPA1
IncreasingAmounts
EPA2
DecreasingAmounts
EPA3
CVVAttack
EPA4
FlashAttack
EPA5
SmallFollowedByBigAmounts
EPA6
MultipleATMWithdrawals
EPA7
SuddenUseNearExpirationDate
EPA8
TransactionsinFarAwayPlaces
Situations
Transaction
EPA3
CVVAttack
EPA9
IncreasingAmountsAfterCVVAttack
EPA10
FlashAttackAfterCVVAttack
EPA11
SuddenCardUseNearExpiration DateAfterCVVAttack
EPA12
MultipleATMWithdrawals AfterCVVAttack
EPA5
SmallFollowedByBigAmounts
EPA13
FlashAttackAfterSmallFollowedByBigAmounts
Situations

Figure 4. Fraud use case initial EPN for the CP case

3.2.1 Fraud Use Case Event Types

Fourteen event types comprise the event inputs, outputs/derived, and situations, as shown in Figure 4. To simplify, we only show the user-defined attributes or the event payload (refer to 4.1.1 for metadata attributes). Also note that the *Transaction* raw event includes more fields or attributes. We present only the ones required for pattern detection in our EPN implementation. During run-time, the other attributes not specified in the event types will be ignored by the CEP engine.

We use some naming conventions for the sake of clarity. We denote event types with capital letters. Metadata attributes start with a capital letter, as well as payload attributes that hold operators values (i.e., *TrendCount* denotes the number of input events that satisfy the Trend operator, and *TransactionsCount* denotes the number of input events that satisfy the Count operator). Pattern operators and built-in functions are capitalized. Built-in and payload attributes start with a lower case letter. Table 1 shows the event definitions for the fraud EPN, where:

card_pan (type: String) - The number that identifies the card. The card BIN, which corresponds to the first six digits of the PAN, can give information such as the issuer of the card.

terminal_id (Type: Long) - The internal identification of the terminal.

cvv_validation (Type: Int) - Variable indicating whether the CVV (Card Verification Value) was used or not. In the positive case, it indicates whether it was valid (code =16) or not. This three digit number printed on the signature panel on the back of the card helps to verify authorized possession of a credit card.

amount_eur (Type: Double) - The amount in euros of the transaction.

acquirer_country (type: Int) - The country of the acquiring bank.

is_cnp (Type: Bit) - Flag that states whether the transaction happened in the CP or CNP context.

card_exp_date (type: Date (YYYYMM)) - The expiration date of the card.

Table 1. Event types for the fraud use case

Event name	Payload
Transaction	card_pan; terminal_id; cvv_validation; amount_eur; acquirer_country; is cnp; card_exp_date
IncreasingAmounts	card_pan; TrendCount; is_cnp
DecreasingAmounts	card_pan; TrendCount; is_cnp
CVVAttack	card_pan; TransactionsCount; is_cnp
FlashAttack	card_pan; TransactionsCount; is_cnp
SmallFollowedByBig Amounts	card_pan; is_cnp
MultipleATMWithdrawals	terminal_id; TransactionsCount
SuddenUseNearExpiration Date	card_pan; TransactionsCount; is_cnp; card_exp_date
TransactionsinFarAway Places	card_pan
IncreasingAmountsAfter CVVAttack	card_pan; TrendCount; is_cnp
FlashAttackAfterCVVAttack	card_pan; TransactionsCount; is_cnp
SuddenCardUseNearExpiration DateAfterCVVAttack	card_pan; TransactionsCount; is_cnp; card_exp_date
MultipleATMWithdrawals AfterCVVAttack	terminal_id; TransactionsCount
FlashAttackAfterSmall FollowedByBigAmounts	card_pan; TransactionsCount; is_cnp

3.2.2 *Fraud Use Case Event Processing Agents*

We describe the EPAs in the following order: event name; meaning; 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 out. For the *filtering step,* we show the filtering expression; for the *matching step,* we denote the pattern variables; and for the *deriving step,* we denote the value assignments and calculations. For the sake of simplicity, we show the assignments that are not copies of values; all other derived event attribute values are copied from the input events. For attributes, we only denote their names without the prefix of *'event_name.'* As aforementioned, we only show the EPAs for the CP case with the filter condition *is_cnp == 0* (the filter condition will be *is_cnp ==1* for the EPAs counterparts in the CNP case). We also check whether the transaction is valid by adding the condition cvv_validation ==16 (code number for a valid CVV) in the filter.

In our current implementation we use the *Sigmoid* probabilistic function to calculate the probability of the derived event. The Sigmoid function has been selected since it fits 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 [7]. Other parameters and functions might be applicable as well, and are one of the topics for future work. 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. The ones shown in the figures have been chosen to run the input events set. By using the Sigmoid function along with *immediate*, *unrestricted*, and *reuse* pattern policies, we get derived events with increased probabilities (higher *Certainty* values) as outputs in the same temporal window, as parameter x gets higher values with increased values in the pattern assertions. For example, when used in the Count operator, the x denotes the number of input events that satisfy the pattern assertion. This value can only increase, therefore increasing the probability of the derived event.

3.2.2.1 *EPA1: IncreasingAmounts*

EPA1 is a pattern that can be used to test the system detection limits. It checks for at least two consecutive valid transactions (i.e., trendN > 1) with increasing amounts.

Event recognition process:

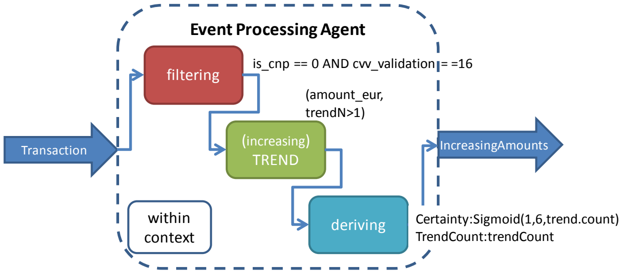


Figure 5. Event recognition process for IncreasingAmounts EPA

Pattern policies:

- Evaluation: Immediate
- Cardinality: Unrestricted
- Repeated: N/A
- Consumption: Reuse

Context:

- Segmentation: by card_pan
- Temporal (non-overlapping windows):
  - Initiator: Transaction
  - Terminator: +5 min
  - Context policy: Ignore

Meaning: A temporal window of 5 minutes opens 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 card. According to the policies selected, we will derive a new event every time the Trend pattern is satisfied (having higher probabilities values in the *Certainty* attribute). In the derived event we also report the actual number of transactions that satisfy the Trend pattern (by the built-in *trendCount* attribute).

3.2.2.2 EPA2: DecreasingAmounts

This pattern is similar to EPA1, as it aims to test system detection limits. EPA2 checks for at least two consecutive valid transactions; that is, correct CVV, with decreasing amounts that passed the card limit. The idea is that by passing the card limit, the fraudster is obliged to decrease the amount of money in the following transaction. The derived event is *DecreasingAmounts*. The contexts and policies are the same as in EPA1. For the current implementation we apply a standard average card limit in Europe.

Event recognition process:

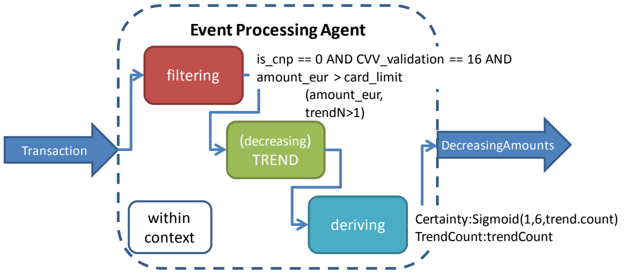


Figure 6. Event recognition process for DecreasingAmounts EPA

3.2.2.3 EPA3: CVV Attack

The fraudsters may only have access to partial information about the card. Therefore, to obtain the rest of the information, such as CVV, they can do a scan over possible CVV values. In EPA3 we look for cases in which at least four attempts with incorrect CVV are made in a very short period of time (two minutes).

Event recognition process:

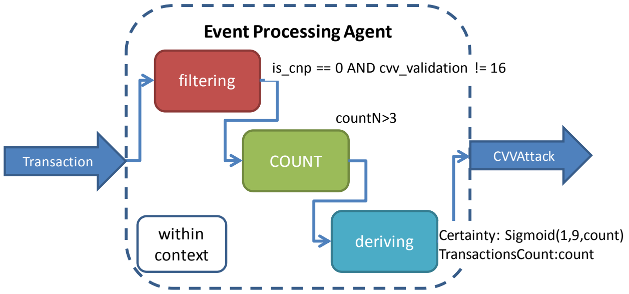


Figure 7. Event recognition process for CVV Attack EPA

Pattern policies:

- Evaluation: Deferred
- Cardinality: N/A
- Repeated: N/A
- Consumption: N/A

Context:

- Segmentation: by card_pan
- Temporal (non-overlapping windows):
  - Initiator: Transaction.cvv_validation != 16
  - Terminator: +2 min
  - Context policy: Ignore

Meaning: A temporal window of 2 minutes opens with the arrival of a first *Transaction* event with wrong CVV (cvv_validation != 16). In this elapsed time, we check for at least 4 transactions with wrong CVV. According to the policies selected, we will derive a single event at the end of the temporal window. In the derived event we also report the actual number of transactions that satisfy the Count pattern (by the built-in *count* attribute).

3.2.2.4 EPA4: FlashAttack

EPA4 is similar to EPA3, but in this case, we look for a high number of transactions in a short period of time.

Event recognition process:

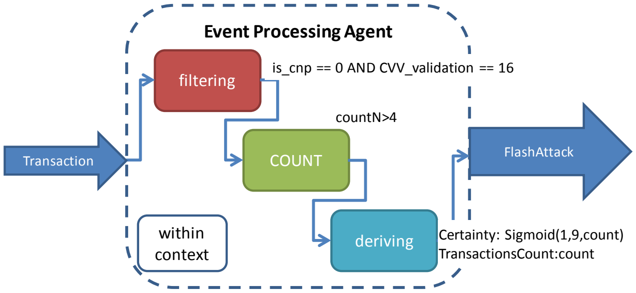


Figure 8. Event recognition process for FlashAttack EPA

Pattern policies and contexts as in EPA3

3.2.2.5 EPA5: Small Amount Followed by Big Amount

This pattern tests for system thresholds. Sometimes the fraudsters test a small amount, and once the transaction succeeds, they attempt a big purchase. We look for a *sequence* pattern where the first transaction is related to a very small amount (cents) and the second one in the sequence is related to a big amount (>200). Note that T1 and T2 are aliases of the event type *Transaction*. The *SmallAmountFollowedByBigAmount* derived event has a 0.8 probability of being a fraudulent transaction due to the nature of this pattern.

Event recognition process:

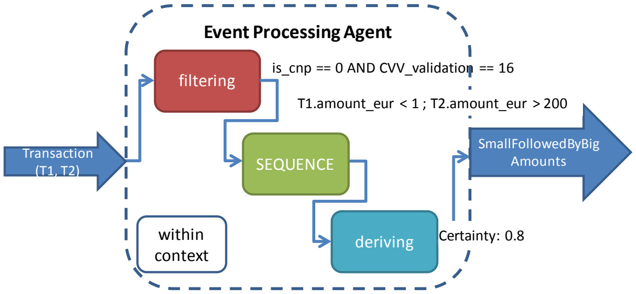


Figure 9. Event recognition process for Small Amount Followed by Big Amount EPA

Pattern policies:

- Evaluation: Immediate
- Cardinality: Single

- Repeated: T1=first; T2=override
- Consumption: N/A

Context:

- Segmentation: by card_pan
- Temporal (non-overlapping windows):
  - Initiator: Transaction.amount_eur < 1.0
  - Terminator: +2 min
  - Context policy: Ignore

Meaning: A short temporal window of 2 min opens with the arrival of a first *Transaction* event with a small amount per card. The pattern detects a small purchase, and if the consecutive transaction is a big purchase it immediately emits the situation.

3.2.2.6 EPA6: Multiple max ATM withdrawals

Given that ATMs have an upper limit for withdrawals, in this kind of attack, fraudsters are simply trying to take as much money as they can in the fewest possible transactions. A typical *large* value in an ATM in Europe is 200 euros. We look for at least three large withdrawals in a single ATM. This EPA holds only for the CP case.

Event recognition process:

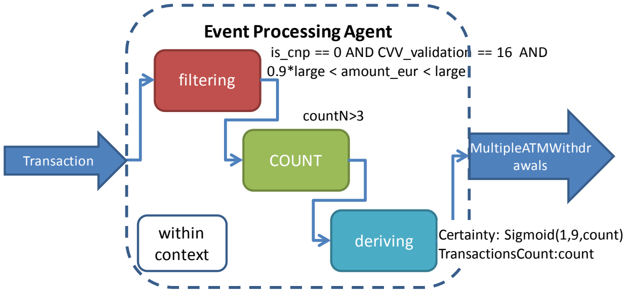


Figure 10. Event recognition process for Multiple max ATM withdrawals EPA

Pattern policies:

- Evaluation: Immediate
- Cardinality: Unrestricted
- Repeated: N/A
- Consumption: reuse

Context:

- Segmentation: by *terminal_pan*
- Temporal (non-overlapping windows):
  - Initiator: Transaction
  - Terminator: +5 min
  - Context policy: Ignore

Meaning: We derive a new event whenever the count pattern is satisfied within a five minute window.

3.2.2.7 EPA7: Sudden Card Use Near the Expiration Date

Fraudsters may obtain credit card credentials, and then sell them to other people. When the expiration date approaches, and they cannot sell those cards, they will try to make as much profit as they can from those cards before they lose control over them. We look for frequent transactions within a short period of time (2 min) for a single card on the day of, or the day before, the card expiration date.

Event recognition process:

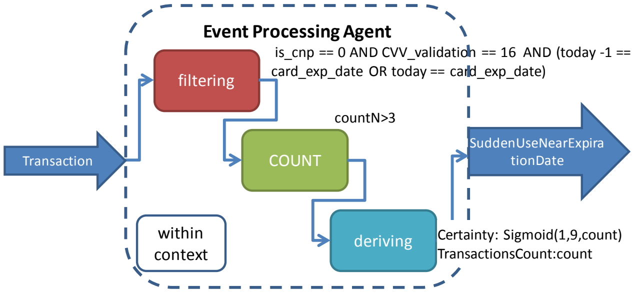

Event Processing Agent
is_cnp == 0 AND CVV_validation == 16 AND (today -1 ==
card_exp_date OR today == card_exp_date)
filtering
countN>3
Transaction
COUNT
SuddenUseNearExpirationDate
within context
deriving
Certainty: Sigmoid(1,9,count)
TransactionsCount:count

Figure 11. Event recognition process for Sudden Card Use Near the Expiration Date EPA

Pattern policies:

- Evaluation: Deferred
- Cardinality: N/A
- Repeated: N/A
- Consumption: N/A

Context:

- Segmentation: by card_pan
- Temporal (non-overlapping windows):
  - Initiator: Transaction
  - Terminator: +2 min
  - Context policy: Ignore

3.2.2.8 EPA8: Transactions in Faraway Places

Due to speed limitations in traveling, it is impossible for the same card to be used in faraway places. This means that the card has been cloned. In this pattern, we check that two consecutive transactions for a specific card cannot be at different physical places within a short period of time. Note that we do not check for CVV validity, as we are only interested in checking for use of the same card in different places. In addition, the derived event has a certainty of 1 (in this case the fraud indication is 100%) and therefore the *Certainty* attribute is not shown in the deriving step (the default is "1"). T1 and T2 are aliases of the event type *Transaction*. This EPA holds only for the CP case.

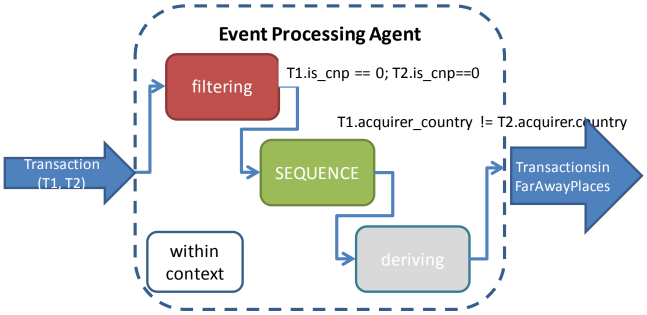

Event Processing Agent
filtering
T1.is_cnp == 0; T2.is_cnp==0
T1.acquirer_country != T2.acquirer_country
Transaction (T1, T2)
SEQUENCE
TransactionsinFarAwayPlaces
within context
deriving

Figure 12. Event recognition process for Transactions in Far Away Places EPA

Pattern policies:

- Evaluation: Immediate
- Cardinality: Unrestricted
- Repeated: T1=first; T2=override
- Consumption: Consume

Context:

- Segmentation: by card_pan
- Temporal (non-overlapping windows):
  - Initiator: Transaction
  - Terminator: +5 min

    - Context policy: Ignore

Meaning: In a 5 minute temporal window we detect and immediately alert every attempt to use the same card for any two consecutive transactions in faraway places.

3.2.2.9 EPA9-EPA13 Combined Patterns

The assumption is that when two patterns occur one after the other, the probability of a fraud is higher than in the separate EPAs. The sequencing between patterns is done by causing one derived event to open the temporal context of another EPA. The new derived event (of the latter new EPA), has the same probability of the original EPA + 0.1. In other words, the new EPA looks the same as the original EPA except for two differences: the probability of the derived event is higher by 0.1, and the context initiator is the derived event of the first EPA. Other policies remain the same.

In EPA9, the *IncreasingAmountsAfterCVVAttack* situation is derived when an Increasing Amount pattern occurs after several attempts with the wrong CVV. The context is opened with the *CVVAttack* derived event (in other words, EPA3 => EPA1).

In EPA10, the *FlashAttackAfterCVVAttack* situation is derived when a Flash Attack occurs after several attempts with the wrong CVV. The context is opened with the *CVVAttack* derived event (in other words, EPA3 => EPA4).

In EPA11, the *SuddenCardUseNearExpirationDateAfterCVVAttack* situation is derived when a Sudden Card Use Near Expiration Date pattern occurs after several attempts with the wrong CVV. The context is opened with the *CVVAttack* derived event (in other words, EPA3 => EPA7).

In EPA12, the *MultipleATMWithdrawalsAfterCVVAttack* situation is derived when a Multiple ATM withdrawals pattern occurs after several attempts with the wrong CVV. The context is opened with the *CVVAttack* derived event (in other words, EPA3 => EPA6). This pattern holds only for the CP case (as EPA6).

In EPA13, the *FlashAttackAfterSmallFollowedByBigAmounts* situation is derived when there is a Flash attack having the sequence of a big purchase after a small one. The context is opened with the *SmallFollowedByBigAmounts* derived event (in other words, EPA5 => EPA3).

4. CREDIT CARD FRAUD USE CASE IMPLEMENTATION

4.1 IBM Proactive Technology Online (PROTON) Complex Event Processing Engine

We employ the IBM Proactive Technology Online (PROTON) open source[3] research asset as our baseline engine, and extend it to cope with uncertainty capabilities as described in the following sections. For further documentation regarding the CEP open source asset refer to[4]. PROTON comprises a run-time engine, producers, and consumers with the characteristics and capabilities described in the Preliminaries section. Specifically, it includes an integrated run-time platform to develop, deploy, and maintain event-driven applications using a single programming model. PROTON has been chosen as it is our in-house CEP engine, and possesses extensible semantic and programming models.

4.1.1 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.

4.1.2 PROTON Interfaces

The PROTON standalone version run-time engine has three main interfaces with its environment as depicted in Figure 13.

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

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 are provided as configuration to the CEP run-time engine. At run-time, the 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 13).

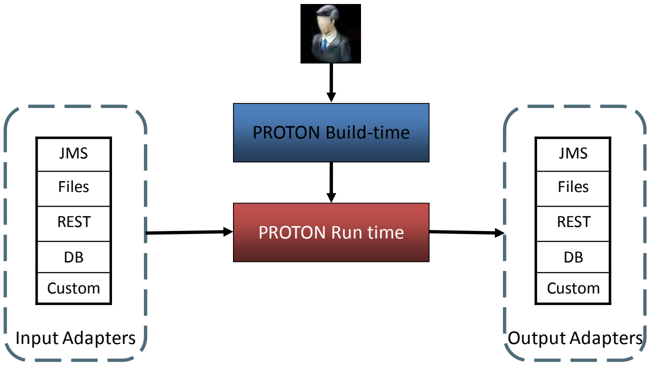


Figure 13. PROTON interfaces

4.1.3 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. The physical entities representing the logical entities of producers

[3] https://github.com/ishkin/Proton/tree/master/IBM%20Proactive%20Technology%20Online

[4] https://forge.fiware.org/plugins/mediawiki/wiki/fiware/index.php/CEP_GE_-_IBM_Proactive_Technology_Online_User_and_Programmer_Guide

and consumers in PROTON are adapter instances. For each producer, an input adapter is determined, 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 run-time.

In the trials carried out to test the use case implementation (see Section 5) we use a comma separated values (CSV) file for input and output. The input file contains real-time data anonymized for privacy. The event types are specified in Section 3.2.1.

4.1.4 PROTON Definitions

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

1. Build-time user interface – The application developer uses this to create the building blocks of the application definitions, by filling in forms, without the need to write any code. The generated file is exported in a JSON format to the CEP run-time engine.
2. Programming – The JSON definitions file can 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 includes 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 run-time engine accesses the metadata file, loads and parses all the definitions, creates a thread per each input and output adapter, starts listening for events incoming from the input adapters (producers), and forwards events to output adapters (consumers).

4.1.5 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 run-time by the **PROTON EEP** (Expandable Expression Parser)

An expression can be any combination of:

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

PROTON EEP uses any of the following operators (Table 2).

Table 2: 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 the operand types: Boolean, Datetime, Double, Integer, Numeric, String, or an array of each of these simple types.

4.2 Extending PROTON's Run-time Engine

Taking into consideration uncertainty aspects imposes fundamental extensions to PROTON's Extendable Expression Parser (EEP) as described henceforth.

4.2.1 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.

4.2.2 New Operand Types

The operand 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 whose 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.

For continuous distributions we currently support the following:

- **Exponential (mu)** – where 1/mu is the expectation
- **Gamma (n, mu)** – where n is the shape and mu is the scale
- **Log-normal (mu, sigma)** – where mu is the expectation and sigma is the standard deviation
- **Normal (mu, sigma)** – where mu is the expectation and sigma is the standard deviation
- **Triangular (lower, middle, upper)** – where lower is the left point of the triangular base, upper is right point and middle is the tip point of the triangular
- **Uniform (a, b)** – where a is the left point of the interval and b is the right point
- **Sigmoid(a,b,x)** function which returns 1/ (1 + e^(-a (x - b))) (see Section 3.2.2).

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

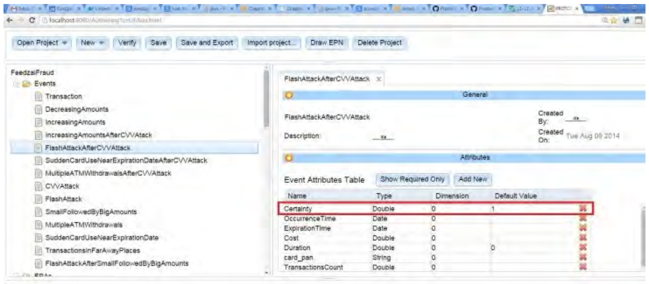
4.2.3 New Built-in Functions

- **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

To cope with these new operand types, the EPAs' operators have been adjusted to deal with uncertain or probabilistic operands and expressions.

4.3 Extending PROTON's Authoring Tool

PROTON's authoring tool forms have been accommodated to support all the extensions described above. Figure 14 shows a screenshot from PROTON's authoring tool showing our use case implementation and the addition of the *Certainty* built-in attribute.



FeedzaiFraud
Events
Transaction
DecreasingAmounts
IncreasingAmounts
IncreasingAmountsAfterCVVAtack
FlashAttackAfterCVVAttack
SuddenCardUseNearExpirationDateAfterCVVAttack
MultipleATMWithdrawalsAfterCVVAttack
CVVAttack
FlashAttack
SmallFollowedByBigAmounts
MultipleATMWithdrawals
SuddenCardUseNearExpirationDate
TransactionsInFarAwayPlaces
FlashAttackAfterSmallFollowedByBigAmounts

FlashAttackAfterCVVAttack

FlashAttackAfterCVVAttack

Description:

Created By:

Created On: Tue Aug 05 2014

Event Attributes Table

Name	Type	Dimension	Default Value
Certainty	Double	0	1
OccurrenceTime	Date	0	
ExpirationTime	Date	0	
Cost	Double	0	
Duration	Double	0	0
card_pan	String	0	
TransactionsCount	Double	0	

Figure 14. PROTON 's authoring tool showing the new Certainty attribute

5. EXPERIMENTATION AND ANALYSIS

5.1 Dataset

The dataset was provided by Feedzai[5], experts in fraud detection and prevention. Feedzai implements their own system for fraud detection, which includes real-time alerts inspected by a human operator for a final verdict. The most interesting question is whether the use of uncertain events as indicators of fraud can improve the accuracy of current systems.

The dataset contains 5600 M transactions comprising a file of 810 GB, and corresponding to nearly 3 years of transactions from 2009 to 2011. It is important to note that the data has been completely anonymized due to security and privacy issues. Of all these transactions, only around 0.05% represent fraud. Therefore, it is crucial that the system is able to detect this small fraction of fraudulent transactions.

The dataset schema includes a total of 27 fields. In our current implementation, we only require 7 fields or attributes of the *Transaction* raw event for pattern recognition (Table 1). The remaining attributes are simply ignored by PROTON's run-time engine. Among these, is the *is_fraud* (Type: Bit), which is a label indicating whether the specific transaction was marked as fraud or

[5] https://www.feedzai.com/

not (by a human operator). This attribute serves for validation testing, but again, is ignored by the CEP engine during run-time.

5.2 Implementation and Analysis

We implemented and tested our event processing network over the input dataset. Our main objective was to understand the general applicability of our event rules, specifically the inclusion of uncertainty in the credit card fraud domain using the available historical data. The preliminary results are encouraging. We were able to recognize 80% out of the fraudulent transactions with high certainty (>70%). We also detected around 0.02% additional transactions with various certainty levels.

However, there are some limitations of historical data that make it difficult for us to assess the level of precision and recall of our results. First, our detected situations (i.e., events with a high certainty of fraud) possess a wide range of uncertainty levels. Therefore, we need a human operator to disambiguate between "potential" to "real" frauds, depending on the uncertainty level of the event. We selected the level of 70% as a threshold value, but this number should be calibrated with the help of a human expert. Second, the dataset used was heavily anonymized. This fact can potentially influence the findings. Third, even with the help of a human, deducing fraudulent situations using historical data is a very complex task due to the time passed.

To overcome these limitations, we are now working on running the tests in real-time on Feedzai's premises. This way, data will be minimally anonymized while receiving the final verdict from the human operators at Feedzai. We hope in this way to get a full validation of the current EPN, with possibilities for refining the rules and parameters as part of the lessons learned.

In terms of system performance requirements and given the current state-of-the-art, the precision should be over 20% and recall over 70%. For precision, it means that out of every 100 events tagged as fraudulent, at least 20 of them are really fraud. For recall, it means that out of every 100 fraudulent transactions that pass through the system, at least 70 are caught. Again, our aim is to demonstrate that the applicability of uncertainty can improve the current metrics achieved.

The false positive rate (FPR) is correlated to the number of alerts that are being raised. As mentioned in Section 3, alerts are ultimately disambiguated by a human operator and therefore, it is important that the system does not overwhelm them with work. Too many alerts will mean that the operator will pay less attention to each of them, increasing the chances of missing true fraud. There also may be too few analysts for the number of alerts, which means some of them must be discarded.

It is important to note that the success rate should not only be measured in terms of transactions caught, but also by the value of the chargebacks. For example, for a merchant, it is certainly more important to catch a fraudulent transaction of €5000 than 100 transactions of €100.

6. RELATED WORK

In this section we survey related work in two areas: fraud detection and uncertainty in event-driven systems.

While predictive models for credit card fraud detection are in active use in practice, there are relatively few studies reported on the use of data mining approaches for credit card fraud detection, possibly due to the lack of available data for research [3]. Survey papers related to data mining and machine learning techniques can be found in [5], [16], [12], and [3]. Methods for fraud detection include: A Fusion Approach Using Dempster-Shafer Theory and Bayesian Learning, Hidden Markov Model, Neural Network, Bayesian Network, Genetic Algorithm, Artificial Immune System, K- nearest neighbor algorithm, Support Vector Machine, Decision Tree, Fuzzy Logic Based System, Random Forests, and Meta Learning Strategy. In this work, we apply complex event processing techniques for real-time detection of fraud. Future work includes the incorporation of machine learning into the design-time process of defining the event patterns.

Event-processing platforms are becoming more widely deployed within the broader financial services industry for activities such as fraud detection. The need to embed real-time intelligence into bank operations is being recognized by banks. "These capabilities are seen as critical for activities as fraud detection, including the real-time detection of patterns across multiple locations or cards" [8]. Although there are several commercial tools that have some implementations in fraud detection, we are not familiar with any tool that offers uncertainty capabilities as part of their built-in building blocks.

With relation to uncertainty in the input events, a first source of uncertainty may involve imprecise measurements on the values of attributes belonging to the event tuple. Previous works include [13], [4], [14], and [21]. In this case, an attribute is accompanied by its probability density function (pdf). Attribute uncertainty can be embodied in two ways. A first alternative is to utilize the information of the pdf of an attribute by incorporating it in the corresponding attribute's value in the event. The second, much more popular, alternative is to interpret the available information (pdf) about a continuous attribute's value to categorical attribute values, along with their respective probabilities. A second form of uncertainty may involve the event occurrence, where event e is examined as an atomic unit. In the latter case, e is represented by a tuple < e, pe > where pe denotes its occurrence probability. The previously mentioned works, with the exception of [21], assume or explicitly present (e.g., [18]) ways of mapping the uncertainty present in the attribute values of an event to an uncertainty value for the event apparition. The third type of uncertainty studied in the literature, see [4], [18], [19], and [20] involves the aspect of uncertain rules.

In the literature, handling uncertain events and their propagation through the event processing network mainly focuses on three approaches: independence of events (e.g., [4]); Markovian property adoption ([13] and [17]); and Bayesian network construction ([4], [18], [19], and [20]).

In our approach, we assume that the input events (i.e., the card transactions) are independent, and accordingly, the EPAs' operators (Trend, Count, and other aggregators) have been adjusted to deal with uncertain or probabilistic operands and expressions. This is in addition to the overloading of mathematical operations on distributions, (e.g., sum of two variables representing distribution values).

7. CONCLUSIONS AND FUTURE WORK

In the state-of-the-art, there are only a few CEP engines that support all varieties of uncertainty. The need to support uncertainty is gradually being acknowledged, and it seems that this may constitute a significant line of research and development for CEP engines [1]. In the short review of probabilistic CEP systems in [1], two limitations were identified for most of the current probabilistic engines: the absence of support for: constructing hierarchies of complex events and uncertainty in the rules defining complex events.

In this work, we present extensions carried out in open source PROTON to cope with the requirements of event processing under uncertainty, as demonstrated in a real scenario of credit card fraud.

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. 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 credit card fraud use case requirements, these have not been implemented ad-hoc, but as generic building blocks and primitives in the complex event processing programmatic language, therefore applicable to any domain and application.

Our preliminary results are encouraging, showing potential benefits that stem from incorporating uncertainty aspects to the domain of credit card fraud detection. Future refined EPNs will include additional probabilistic functions (besides the Sigmoid used in this first implementation); as well as other threshold and temporal windows values.

In our current implementation, we assume input events are independent, which simplifies the evaluation of the corresponding probabilistic formulas. For instance, the probability of a complex event detected based on an OR operator inherits the sum of probabilities of the input events as its uncertainty tag. Moreover, events derived from *Seq* or *And* (All) operator pattern matching, obtain an uncertainty value that is the product of the probabilities of input events. However, for other domains, this assumption might not hold, and dependency should be taken into account. Future work will include investigating handling dependent event occurrences.

We plan to consider learning mechanisms to automatically generate rules from historical data analysis. Defining parameters such as event patterns, thresholds, and sizes of time windows is especially complicated in most cases; users have to try several configurations until they find a setting that works well. Once founded, these parameters should be revised regularly, as fraudsters change their modus operandi from time to time. Therefore, the system should be updated with new definitions when conditions change.

8. ACKNOWLEDGMENTS

The research leading to these results has received funding from the European Union's Seventh Framework Programme FP7/2007-2013 under grant agreement 619435 (SPEEDD).

9. REFERENCES

[1] Alevizos E., Skarlatidis A., Artikis A. and Paliouras G. 2015. Complex Event Recognition under Uncertainty: A Short Survey. *Event Processing, Forecasting and Decision-Making in the Big Data Era (EPForDM),* EDBT/ICDT Workshops, 97-103.

[2] Artikis A., Etzion O., Feldman Z., and Fournier, F. 2012. Event Processing under Uncertainty. *International Conference on Distributed Event-Based Systems (DEBS12)*, 32-43.

[3] Bhattacharyya S., Tharakunnel S. J, K., and Westland J. 2011. Data mining for credit card fraud: A comparative study. Decision Support Systems, 50, 602-613.

[4] Cugola G., Margara A., Matteucci M., and Tamburrelli G. 2014. Introducing uncertainty in complex event processing: model, implementation, and validation. *Computing,* 1–42.

[5] Durga K. and Lovelin Ponn Felciah M. 2014. A Survey - Fraud Detections on Credit Cards. *International Journal of Innovative Science, Engineering & Technology (IJISET)*, 1(3), May 2014.

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

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

[8] Knox M. 2013. *Hype Cycle for Bank Operations Innovation*. Gartner report G00252360. Published: 24 July 2013.

[9] LeHong H., Fenn J., and Toit R. L-du. 2014. *Hype Cycle for Emerging Technologies.* Gartner report G00264126. Published: 28 July 2014.

[10] LeHong H. and Velosa A. 2014. *Hype Cycle for the Internet of Things*. Gartner report G00264127. Published: 21 July 2014.

[11] Linden A. 2104. *Hype Cycle for Advanced Analytics and Data Science.* Gartner report G00262076. Published: 30 July 2014.

[12] Phua C., Lee V., Smith K. and Gayler R. 2010. *A Comprehensive Survey of Data Mining-based Fraud Detection Research*. School of Business Systems, Faculty of Information Technology, Monash University, Australia.

[13] Ré C., Letchner J., Balazinksa M., and Suciu D. 2008. Event queries on correlated probabilistic streams. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data (SIGMOD08)*, 715–728.

[14] Shen Z., Kawashima H., and Kitagama H. 2008. Probabilistic event stream processing with lineage. In *Proceedings of Data Engineering Workshop (DEWS08).*

[15] Steenstrup K. 2014. *Hype Cycle for Operational Technology*. Gartner report G00263170. Published: 23 July 2014.

[16] Tripathi K.K. and Pavaskar M.A. 2012. *Survey on Credit Card Fraud Detection Methods*. International Journal of Emerging Technology and Advanced Engineering, 2(11), November 2012.

[17] Wang Y.H., Cao K., and Zhang X.M. 2013. Complex event processing over distributed probabilistic event streams. *Computers & Mathematics with Applications*, 66(10), 1808 – 1821.

[18] Wasserkrug S., Gal A., Etzion O., and Turchin Y. 2008. Complex event processing over uncertain data. In *Proceedings of the Second ACM conference on Distributed Event-Based Systems (DEBS08),* 253–264.

[19] Wasserkrug S, Gal A., Etzion O., and Turchin Y. 2012. Efficient processing of uncertain events in rule-based systems. *IEEE Transactions on Knowledge and Data Engineering*, 24(1), 45–58.

[20] Wasserkrug S., Gal A., and Etzion. 2012. O. A model for reasoning with uncertain rules in event composition systems. In *Proceedings of CoRR.*

[21] Zhang H., Diao Y., and Immerman N. 2010. Recognizing patterns in streams with imprecise timestamps. *Proc. VLDB Endowment*, 3(1-2):244–255.