IBM Proactive Technology Online User Guide

IBM Research – Haifa

Version 1.1.0: July 2015

Contents

PART I INTRODUCTION 5

Chapter 1: What is IBM Proactive Technology Online? 7

Overview 7
Highlights 9
  Functional Highlights 9
  Technical Highlights 9

Chapter 2: What is a Proton Project? 10

Overview 10
Proton Building Blocks 10
  Events 10
  Event Classes 10
  Producers 11
  Consumers 14
  Contexts 15
  Event Processing Agents 18
  Templates 22
  EPN 28
Proton Special Fields 28
  Time Format 29
  Expressions in Proton 29
Recommended Building Process 34

PART II PROTON DEVELOPMENT WEB USER INTERFACE 36

Chapter 3: Proton Development Web UI 37

Environment and Project Actions 37
  Opening a Project 38
  Creating a New Project 38
  Verifying a Project 38
  Saving a Project 38
  Exporting a Project 38
  Importing a Project 38
  Drawing the EPN 39
  Delete a Project 39
Editing Actions 40
  Creating a New Resource 40
  Opening/Editing an Existing Resource 40
  Closing a Resource 40
  Deleting a Resource 40

Appendix 41

Appendix A: Integration with the Context Broker 42

Integration with the Context Broker 42
Getting events from the Context Broker...42
Sending output events to the Context Broker 44
Live Demo Design..46

PART I

INTRODUCTION

Chapter 1: What is IBM Proactive Technology Online?

Overview

Event Detection is Not Enough

Many applications are reactive in the sense that they respond to the detection of events. These applications exist in many domains and are very useful for many applications (stock market, business opportunities, sales alerts, etc.). While the event types are known, the exact timing and content of the event instances are usually not known prior to their occurrence. Many tools in different domains have been built to detect events and to couple their detection with appropriate actions. These tools exist in products that implement active database capabilities, event management systems, "publish/subscribe" protocols, real-time systems, and similar products.

Current tools enable applications to respond to a single event. A major problem in many reactive applications is the gap between the events that are supplied by the event source, and the situations to which the clients are required to react. To bridge this gap, the client must monitor all the relevant events and apply an ad hoc decision process to decide if the conditions for reactions have been met.

From Single Events to Situation Detection

IBM Proactive Technology Online, also known as Proton, is a scalable integrated platform to support the development, deployment, and maintenance of event-driven and complex event processing (CEP) applications. While standard reactive applications are based on reactions to single events, the Proton engine component reacts to situations rather than to single events. A *situation* is a condition that is based on a series of events that have occurred within a dynamic time window called a *context*. Situations include composite events (e.g., sequence), counting operators on events (e.g., aggregation) and absence operators.

The Proton engine is a runtime tool that receives information on the occurrence of events from event producers, detects situations, and reports the detected situations to external consumers.

Examples of situations that could be reported:

- The client wishes to receive an alert if at least two of four stocks in a portfolio are up 5 percent since the beginning of the trading day.
- The client wishes to activate an automatic "buy or sell" program if, for any stock that belongs to a predefined list of stocks that are traded in two stock markets, there is a difference of more than 5 percent between the values of the same stock in different stock markets, where the time difference of the reported values is less than five minutes ("arbitrage").

In other systems, the client side software needs to store and process all the stock quotes from the different markets and decide when to issue the alert (in the first case), or when to operate the "buy or sell" program (in the second case). This may be impossible in some cases, such as for "thin" clients with low storage and processing capabilities. Even if it is possible, the solution that requires a client to process single events may result in a substantial overhead (such as ad-hoc programming efforts, increased communication traffic, or redundant storage).

Proton engine enables each client to detect customized situations without having to be aware of the occurrence of the basic events.

The major domains in which Proton has been successfully integrated include customer relationship management, policy management, multi-sensor diagnostic systems, systems management, network management, active services in wireless environments, location-based decision support systems, maintenance management, business process management, monitoring systems, service management, personalized publish/subscribe, and command and control systems.

Highlights

Functional Highlights

Proton's generic application development tool includes the following features:

- Enables fast development of CEP (complex event processing) applications, also known as Event Processing Networks (EPN).
- Resolves a major problem—the gap that exists between events reported by various channels and the reactive situations that are the cases to which the system should react. These situations are a composition of events or other situations (e.g., "when at least four events of the same type occur"), or content filtering on events (e.g., "only events that relate to IBM stocks"), or both ("when at least four purchases of more than 50,000 shares were performed on IBM stocks in a single week").
- Enables an application to detect and react to customized situations without having to be aware of the occurrence of the basic events.
- Supports various types of contexts (and combinations of them): fixed-time context, event-based context, location-based context, and even detected situation-based context. In addition, more than one context may be available and relevant for a specific event-processing agent evaluation at the same time.
- Offers easy development using web-based user interface, point-and-click editors, list selections, etc. Rules can be written by non-programmer users.
- Receives events from various external sources entailing different types of incoming and reported (outgoing) events.
- Offers a comprehensive event-processing operator set, including joining operators, absence operators, and aggregation operators.
- Includes context-based rules such as “If it is 10 minutes before trade closing time and we have more than 100 transactions to commit” or “If 4 disk failure events have occurred on the same server in the last 20 minutes”.

Technical Highlights

- Is a standard Java web application with REST APIs for administration.
- Based on a modular architecture.

Chapter 2: What is a Proton Project?

Overview

A Proton project is a set of definitions representing the incoming events and the event-processing agents that implement a certain application.

Proton's engine processes these definitions and takes actions whenever needed, reporting anything that is required to the engine consumers.

This chapter explains the different terms used in Proton, starting from the basic building blocks and ending with the more complex dependent definitions.

Proton Building Blocks

A proton project is built from the following definitions: events, producers, consumers, temporal contexts, segmentation contexts, composite contexts, and event processing agents. This definition collection defines an event processing network application.

Events

Events enter the Proton engine during runtime. They carry information about things that happen in the system domain or in the user's business domain. For example, disk failure, too many users, adding a new disk to the server, or user actions such as share purchase order.

An event is an object of an event class and its attributes are defined based on the event class. The attribute's values are always validated in respect to the attribute’s type. The events that are defined in the tool are the event classes.

Event Classes

Event classes describe the different event structures of which Proton should be aware. In a stock trading scenario, for example, this could be stockPurchase, stockSell, or tradingDayEnd.

Events are actual instances of the event classes and have specific values. For example, the event "Today, at 10pm, a customer named John Doe purchased 1000 units of IBM shares at the price of $200 each" is an instance of the "stockPurchase" event class.

Built-in Attributes

Every event instance has a set of built-in attributes in addition to its user-defined attributes. Event built-in attributes include the following:

- **Name** – the name of the event class the event is the instance of (such as "stockPurchase"). This attribute must be provided.
- **OccurrenceTime** – an attribute of type Date which the event source may assign as the occurrence time of the event. If omitted or left empty, the engine will construct the attribute and set the value to equal the DetectionTime, see below.
- **DetectionTime** – an attribute of type Date that records the time the Proton engine detects 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. The engine will construct the attribute and set its value.
- **Duration** – an attribute of type double that represents the time duration of the event in milliseconds in case the event occurs within a time interval. If omitted or left empty, the engine will construct the attribute and set its value to 0.0.
- **Certainty** – an attribute of type double that represents the certainty of this event. An event certainty can have any value between 0.0 to 1.0. If omitted or left empty, the engine will construct the attribute and set its value to 1.0.
- **Cost** – if omitted or left empty, the engine will construct the attribute and set its value to 0.0.
- **Annotation** – if omitted or left empty, the engine will construct the attribute and set its value to an empty string.
- **EventId** – a string identification of the event, which can be set by the event source. If omitted or left empty, the engine will construct the attribute and set its value to an auto-generated identifier.
- **EventSource** – holds the name of the source of the event. If omitted or left empty, the engine will construct the attribute and set its value to an empty string.

Only the built-in attribute **Name** must be provided when producing an event to the engine.

The above built-in attributes can be used in an expression in the same manner as user-defined attributes. User-defined attributes may not have the same name as any of the built-in attributes.

User-defined Attributes

You can also add your own attributes to the event class and define their types. If the attribute is an array, you must specify its dimension (array of arrays are supported).

Values to user-defined attributes must be provided if used in any expression.

Derived Event

When one of the event processing agents detects a situation, it can create one or more derived event instances. These event instances have the same characteristics as an input event; they have both user-defined attributes and built-in attributes and are needed to have event definition as any input event.

Producers

A producer introduces events from the outside world to the event-processing network. A producer definition includes the following:

- **Type** – the adapter type this producer is using to push or pull events into the EPN. The supported types are File, JMS, Rest, and custom adapter. Each adapter type has built-in parameters and other parameters can be added. Each parameter has a name and value. The adapter types and their parameters include the following:
    - **File** – using this adapter type, the producer's events would be read from a given file. A **file** producer has the following additional built-in parameters:
        - **filename** – full path file name.
    - **Timed** – timed file adapter. The events from the file will be injected not at a constant rate, but based on the relative difference of OccurenceTime attribute value of the event as specified in the event row in the file from start of injection. The first event will always be injected immediately at the start of the adapter. Its occurrence time is considered as time zero. The other events will be injected based on the relative difference between their occurrence time and the occurrence time of the first event.

      For example, if the second event's OccurenceTime (timestamp representation) is 134566788 and the first event's OccurrenceTime is 134956587 then the first event will be injected at time 0, while the second one will be injected 134956587 - 134566788 = 389799 millisecs later.

      The timed adapter has the same properties as file adapter
    - **JMS** – using this adapter type, the producer's events would be read from a JMS (Java Message Service) queue.
      Note: JMS producer is not supported in the open source version.
      A **JMS** type producer has the following additional built-in parameters:
        - Host name
        - Communication Port
        - ConnectionFactory
        - DestinationName
        - Timeout

      If a tag formatter is specified, then Proton assumes the message object is a text message and formats it with tag-delimited text formatter. If no formatter is specified, Proton assumes the content of the message is ObjectMessage and produces instances of data object implementing com.ibm.hrl.proton.adapters.jms.IObjectMessage interface
    - **Rest** –this adapter type is a REST client that GETs events from an external REST service periodically. A **Rest** type producer has the following additional built-in parameters:
        - **URL** – the fully qualified URL of the REST service for event pull operation using a GET method.
        - **ContentType** – can be "text/plain", "application/xml", or "application/json". This is defined by the REST service.
        - **PollingMode** – whether the web service returns a single instance or batch of event instances.

Note: Proton includes a REST service that provides the ability to push (notify) events to the engine, see CEP open specification document

- **Custom** – using this adapter type, the producer's event would be read using a custom mechanism defined by the user. In this case, a new type of adapter needs to be added to the adapter framework, as described in the Proton programmer's guide.

Additional parameters common to all producer types are:

- **pollingInterval** – the time to wait between two consecutive accesses to the source to pull events.
- **sendingDelay** – a delay between sending events into the EPN (mainly for demo purposes).
- **formatter** – the format of the input events (the supported formatters are **tag**, **csv**, and **json**).
- **delimiter** – the delimiter used to separate between different event attributes.
  - '**tag**' type formatter – the delimiter defines the separator between key-value pairs. Default is ";".
  - '**csv**' type formatter – the delimiter defines the separator between values. Default is ",".
- **tagDataSeparator** – for a **tag** type formatter, the separator between event attribute name and its value. Default is "=".
- **csvEventType** – for **csv** type formatter, the name of the event that is received from the producer.
- **csvAttributeNames** – for **csv** type formatter, since CSV files only list values, and not keys, of event's attributes, csvAttributeNames are used as keys. csvAttributeNames is a comma-separated string of the attributes in the order they appear in the CSV file (e.g Attribute1, Attribute2, Attribute3…). When the CSV file is read, it will link the first value to the first attribute in csvAttributeNames, and so on.
- **dateFormatter** - the default date format is dd/MM/YYYY-HH:mm:ss. If you would like to use a different format for your input events, you have to specify a date formatter (e.g., dd.MM.yyyy G 'at' HH:mm:ss z).

For custom adapters, additional required parameters can be added. Each such parameter has a name and a value.

Consumers

A consumer consumes events generated by the EPN and sends them to the outside world. A consumer definition includes the following:

- **Type** – the adapter type that is used to push or pull events from the EPN. The supported types are File, JMS, Rest, and custom adapter. Each adapter type has built-in parameters and other parameters can be added. Each parameter has a name and value. The adapter types and their parameters:
  - **File** – using this adapter type, the consumer's events would be written to a given file. A **file** consumer has the following additional built-in parameters:
    - **filename** – full path file name.
  - **JMS** – using this adapter type, the consumer's events would be written to a JMS (Java Message Service) queue.
    Note: JMS consumer is not supported in the open source version.
    A **JMS** type consumer has the following additional built-in parameters:
    - Host name
    - Communication port
    - ConnectionFactory
    - DestinationName

    If a tag formatter is specified, then Proton assumes the message object is a text message and formats it with tag-delimited text formatter. If no formatter is specified, Proton assumes the content of the message is ObjectMessage and receives instances of data object implementing com.ibm.hrl.proton.adapters.jms.IObjectMessage interface.
  - **Rest** –this adapter type is a REST client that POSTs events to an external REST service upon detection of derived events. A **Rest** type consumer has the following additional built-in parameters:
    - **URL** – the fully qualified URL of the REST service for event push operation using the POST method.
    - **ContentType** – can be "text/plain", "application/xml", or "application/json". This is defined by the REST service.
    - **AuthToken** – an optional parameter, that when set, is added as an X-Auth-Token HTTP header of the request.
  - **Custom** – using this adapter type, the consumer's events would be written using a custom mechanism defined by the user. In this case, a new type of adapter needs to be added to the adapter framework, as described in the Proton programmer's guide.

Additional parameters common to all producer types are:

- **formatter** – the format of the input events (the supported formatters are **tag**, **csv**, and **json**).
- **delimiter** – the delimiter used to separate between different event attributes.

  - ‘**tag**’ type formatter – the delimiter defines the separator between key-value pairs. Default is “;”.
  - ‘**csv**’ type formatter – the delimiter defines the separator between values. Default is “,”.
- **tagDataSeparator** – for a **tag** type formatter, the separator between event attribute name and its value. Default is “=”.
- **csvEventType** – for **csv** type formatter, the name of the event that is received from the producer.
- **csvAttributeNames** – for **csv** type formatter, since CSV files only list values, and not keys, of event’s attributes, csvAttributeNames are used as keys. csvAttributeNames is a comma-separated string of the attributes in the order they appear in the CSV file (e.g Attribute1, Attribute2, Attribute3…). When the CSV file is read, it will link the first value to the first attribute in csvAttributeNames, and so on.
- **dateFormatter** - the default date format is dd/MM/YYYY-HH:mm:ss. If you would like to use a different format for your output events, you have to specify a date formatter (e.g., dd.MM.yyyy G 'at' HH:mm:ss z).

For custom adapters, additional required parameters can be added. Each such parameter has a name and a value.

Contexts

A context determines when a particular event-processing agents are relevant. An event processing agent can have several open context instances at the same time. In such cases, an evaluation is done for each open context in parallel. There are three types of contexts:

Temporal context

A temporal context defines a time window in which the event-processing agent is relevant. It starts with an initiator and ends with a terminator.

- **Initiator** – starts the temporal context. The initiator can be an event, system startup, or absolute time.
- **Terminator** – ends the temporal context. The terminator can be an event, relative expiration time, or an absolute expiration time. A terminator definition is not mandatory; if there is no terminator defined, the lifespan never ends.

Sometimes, more than one temporal context of the same temporal context definition can be open simultaneously. If an event-processing agent uses a segmentation context or a composite context that contains a segmentation context, for every segmentation context value (for the initiator that arrives), a different temporal context opens. Furthermore, even when the temporal context does not use a segmentation context or when it has the same segmentation context value, more than one temporal context may be open simultaneously. This can happen when two initiator conditions occur before a terminator condition is fulfilled (depending on the initiator duplication policy).

A temporal context includes the following characteristics:

- Unique name.
- Type. The supported types are:

    - **Temporal Interval** – This is the regular temporal context
    - **Sliding Time Window** – A sliding window has additional two parameters:
        - Sliding Period
        - Window Duration

      In a sliding Time Window, a temporal contexts is created every Sliding Period, and each such window is active for Window Duration time. Those contexts are created as long as the temporal context is active (from its initiation till its termination)

- Initiator element and terminator element – see details below.

Temporal Context Initiator

The temporal context initiator can be one of the following:

- **Startup** – the temporal context is open at the beginning of the run or when the event processing agent is defined.
- **Event initiator** – this event acts as the initiator for this temporal context.
- **Absolute time** – this specifies the exact time that the temporal context is initiated.

A temporal context may have several initiators of several kinds.

Event Initiator Features

The event initiator has the following features:

- It is **not necessarily unique**. A temporal context can have more than one event initiator. When an event instance that is a possible initiator is detected by the Proton engine, it initiates the temporal context using the first initiator definition that the event satisfies, ordered by appearance in the temporal context definition.
- It may be **conditional**. The condition refers to the initiator event.
- It has a **correlation** policy. The **correlation** policy determines whether to open a new temporal context if another temporal context instance of this event processing agent is already open. The possible correlation values are **add** and **ignore**. **Add** – initiates a new temporal context, even if another appropriate temporal context is active. **Ignore** – does not initiate a new temporal context if another appropriate temporal context is already active.

Absolute Time Additional Features

Absolute time is the predefined time when the temporal context should be initiated. The **time** is specified in this format: ‘dd/MM/yyyy-HH:mm:ss’.

The **correlation** policy determines whether to open a new temporal context if another temporal context of this event processing agent is already open. The possible correlation values are **add** and **ignore**. **Add** initiates a new temporal context, even if another temporal context is active. **Ignore** does not initiate a new temporal context if another temporal context is already active. This is the default option.

Temporal Context Terminator

A terminator can be one of the following types:

- **Event terminator** – the event that acts as terminator for this temporal context.
- **Absolute time** – this is the exact time at which the temporal context is terminated.
- Relative time – the temporal context is terminated after a predefined interval of time that has passed from the initiation of the temporal context to its termination
- **Never ends** – the temporal context never ends and remains open until the end of the run.

A temporal context can have more than one terminator. For example, it can have several event terminators, one expiration time, and one expiration interval element. If no terminator is needed, you should choose the **Never ends** option.

Event Terminator Features

When an event instance that is a possible terminator is detected by the IBM Active Middleware Technology engine, it terminates one or more temporal context instances of the same temporal context. The terminators are activated according to their order in the temporal context definition.

Termination may be conditional. The conditions are based on the terminating event.

The first, last, or every temporal context can be terminated. This is specified by the quantifier parameter.

A temporal context can be terminated or discarded. If the temporal context is terminated, an event-processing agent can still derive events on termination. If the temporal context is discarded, the event instances that have accumulated during this temporal context are discarded, and no detection can occur for this temporal context instance. The terminator behavior is specified in the **type** parameter.

Absolute Time Features

The temporal context is terminated or discarded at a predefined time (if it is still open). The **expiration time** is specified in this format: ‘dd/MM/yyyy-HH:mm:ss’.

There are two possible expiration types: **terminate** and **discard**, which are semantically equivalent to the possible termination types of an **event terminator**.

Relative Time Features

The temporal context is terminated or discarded after a predefined amount of time passes from its initiation (if it is still open). The expiration interval is specified in milliseconds.

There are two possible expiration types: **terminate** and **discard**, which are semantically equivalent to possible termination types of **event terminator**.

Segmentation Context

A Segmentation context defines a semantic equivalent that groups events that refer to the same entity, according to a set of attributes.

For example, the **job_id** attribute in the **job_queued** event, and the **job_id** attribute in the **job_canceled** event are semantically equivalent, in the sense that they refer to the same **job** entity.

A segmentation context value can be either an attribute or an expression based on some attribute values of a certain event. The expression has to be a valid EEP expression.

Each segmentation context has a unique name and a collection of segmentation context segments. A segmentation context segment refers to the value of an expression that belongs to a specific event that participates in the segmentation context. A segmentation context segment consists of an event name and an expression.

Composite Context

A composite context groups several other contexts. An event-processing agent with a composite context may have several open context instances in parallel. A composite context instance is open if all the contexts listed in the composite context are matched (conjunction). If the composite context contains a segmentation context, this segmentation context should be defined over all the event initiators and event terminators of the temporal contexts of this composite context

Each segmentation context has a unique name and a list of contexts.

Event Processing Agents

The goal of the Proton engine is to detect predefined situations according to rules and to generate derived events.

In addition, the engine re-enters the derived events as input events. This feature enables the mechanism of nested situations.

All EPAs include most of the following general characteristics:

- Unique name
- EPA type (operator)
- Context
- Other properties such as condition
- Participating events
- Segmentation contexts
- Derived events

Other event-processing features depend on the EPA type.

Example

When three login authentication failures occur within 30 minutes, a login error situation is detected, a and login alert derived event is derived. If a segmentation context is set to **login-id**, the situation is detected only if three failures of the same login ID occur within 30 minutes.

EPA Properties

Operator Type

The operator type defined the pattern above the input events that are required for a situation detection. The EPAs are divided into the following groups according to their types:

Basic (Filter) operator - the situation is detected if the incoming event passes a threshold condition. The basic is a stateless operator, that does not correlate between its participant events.

Join operators:

- **All** – the situation is detected if all its listed participant events arrive in any order.
- **Sequence** – the situation is detected if all its listed participant events arrive in exactly the order of the operands.

Absence operator - none of the listed events have arrived during the context.

Aggregation operator – an Aggregate EPA is a transformation EPA that takes as input a collection of events and computes values by applying functions over the input events. Those computed values can used for the EPA condition and for its derived events.

Trend operator – Trend patterns are patterns that trace the value of a specific attribute over time. A Trend EPA detects increment, decrement, or stable patterns among a series of input events. For example, the rise / fall of a stock share price. The Trend operator operates only on a single event type, and detects trends among a minimum specified number of event instances (for example, an increment in value for 5 event instances in a row).

For each operator, a different sets of properties and operands are applicable.

Context

A context can be either a temporal context or composite context. It determines the time interval during which particular situations are relevant (in the above example, 30 minutes from the first login authentication failure). If the context is a composite context that contains a segmentation context, then only events that fit this segmentation context would be considered as input events for the EPA. It is not necessary for a segmentation context to have a segmentation context segment for every EPA's input event; if the event has no segmentation in the segmentation context, it will be considered as input event for the EPA. Several EPAs can be relevant during a given context (i.e., have the same context definition)

Evaluation Policy

The evaluation policy determines when the detected situation is calculated and reported. The available evaluation policies are **immediate** and **deferred**.

In the **immediate** mode, a situation is detected and reported immediately when a new event instance occurs, provided that the conditions of the situation composition are satisfied.

In the **deferred** mode, a situation is detected at the end of the context, provided that the conditions for a situation composition are satisfied. In this mode, the composition process itself is performed only when the context is terminated. This may yield different results. For example, when defining a situation that looks for the third event (aggregation with condition on a count variable), and five events occurred during the context, **immediate** detects and reports the situation as soon as the third event arrives. However, **deferred** does not detect the situation, because when the context is finished, five events remain.

In some cases, the evaluation policy is predefined by the operator. It must be **deferred** in the operator's **absence**, since the situation detection cannot be determined by the arrival of the EPA operands. The operators that use the evaluation policy attribute including: **all**, **sequence, aggregation** and **trend.**

Cardinality Policy

A situation can be detected once or multiple times in a context. The **cardinality policy** attribute is set to **unrestricted** if the situation is calculated any time its conditions are satisfied during a context, no matter how many times. However, it is set to **single** if the situation should be detected only once during its context. The operators that use the cardinality policy attribute include: **all**, **sequence, aggregation** and **trend.**

Condition

The **condition** property is the general EPA condition. A situation is reported only if it fulfills this condition. The **condition** property may refer to the different input events (operands), or the computed variables in case of aggregation EPA.

The **condition** property is applicable for the operators **all, sequence, aggregate**, and trend.

Participant Events (Operands)

The participant events are the input events to an EPA. Each participant event has its properties; some of them are the same for all EPA types, and some are relevant to specific types.

The participant events comprise following properties:

- **Event name** – name of the event.
- **Alias** – symbol of the event. When using the same event class twice as two different operands in the same situation, the alias helps us distinguish between them. Event alias is also mandatory in cases where the same event class is used both in the EPA and in its context definition.
- **Condition** – filters the events that participate in the EPA and ignores those that do not satisfy the condition.
- **Consumption** – defines the condition for events to be later reused in the same situation. This operand is applicable only for **join** operators, **aggregation** operators, and **selection** operators.

Operand Properties for Aggregation Operators

Instance Selection decides what to do when multiple events of the same operand occur. The **quantifier** is set to **First**, which selects the first event of the operand that satisfies the situation conditions, or **Last**, which selects the last event of the operand that satisfies the situation conditions.

Operand Properties for Aggregation Operands

In an aggregation operator, the user can declare computed variables. These variables are computed during runtime and can be used in the EPA condition and in the expressions assigned to derived events' attributes.

Each computed variable has the following parameters:

- **Name** – a unique (within this EPA) variable name.
- **Aggregation Type** – the type of aggregation to compute, which be set to one of the following:
    - **Count** – counts the number of participant events.
    - **Sum** – summarizes the expression value for all the participant events.
    - **Max** – maximum function over the expression value for all the participant events.
    - **Min** – minimum function over the expression value for all the participant events.
    - **Average** – average across the expression value for all the participant events.
- **Expression for every participant event** – The expression value is used to calculate the computed variable according to the aggregation type.

Operand and EPA Properties for Trend Operands

In a trend operand, the user must declare which value based on the operands' attributes will be measured for trends. For this, each operand has another attribute, **Expression**, in which the user must specify the operand's attribute, or calculated expression, which will be used.

Furthermore, the user must specify the number of events to satisfy the trend pattern. This is indicated by **Trend Count**. In addition, the user is required to specify the trend direction, under **Trend Relation**; the options for this are Increase, Decrease, and Stable.

Segmentation contexts

An EPA can have segmentation contexts and apply only to events that are semantically related. The EPA candidate events are partitioned according to the values of the attributes (or expression) defined by the segmentation context. The detection process is performed separately for each such partition.

If an EPA has a segmentation context, the segmentation context must have a segmentation context segment for every EPA operand.

An EPA segmentation context defines matching between the EPA's operands only, while the EPA general context, which can be composite context that includes segmentation contexts, defines matching between the initiators, terminators, and operands of the EPA.

The operators **basic** and **absence** do not use EPA segmentation contexts, since they do not correlate between operands.

Derived Events

When an EPA detect a situation, it can generate derived events. An EPA can generate more than one derived event. The derived event need to be defined as any other event. For each derived event, the following properties must be defined:

- **Event** – the name of the event (one of the already defined events).
- **Condition** – condition for this event derivation. When no condition is specified, the default is to derive the event.
- **Report participant** – if set to **true**, the events that participated in the situation detection would be reported when the derived event is generated.
- For each derived event's attribute, an **expression** defines how to calculate the attribute value. The expression may include attributes of events participating in this situation. For aggregation operators, the expression may include the computed variables.

Note: *In the aggregation operator, if we refer to an operand attribute, we get an array containing the values of all the instances of the same operand. There is no option to refer to a specific instance. Hence, the type of such attribute must be an array.*

Templates

To make an easier start-up with a new application, a set of pre-defined event rules has been created for the following common event patterns: FILTER, COUNT, ABSENCE and TREND. The idea is to overwrite these generic templates to speed up the creation of a new application. Templates are based on the idea that some scenarios are very common in many domains and could be easily implemented using some kind of common approach. Such scenarios include detecting an absence of event in certain time window, filtering out/in input events and creating a derived event in case the filter evaluation is true, detecting a trend of some value within a time window, and counting instances of events within a time window.
Templates are based on Proton's existing EPN event model, meaning we assume the events figuring in the template have already been defined by the time the template is created, and the template will create all the other relevant artifacts (EPA with policies, temporal, segmentation and composite contexts).

The general idea is to add a template to the EPN definition project, choose the template type, fill in the template's parameters values according to the type, and then export the definitions into JSON. The JSON created will already include all the template artifacts for the pattern, and it can be imported back to the authoring tool for additions/changes.

To create a definition using templates the following steps should be followed:

- Relevant events and their attributes should be defined. For example, if you are using a **Filter** template, the relevant input and derived event of the pattern should already be defined before using a template.

  The assumption is that the templates would just define part of the application, and all the event model of the application should already exist before using the templates.

  What events are relevant is dependent on the template: for the **Filter** template it is just the input and the output event, for other templates it is dependent on the template, please see the template description.

- In Authoring Tool click on the New menu and choose the option of creating a new template:

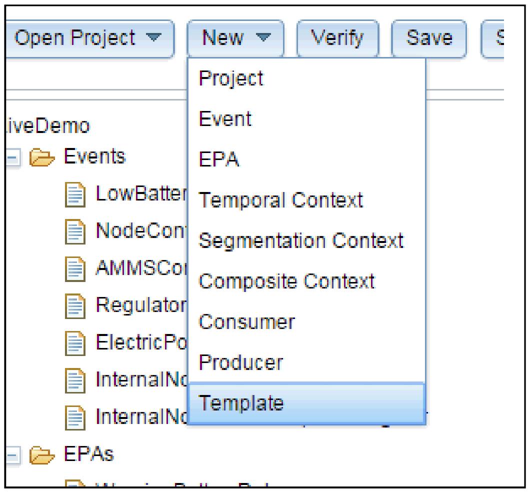


Figure 1: Authoring Tool – New Menu

This will allow specifying the name for the template. The created template will be added to projects artifacts tree:

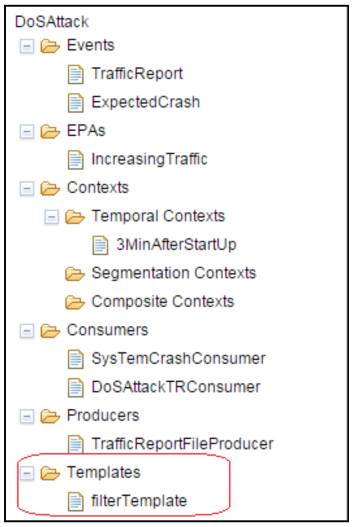
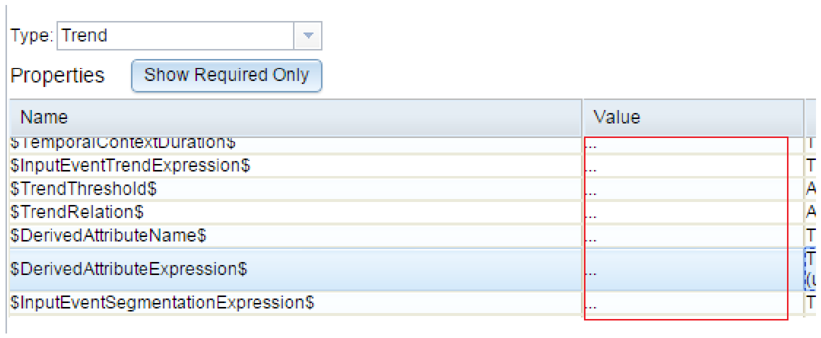


Figure 2: Authoring Tool – Created template

- After the template is created, the template type should be chosen from the drop down box. According to the chosen type, different values should be filled in into template's parameters:


Type: Trend

Properties

Name	Value	
$TemporalContextDuration$	...	T
$InputEventTrendExpression$	...	T
$TrendThreshold$	...	A
$TrendRelation$	...	A
$DerivedAttributeName$	...	T
$DerivedAttributeExpression$	...	T (
$InputEventSegmentationExpression$	...	T

Figure 3: Authoring Tool – Type selection

The following values are relevant for different templates types:

- **Filter:** the template is for filtering out events. It is based on temporal context lasting for the whole lifetime of the application (starting at startup and never ending). Filter creates derived events multiple times, every time the input event passes evaluation.
    - $InputEvent$ - the name of the input event which should be filtered out
    - $FilterExpression$ - the expression on input events attribute. Only those events for which this expression holds true will cause derivation of output event
    - $OutputEvent$ - the name of the derived event to create
    - $DerivedAttributeName$ - the name of the attribute of the derived event into which we want to assign some value, usually based on input event attributes
    - $DerivedAttributeExpression$ - The expression for the derived attribute. Can be a constant value, or based on an attributes of input event.
- **AbsenceEventInitiator:** the template is for detecting absence of event in a certain time window, started by some initiator event and lasting for N millis. The absence is detected at the end of time window. The absence is detected for a certain segmentation context, for example absence of withdrawal following deposit for certain customer would mean segmentation on customerID.
    - $InputEvent$ - the name of the input event we are monitoring the absence of
    - $OutputEvent$ - The name of the event to create if such absence is detected
    - $InitiatorEvent$ - the name of the event to start the temporal context during which we monitor for absence
    - $ContextWindowSize$ - defines the length of temporal window during which we monitor the absence, in millis, from initiator event.
    - $DerivedAttributeName$ - The name of the attribute in the derived event which is derived if absence is detected

    - $DerivedAttributeExpression$ - The expression for the derived attribute. Can be a constant value, or the partition of the segmentation context.
    - $InputEventSegmentationAttributeExpr$ - The segmentation expression based on the input event attributes for the segmentation context. (for example, if we monitor absence of Withdrawal for certain customer, then it is the Withdrawal.customerID attribute)
    - $InitiatorEventSegmentationAttributeExpr$-The segmentation expression based on the initiator event attributes for the segmentation context. (For example, if the context is initiated by a first deposit of a customer, then the initiator event is the first deposit, and the expression here is Deposit.customerID)
- **Count:** the template is for counting instances of event in a certain time window, started by some initiator event and lasting for N millis. The count is done within a certain segmentation context, for example counting withdrawals for a specific customer (segmentation by customer id). The count is at the end of time window.
    - $InputEvent$ - the name of the input event we counting
    - $OutputEvent$ - The name of the event to emit
    - $InitiatorEvent$ - the name of the event to start the temporal context during which we calculate count
    - $TemporalContextDuration$ - defines the length of temporal window during which count, in millis, from initiator event.
    - $DerivedAttributeName$ - The name of the attribute in the derived event which will hold the count value.
    - $InputEventSegmentationAttributeExpr$ - The segmentation expression based on the input event attributes for the segmentation context. (for example, if we count the number of Withdrawals for certain customer, then it is the Withdrawal.customerID attribute)
    - $InitiatorEventSegmentationAttributeExpr$-The segmentation expression based on the initiator event attributes for the segmentation context. (For example, if the context is initiated by a first deposit of a customer, then the initiator event is the first deposit, and the expression here is Deposit.customerID)
- **Trend:** the template is for searching for a trend for a certain attribute’s value of input event in a certain time window, started by some initiator event and lasting for N millis. The trend is searched for within a certain segmentation context, for example counting withdrawals for a specific customer (segmentation by customer id). The trend is reported once, at the moment the trend count reaches the specified threshold.
    - $InputEvent$ - the name of the input event we are looking for trend over attribute of/expression

    - $OutputEvent$ - The name of the event to when trend count reaches a certain threshold
    - $InitiatorEvent$ - the name of the event to start the temporal context during which we look for trend
    - $TemporalContextDuration$ - defines the length of temporal window during which we look for trend, in millis, from initiator event.
    - $InputEventTrendExpression$ - the expression over attributes of input event over which we evaluate if trend exists
    - $TrendTreshold$ - once the number of observations of trend within the value of previously specified expression reaches this threshold, output event will be derived
    - $TrendRelation$ - specifies what kind of trend we are looking for, whether 'Increase','Decrease' or 'Stable'
    - $DerivedAttributeName$ - The name of the attribute in the derived event where we want to assign value during derivation.
    - $DerivedAttributeExpression$ - The expression for the derived attribute. It can be a trend count (use 'trend.count' for that), a segmentation partition value a segmentation partition value or a constant value
    - $InputEventSegmentationAttributeExpr$ - The segmentation expression based on the input event attributes for the segmentation context. (for example, if we looking for trend in the Withdrawals for certain customer, then it is the Withdrawal.customerID attribute)
    - $InitiatorEventSegmentationAttributeExpr$-The segmentation expression based on the initiator event attributes for the segmentation context. (For example, if the context is initiated by a first deposit of a customer, then the initiator event is the first deposit, and the expression here is Deposit.customerID)

- Once the template parameters are filled in, Verify the correctness of the values you provided by pressing the "**Verify**" button
- In order to add the artifacts (EPAs and contexts) defined by the template, do **Save and Export**. This will create the JSON representing the project together with template artifacts. This JSON can be imported back into a project to see the artifacts in the project tree in order to review/change things.

Note: Take into account that template is not a constant artifact belonging to the project. This is a temporal artifact. That means it is not saved as part of the projects artifacts, and if you do **Save** or **Save and Export** what will happen is that the EPA and contex artifacts defined by the template will be added to the project, and you will see it in the tree under **EPAs** and **Contexts** headers, but the template itself is not saved as part of the project.

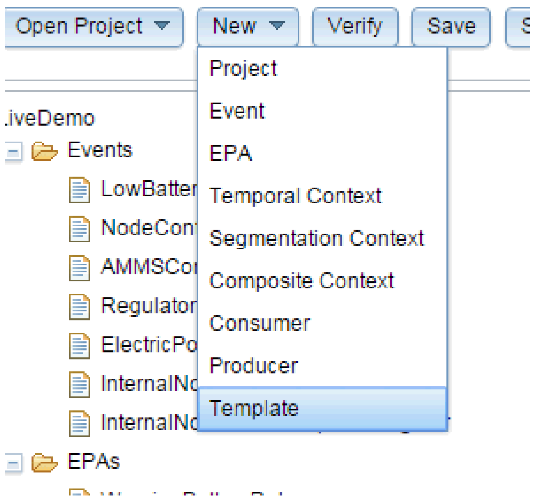
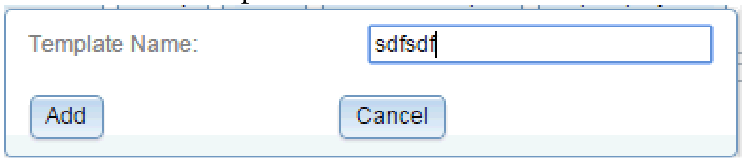


Figure 4: Authoring Tool – Template Menu

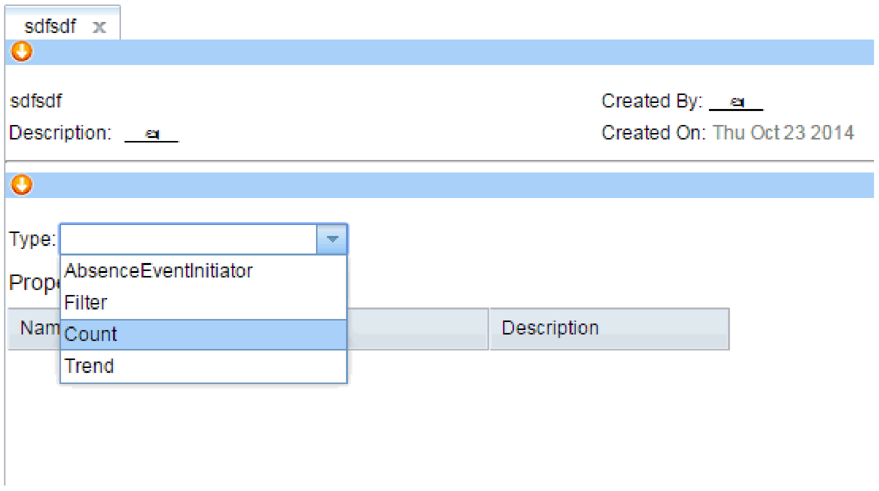
And then you choose the name of the template:



Template Name:	sdfsdf

Figure 5: Authoring Tool – Choose template name

And choose the type and fill in the values:



sdfsdf

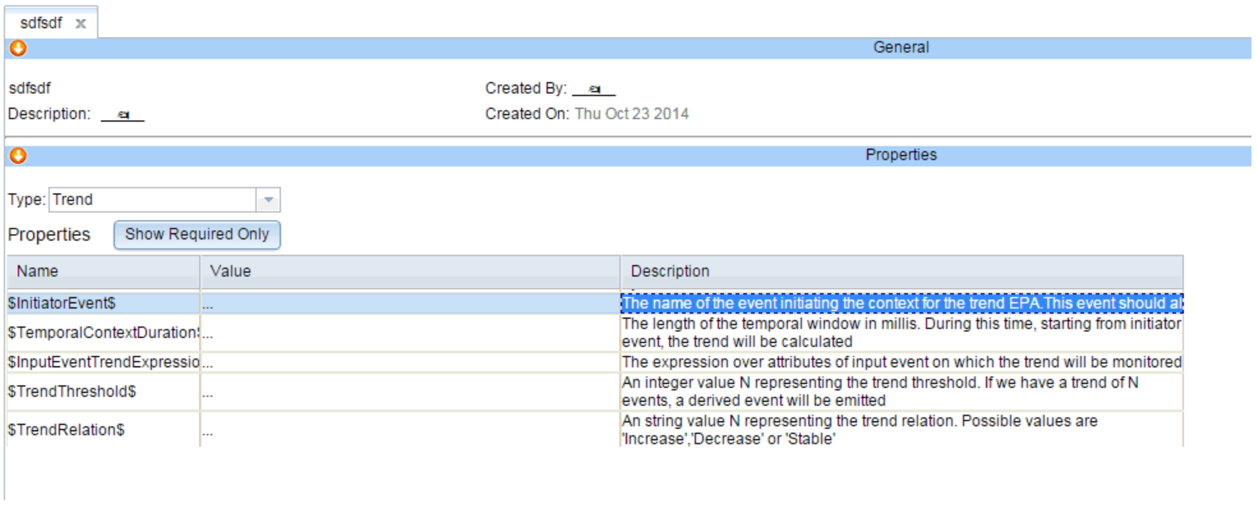
Description:

Created By:

Created On: Thu Oct 23 2014

Type:

Figure 6: Authoring Tool – Choose template type


sdfsdf

General

sdfsdf

Description:

Created By:

Created On: Thu Oct 23 2014

Properties

Type: Trend

Properties

Name	Value	Description
$InitiatorEvent$	...	The name of the event initiating the context for the trend EPA.This event should al
$TemporalContextDuration!	...	The length of the temporal window in millis. During this time, starting from initiator event, the trend will be calculated
$InputEventTrendExpressio	...	The expression over attributes of input event on which the trend will be monitored
$TrendThreshold$	...	An integer value N representing the trend threshold. If we have a trend of N events, a derived event will be emitted
$TrendRelation$	...	An string value N representing the trend relation. Possible values are 'Increase','Decrease' or 'Stable'

Figure 7: Authoring Tool – Fill template values

After that, this template is placed with as part of artifacts of the project and the tab can be opened or closed.

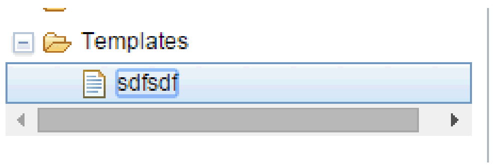


Figure 8: Authoring Tool – Save / Export the template

But the template is not saved as part of the project. So if you open the project a new without doing anything, this information would be lost.
In order to generate the artifacts from template, you do Save and Export-> and then the JSON including all the artifacts is created.

EPN

A Proton project is represented by an Event Processing Network (EPN). The EPN is a directed graph, that's nodes are the EPAs and the event classes. If an EPA epa1 generates a derived event of class e4 that is a participant event for another EPA, epa4, then in the project's EPN, EPA epa1 is connected to event class e4 by a direct edge, and e4 is connected to EPA epa4 by a direct edge. The EPN partially described here is depicted in Figure 3 The EPN makes it easier to understand the event flow in the project and to understand the hierarchy among the EPAs.

Proton Special Fields

Specialized fields within an EPN are described in this section. These are fields that

- represent time and the supported format is described
- define expressions and the expression language is described

Time Format

The default format of a date field is the standard time format of Java: ‘dd/MM/yyyy-HH:mm:ss’, where

- dd – day of the month
- MM – ordinal number of the month
- yyyy – four-digit representation of the year
- HH – hour in a 24-hour format
- mm – minutes
- ss – seconds

For example, 28/12/2003-18:30:00 means December 28, 2003, 6:30 p.m.

However, if the date arrives in different formats from the producer, a date formatter can be defined as part of the producer parameters. In the same way, a date attribute can be provided to the consumer in a different format by defining a date formatter as part of the consumer properties.

Expressions in Proton

EEP – Expandable Expression Parser

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

What is an Expression?

An expression can be a combination of the following:

- **Constant** (5, true, false, "silver", …)
- **Field** (<EventClassName>.<EventAttributeName>)
- **Built-in attribute** (DetectionTime, count, …) and built-in aggregation attributes (sum, max, …)
- **Operator** (+,-,=, …)
- **Context** (context.<segmentationContextName>, context.windowSize)
- **Built-in function** (arrayContains(a,v), distance(x1,y1,x2,y2), …)

Examples:

```
Max(DayStart.InitialStockLevel,0)
if CustomerRating="gold" then "approve" else "reject" endif
```

Operators

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

Operands

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

Context

- **Segmentation context value** –context.<segmentationContextName> returns the value of a segmentation context. It can be used in an EPA expression. Can't be used in a **basic** type EPA.
- **Temporal context duration** - context.windowSize returns the time duration of the temporal context in milliseconds. It can be used in an EPA expression. Can't be used in a **basic** type EPA.

Built-in Functions

The built-in functions can be grouped in the following categories:

Mathematical

- **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.
- **Ceil** – Ceil(x) returns the smallest integer value that is not less than x.
- **Floor** – Floor(x) returns the highest integer value that is not greater than x.

- **Crosses** – Crosses(statFunc ,Resolution, …) checks which boundary was crossed by the status function, while considering the resolution of the boundary.

Structures and Arrays

- **ArrayAppend** – ArrayAppend(a,b) appends arrays **a** and **b**.
- **ArrayIntersection** – ArrayIntersection(a,b) returns the intersection of the two arrays **a** and **b** (common elements).
- **ArraySize** – ArraySize(a) returns the size of the array **a** (the number of elements).
- **ArrayContains** – ArrayContains(a,v) returns Boolean true if the array **a** contains the value **v**; otherwise, returns Boolean false.
- **ArrayGet** – ArrayGet(a,i) returns the value of the i-th element of the array **a**.
- **ArrayHasGreaterThan** – ArrayHasGreaterThan(a,v) checks whether the array **a** contains an element with a value greater than **v**. Returns a Boolean value.
- **ArrayHasLessThan** – ArrayHasLessThan(a,v checks whether the array **a** contains an element with a value less than **v**. Returns a Boolean value.
- **ArrayIndexOf** – ArrayIndexOf(a,v) returns the index string of the location of the value **v** in the array **a**. For example, if the value is found in location [3][5][6], then the string "3.5.6" is returned; otherwise, "-1" is returned.
- **ArrayInit** – ArrayInit(Data, Type) initializes a new array where the Data string stands for array and the Type string represents for the array type.
- **ArrayMaxValue** – ArrayMaxValue(a) returns the maximal value of the elements of the array **a**.
- **ArrayMinValue** – ArrayMinValue(a) returns the minimal value of the elements of the array **a**.
- **ArraySum** – ArraySum(a) summarizes the values of the array’s elements.
- **In** – In(v,a) returns Boolean true if the array **a** contains the value **v**; otherwise, returns false.
- **SizeOf** – SizeOf(x) returns the size of the element **x**, that is an instance of a map class.

Lexical

- **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.
- **CompareToIgnoreCase** – CompareToIgnoreCase(str1,str2) compares two strings lexicographically, ignoring case differences. 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.
- **CompareToShort** – CompareToShort(str1,str2) compares a “short” string to the prefix of a “long” string lexicographically. Return value equals CompareTo.

- **CompareToShortIgnoreCase** – CompareToShortIgnoreCase(str1,str2). A combination of CompareToShort() and CompareToIgnoreCase. Compares the prefix of the “long” string with the “short” string lexicographically, ignoring case differences.
- **EndsWith –** EndsWith(str1, str2), tests whether string str1 ends with the specified suffix str2. Returns a Boolean value.
- **EqualsIgnoreCase –** EqualsIgnoreCase(str1,str2) compares the string str1 to the String str2, ignoring case considerations. Two strings are considered equal ignoring case if they are of the same length, and the corresponding characters in the two strings are equal ignoring case. Returns a Boolean value.
- **EqualsShort** – EqualsShort(str1,str2) compares a “short” string to the prefix of a “long” string lexicographically, the long string prefix size equals the short string size. Returns a Boolean value.
- **EqualsShortIgnoreCase** – EqualsShortIgnoreCase(str1,str2) a combination of EqualsShort and EqualsIgnoreCase. Compares the prefix of the “long” string with the “short” string lexicographically, ignoring case differences. Returns a Boolean value.
- **FindSubString** – FindSubString(str1,str2) returns the index within the string str1 of the first occurrence of the specified substring str2. If no such index exists, returns -1.
- **IndexOf –** IndexOf(str1,str2) returns the index within the string str1 of the first occurrence of the specified substring str2.
- **IsAlpha –** IsAlpha(str) tests whether the string str does not contain digits. Returns a Boolean value.
- **IsDigit** – IsDigit(str) tests whether the string str contains only digits. Returns a Boolean value.
- **Length** – Length(str) returns the length of string str. The length equals the number of characters in the string.
- **NthToken** – GetNthToken(str,delimiter,index) performs tokenizing of the string str using the delimiter and returns the token in the place index.
- **NumberOfDigits** – NumberOfDigits(str) returns the number of digits within str.
- **Pad –** Pad(str1,str2,num) appends str2 successively until it crosses the length of num, then cuts the last k characters, where k indicates the length of str1, then concatenates str1 to it and returns the result.
- **Replace** – Replace(str, target, replacement) replaces the target character in string str with the replacement character and returns the new string.
- **ReplaceAll -** ReplaceAll(str, regex, replacement) replaces each substring of the string str that matches the given regular expression regex with the given replacement.
- **Split –** Split(str , regex) splits the string str around matches of the given regular expression regex.
- **StartsWith –** StartsWith(str, prefix) tests whether the string str starts with the specified prefix. Returns a Boolean value.

- **StringToBoolean –**StringToBoolean(str) returns the value of the string str as Boolean.
- **StringToDouble –**StringToBoolean(str) returns the value of the string str as Double.
- **StringToInt –** StringToBoolean(str) returns the value of the string str as Int.
- **SubString** – SubString(str, beginIndex, (optional)endIndex) returns a new string which is a substring of the string str. The substring begins at the specified beginIndex and extends to the character at index endIndex—1 if it exists; otherwise, extends to the end of str.
- **ToLowerCase –** ToLowerCase(str) converts all the characters of the string str to lower case.
- **ToUpperCase –** ToLowerCase(str) converts all the characters in the string str to upper case.

Geometrical

- **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).
- **CenterOfGravity** – Center(x1,y1,x2,y2,x3,y3,...) returns the center of gravity of the listed points (x1,y1), (x2,y2), (x3,y3), etc. Returns an array with the center coordinates as its two elements.
- **InsideCircle** – InsideCircle(x,y,r,cx,cy) returns the Boolean value **true** if the point (x,y) is inside a circular area defined by the radius r and the center (cx,cy). Otherwise, returns the Boolean value **false**.
- **InsideRectangle** – InsideRectangle (x,y,ax,ay,bx,by) returns the Boolean value **true** if the point (x,y) is inside a rectangular area defined by the upper left corner (ax,ay) and the lower right corner (bx,by). Otherwise, it returns the Boolean value **false**.

Calendar

- **TimeDiff** - TimeDiff(date1 ,date2) computes the time difference between two dates. Returns the difference in milliseconds.
- **CompareDates** – CompareDates(date1,date2) compares two dates up to date precision. Returns -1 if date1 precedes date2, returns 1 if date2 precedes date1, and returns 0 if the dates are equal.
- **CompareDay** – CompareDay(date1,day) checks whether day equals the day in date1. If equal, returns 0; otherwise, returns -1.
- **WorkingDays –** WorkingDays(date1,date2) returns the number of working days between the two specified dates.
- **YearsSince –** YearsSince(date1) calculates how many years have passed from the given date1 until the current day.
- **CreateDate –** CreateDate(date1, hours1, minutes1,seconds1) creates a new date with year, month, and day of date1, and hours, minutes, and seconds equal to hours1, minutes1, and seconds1 respectively.
- **GetDate –** GetDate(date1) returns date1 at time 00:00:00 (the date without its time)

- **GetDay –** GetDay(date1) returns the day of the given date1.
- **GetMonth –** GetMonth(date1) returns the moth of the given date1.
- **GetYear –** GetYear(date1) returns the year of the given date1.
- **GetSeconds –** GetSeconds(date1) returns the number of seconds of the given date1.
- **GetMinutes –** GetMinuts(date1) returns the number of minutes of the given date1.
- **GetHours -** GetHours(date1) returns the number of hours of the given date1.

General

- **IsNull** – IsNull(val) checks whether the given val equals null. Returns a Boolean value.
- **ToDouble** – ToDouble(num) converts the given number to double.
- **ToInteger –** ToInteger(num) converts the given number to integer representation.

EEP Null Values Handling

When one or more of the operands equals null, EEP adapts the Java language standards. Each of the expression evaluations that is reported as an error in Java causes EEP to raise an exception, which results in the EPA context closing. For example, when trying to evaluate *null* + *6*, an exception is raised and the relevant EPA context is closed. However, there are several evaluations that can be performed when dealing with a null operand, such as concatenation of a null value operand to a string. Example: *"test"++null* equals "testnull" ('++' stands for concatenation) or a Boolean operator such as *true AND null* where the result is *true.*

Built-in Attributes

The complete list of built-in attributes can be found in the Event Classes section.

Recommended Building Process

The main goal of the Proton IDE (Integrated Development Environment) is to help you write correct definitions, through the use of checkboxes, property lists, and point-and-click. For example, when you define a temporal context, you select the initiator event and the terminator event from a list of existing events. When defining an EPA, you select the operand from a list of existing operands, the context from a list of previously defined contexts, and the segmentation context from previously defined segmentation contexts.

Therefore, we recommend using a bottom-up approach, starting with basic definitions, and then going to higher level definitions. This way, you have all the required building blocks in place when you start to define a new, higher level, definition.

We recommend following this definition order. Note that not all the definition types are required for all projects.

1. Event classes
2. Segmentation contexts (optional)
3. Temporal contexts

4. Composite contexts (optional)
5. EPAs
6. Producers
7. Consumers

PART II

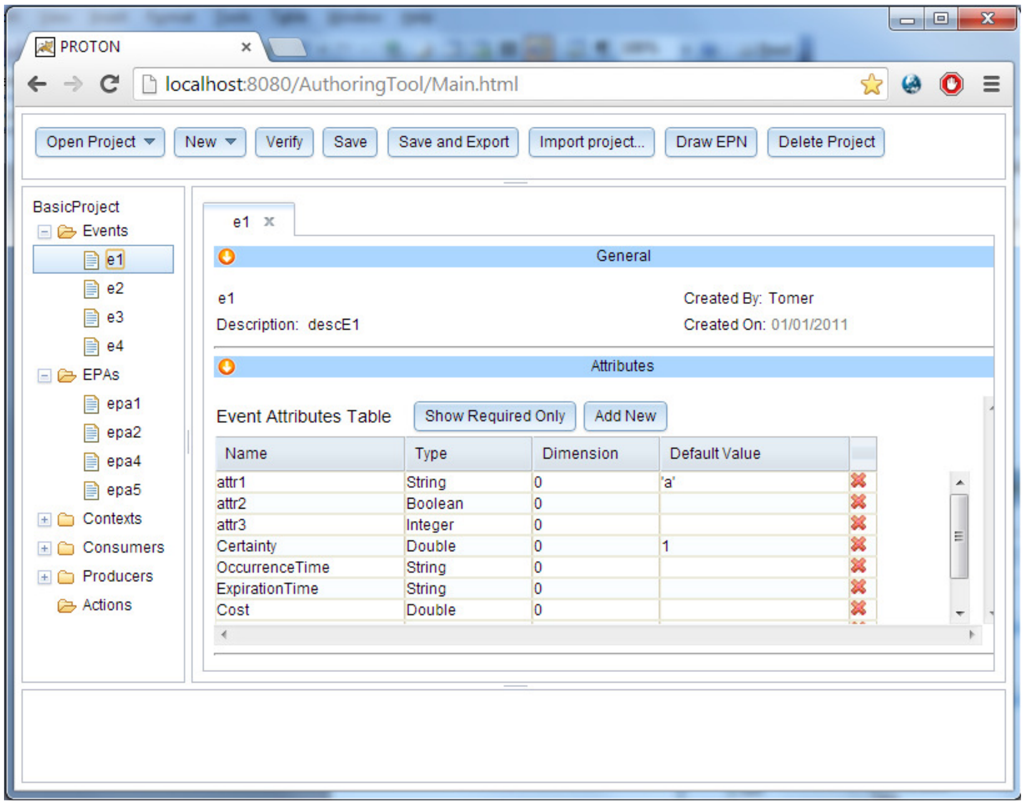
PROTON DEVELOPMENT WEB USER INTERFACE

Chapter 3: Proton Development Web UI

Environment and Project Actions

The Proton user interface is a web-based application . The Proton application is divided into the following parts:

- Buttons for generic actions (at the top).
- Resource navigator area (Explorer) (to the left).
- Editing area (in the center).
- Messages and errors area (at the bottom).



e1
Description: descE1
Created By: Tomer
Created On: 01/01/2011

Event Attributes Table

Name	Type	Dimension	Default Value
attr1	String	0	'a'
attr2	Boolean	0	
attr3	Integer	0	
Certainty	Double	0	1
OccurrenceTime	String	0	
ExpirationTime	String	0	
Cost	Double	0	

Figure 2 – Proton Authoring Tool

You will generally perform project activities using the buttons at the top, and editing activities using the editing area.

Opening a Project

To open a project, click **Open Project** and choose an existing project.

Creating a New Project

To create a new blank project:

1. Click **New**.
2. Choose **Project**.
3. Enter a project name.
4. Click **Add**.

A new project is displayed in the navigator area with the new name and all the required folders.

Verifying a Project

To verify an open project is open, click **Verify**. If the project definition has errors or warnings, they are presented in a table in the bottom of the page. You can click an error to open the relevant definition.

Saving a Project

To save the definitions of an open project, click **Save**.

The project is saved on the server. If the project has errors, you are presented with the errors so you may determine if you wish to save the project in this state.

Exporting a Project

When a project is opened, in order to export a project, click **Save and Export**. You cab either save the project locally and downloaded it according to your browser setting, or, assuming the IBM Proactive Technology Online engine is running, export the definition to the engine's definition repository. Before a project is exported it is saved on the server.

Importing a Project

To import a project that is saved in a file in the of IBM Proactive Technology Online JSON format, click **Import**.

The imported project is saved on the server and it presented in UI.

Drawing the EPN

To view the defined Event Processing Network (EPN) of an open project, click on the **Draw EPN** button.

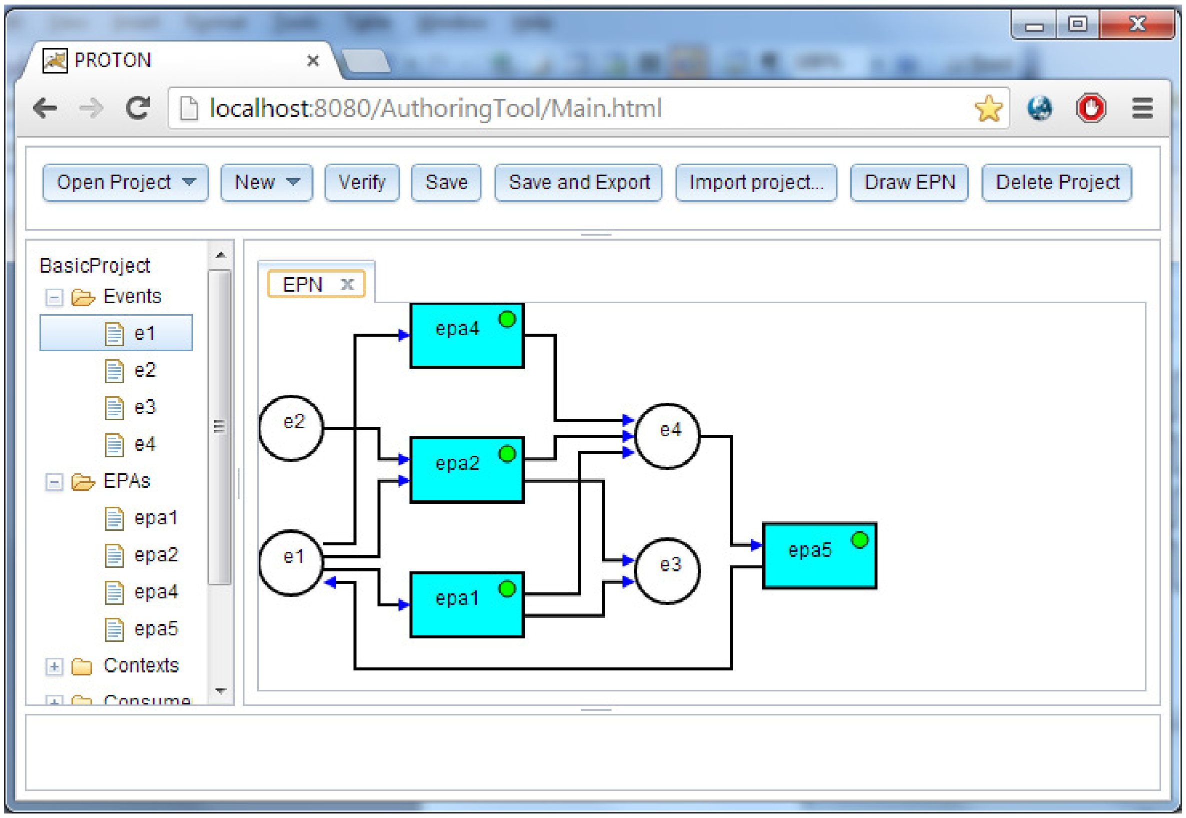


Figure 3– An EPN network

Delete a Project

To delete an open project, click **Delete**.

The project is deleted from the server.

Editing Actions

Creating a New Resource

To create a new resource:

1. Click **New**.
2. Select the type of resource you want to create and enter a name for it.
3. Click **Add**.

The editing area displays the new resource and you can edit it.

Another option is to replicate existing resource to a new one:

1. Find the existing resource in the navigator area.
2. Right click on the resource and choose **Replicate**.
3. The editing area displays the new resource. You can edit its name in the **general** section.

Opening/Editing an Existing Resource

To open an existing resource for editing:

1. Find the resource in the navigator area.
2. Double-click the resource you would like to edit. The editing area displays the resource's data.
3. Edit the resource's attributes and properties.

Closing a Resource

To close the editing area of a resource, click the **X** icon in the editor tab.

Deleting a Resource

To delete a resource:

1. Find the resource in the navigator area.
2. Right click the resource and select **Delete**.

Appendix

Appendix A: Integration with the Context Broker

Integration with the Context Broker

The integration is based on the NGSI/XML format supported by the Context Broker. There are two directions to this integration. The IBM Proactive Technology Online can get input events from the Context Broker and it can also send output events to the context broker. A specific solution can use both directions or just one of them.

Although the support of IBM Proactive Technology Online in the NGSI/XL format was designed as part of the integration with the Context Broker, any other application can use it and communicate with the IBM Proactive Technology Online in this manner.

Getting events from the Context Broker

An external application should subscribe the IBM Proactive Technology Online to changes in some entities managed by the Context Broker. This subscription should include the REST service URL of the IBM Proactive Technology Online (see the CEP open specification document). Whenever the subscription conditions are met, the Context Broker activates a POST REST of notifyContextRequest, in NGSI XML format, to the IBM Proactive Technology Online. This REST call is treated as an input event by the IBM Proactive Technology Online.

Example for such notifyContextRequest notification sent by the Context Broker is given below:

```
POST http://cep.lab.fi-ware.eu:8089/ProtonOnWebServer/rest/events
Content-Type: application/xml
Data:
<notifyContextRequest>
  <subscriptionId>51a60c7a286043f73ce9606c</subscriptionId>
  <originator>localhost</originator>
  <contextResponseList>
    <contextElementResponse>
      <contextElement>
        <entityId type="Node" isPattern="false">
          <id>OUTSMART.NODE_3505</id>
        </entityId>
        <contextAttributeList>
          <contextAttribute>
            <name>TimeInstant</name>
            <type>urn:x-ogc:def:trs:IDAS:1.0:ISO8601</type>
        <contextValue>2013-05-31T18:59:08+0300</contextValue>
      </contextAttribute>
      <contextAttribute>
        <name>presence</name>
        <type>urn:x-ogc:def:phenomenon:IDAS:1.0:presence</type>
        <contextValue></contextValue>
      </contextAttribute>
      <contextAttribute>
        <name>batteryCharge</name>
        <type>urn:x-ogc:def:phenomenon:IDAS:1.0:batteryCharge</type>
        <contextValue>2</contextValue>
      </contextAttribute>
      <contextAttribute>
        <name>illuminance</name>
        <type>urn:x-ogc:def:phenomenon:IDAS:1.0:illuminance</type>
        <contextValue></contextValue>
      </contextAttribute>
      <contextAttribute>
        <name>Latitud</name>
        <type>urn:x-ogc:def:phenomenon:IDAS:1.0:latitude</type>
        <contextValue></contextValue>
      </contextAttribute>
      <contextAttribute>
        <name>Longitud</name>
        <type>urn:x-ogc:def:phenomenon:IDAS:1.0:longitude</type>
        <contextValue></contextValue>
      </contextAttribute>
     </contextAttributeList>
    </contextElement>
    <statusCode>
     <code>200</code>
     <reasonPhrase>OK</reasonPhrase>
    </statusCode>
  </contextElementResponse>
 </contextResponseList>
</notifyContextRequest>
```

The IBM Proactive Technology Online transforms this message to an input event of type:
<entity type>ContextUpdate

In the example above, the entity type is "Node", hence the generated event is of type: NodeContextUpdate

In the IBM Proactive Technology Online application, such event type must be defined. This event must have all the context attributes defined in the subscription, and have two additional mandatory attributes:

- entityId – of type String. This attribute holds the entityId value provided in the message ("OUTSMART.NODE_3505" in the example above)

- entityType – of type String. This attributes holds the entity type provided in the message ("Node" in the example above)

Sending output events to the Context Broker

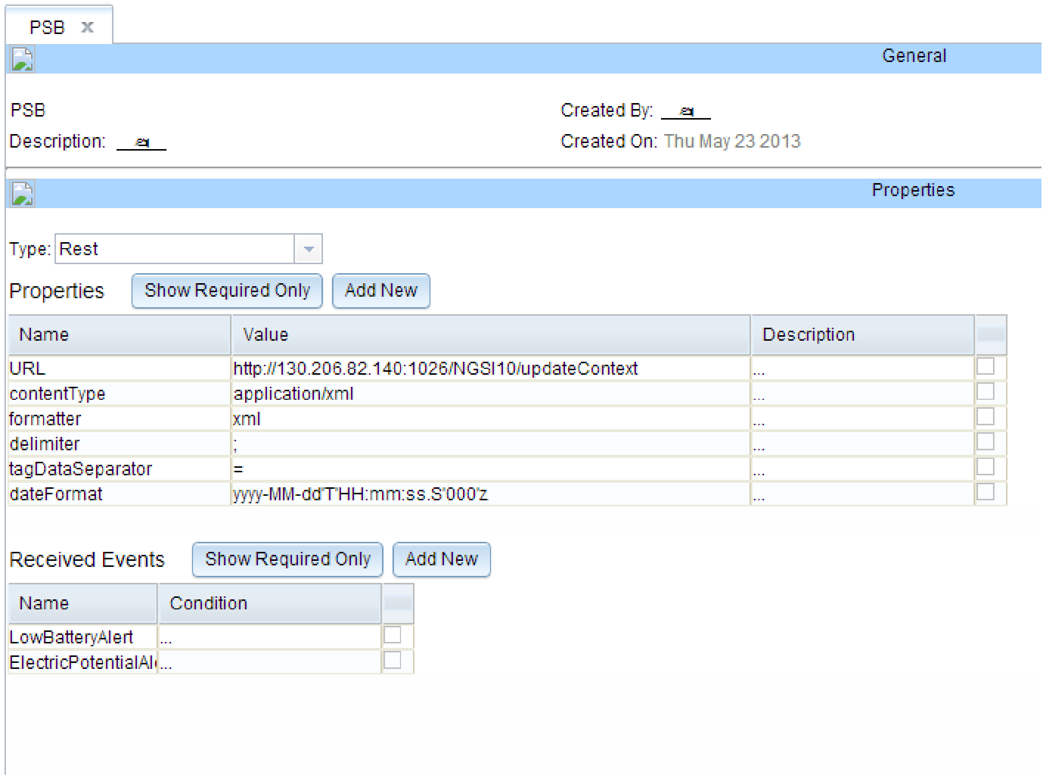
Every output event that is targeted to be sent to the Context Broker must have the following attributes:

- entityId – of type String
- entityType – of type String.

All the other attributes defined in the event should be attributes defined as context attributes in the corresponding Context Broker entity.
At runtime, the Context Broker should have a predefined entity with the entityId and entityType listed in IBM Proactive Technology Online event.
The Context Broker entity should have the IBM Proactive Technology Online built-in attributes as well (see the Built-in Attributes list).

The IBM Proactive Technology Online application should include a REST type consumer that sends the IBM Proactive Technology Online output events to the Context Broker.

An example for such consumer is given here:



PSB

Description:

Created By:

Created On: Thu May 23 2013

Type: Rest

Properties

Name	Value	Description
URL	http://130.206.82.140:1026/NGSI10/updateContext	...
contentType	application/xml	...
formatter	xml	...
delimiter	;	...
tagDataSeparator	=	...
dateFormat	yyyy-MM-dd'T'HH:mm:ss.S'000'z	...

Received Events

Name	Condition
LowBatteryAlert	...
ElectricPotentialAl	...

Note that the content type of this consumer is application/xml and its formatter is xml.

Whenever the IBM Proactive Technology Online detects event listed in such a consumer definition, the IBM Proactive Technology Online generates an updateContextRequest message and sends it via REST POST to the Context Broker.

An example for such message data is given below. Note that the entityType and entity id are given as part of the context element, while the other event attributes are given as contextAttribute elements. The IBM Proactive Technology Online filters out event attributes with empty values (since this has special meaning in the Context Broker). The IBM Proactive Technology Online doesn't send the type of the context attributes (this means that if the Context Broker entity has more than one attribute with the same name, all of those attributes are updated).

Example of output event data generated by the IBM Proactive Technology Online:

```
<updateContextRequest>
  <contextElementList>
   <contextElement>
            <entityId type="CEPEventReporter" isPattern="false">
                 <id>CEPEventReporter_Singleton</id>
            </entityId>
           <contextAttributeList>
      <contextAttribute>
       <name>EventId</name>
            <contextValue>4be0ab1c-ec30-4525-b278-78222f3ce081</contextValue>
     </contextAttribute>
      <contextAttribute>
       <name>EventType</name>
            <contextValue>LowBatteryAlert</contextValue>
     </contextAttribute>
      <contextAttribute>
       <name>DetectionTime</name>
            <contextValue>2013-06-05T08:25:15.804000CEST</contextValue>
     </contextAttribute>
      <contextAttribute>
       <name>EventSeverity</name>
            <contextValue>Critical</contextValue>
     </contextAttribute>
      <contextAttribute>
       <name>Cost</name>
            <contextValue>0.0</contextValue>
     </contextAttribute>
      <contextAttribute>
       <name>Certainty</name>
            <contextValue>1</contextValue>
     </contextAttribute>
      <contextAttribute>
       <name>Name</name>
            <contextValue>LowBatteryAlert</contextValue>
     </contextAttribute>
      <contextAttribute>
       <name>OccurrenceTime</name>
            <contextValue>2013-06-05T08:25:15.804000CEST</contextValue>
     </contextAttribute>
     <contextAttribute>
       <name>TimeInstant</name>
            <contextValue>2013-06-05T08:24:45.581000CEST</contextValue>
     </contextAttribute>
      <contextAttribute>
```

```
        <name>Duration</name>
            <contextValue>0</contextValue>
      </contextAttribute>
       <contextAttribute>
        <name>AffectedEntityType</name>
            <contextValue>Node</contextValue>
      </contextAttribute>
       <contextAttribute>
        <name>AffectedEntity</name>
            <contextValue>OUTSMART.NODE_3505</contextValue>
      </contextAttribute>
    </contextAttributeList>
   </contextElement>
  </contextElementList>
  <updateAction>UPDATE</updateAction>
</updateContextRequest>
```

Live Demo Design

As an example for application that integrates the IBM Proactive Technology Online and the Context Broker we can examine the FI-WARE live demo application.

In the live demo, the IBM Proactive Technology Online is used to detect alerts regarding the status of various entities managed by the Context Broker. Whenever a status of monitored entity is changed, the IBM Proactive Technology Online is notified. The IBM Proactive Technology Online processes those events and generates alerts when some pattern are detected. In order to manage the alerts generated by the IBM Proactive Technology Online, a singleton entity of entity type CEPEventReporter and entity id CEPEventReporter was defined in the Context Broker. This entity was updated with all the updateContextRequest events generated by the IBM Proactive Technology Online.

This CEPEventReporter entity has the attributes given in the example above. In particular, it has an attribute called EventType that holds the actual alert type detected by the IBM Proactive Technology Online and an EventSeverity attribute that holds the alert severity. In addition, this singleton entity has the attributes AffectedEntityType and AffectedEntity that allow to identify the entity that caused the alert.