



D6.3

Real World Event Driven Architecture and Events Processing First

Document Owner:	Fabiana Fournier (IBM), Inna Skarbovsky (IBM)
Contributors:	Fernando Gigante (AIDIMA)
Dissemination:	Public
Contributing to:	WP6
Date:	1/3/2016
Revision:	3

VERSION HISTORY

NBR	DATE	NOTES AND COMMENTS
0.1	02/09/2015	ToC
0.2	25/11/2015	Revision of input from use cases
0.3	10/12/2015	Input from Fernando Gigante regarding use case
0.4	30/12/2015	First complete draft
0.5	16/1/2016	Added summary and conclusions. First version available for external comments.
1	16/1/2016	Submitted version to internal review
2	26/2/2016	Commented version after review
3	1/3/2016	Final version for submission

DELIVERABLE PEER REVIEW SUMMARY

ID	Comments	Addressed (X) Answered (A)
1-4	Comments related to pattern policies	A
5	Discrepancy between the derived event name in the figure and in the text	A
6	“min” typo	A
7-8	Comments related to the temporal windows	X

Contents

Acronyms	5
1 Introduction	7
1.1 Objectives of the deliverable	7
1.1 Relationship with other documents	7
1.2 Structure of this deliverable	7
2 Analysis phase	8
2.1 Complex event processing background	8
2.1.1 Motivation	8
2.1.2 Complex event processing and the Internet of Things	9
2.1.3 Complex event processing platform in PSYMBIOSYS	10
2.2 Preliminaries	10
2.2.1 Event Processing Network (EPN)	11
2.2.2 Pattern matching process	11
2.2.3 Pattern policies	12
2.2.4 Context initiator policies	13
3 PSYMBIOSYS CEP component (standalone version)	14
3.1 Software Description	14
3.1.1 Overall Data	14
3.1.2 Purpose of the tool	14
3.1.3 Summary of Functionalities	14
3.2 Technical Information	15
3.2.1 Internal Architecture	15
3.2.2 Technological stack	18
3.2.3 Technical Manual	18
3.2.4 Licensing	19
3.3 User Manual and use-case	19
3.3.1 Description of the scenario use case	19
3.3.2 Define Event types	20
3.3.3 Event processing agents	21
3.3.4 Visualisation	27
3.4 Conclusions and Future plans	28
4 References	30

List of the figures

Figure 1: Illustration of an event processing network.....	11
Figure 2: Event recognition process in an EPA	11
Figure 3: PROTON interfaces.....	15
Figure 4: PROTON high level overview	15
Figure 5: PROTON standalone high level architecture	17
Figure 6: PROTON's components	17
Figure 7: EPN.....	20
Figure 8: SensorReading temperature raw events examples	21
Figure 9: Event recognition process for HighTemperature EPA	22
Figure 10: Context for HighTemperature EPA	23
Figure 11: Event recognition process for NoiseLevel EPA	23
Figure 12: Context for NoiseLevel EPA	24
Figure 13: Event recognition process for PressureSummary EPA	25
Figure 14: Context for PressureSummary EPA	25
Figure 15: Event recognition process for UserNotSitting5Mins EPA	25
Figure 16: Context for UserNotSitting5Mins EPA (case 1)	26
Figure 17: Context for UserNotSitting5Mins EPA (case 2)	26
Figure 18: Event recognition process for UserSittingMoreThanOneHour EPA	27
Figure 19: Context for UserSittingMoreThanOneHour EPA	27
Figure 20: Screenshot of the application dashboard	28

List of the tables

Table 1: EPN.....	20
-------------------	----

Acronyms

Acronym	Definition
CEP	Complex Event Processing
DBMSs	Data Base Management Systems
DSMSs	Data Streams Management Systems
EPA	Event Processing Agent
EPN	Event Processing Network
JSON	JavaScript Object Notation
IoT	Internet of Things
PROTON	IBM PROactive Technology ONline
PSYMBIOSYS	Product – Service sYMBIOTic SYStem

Project ID 636804	PSYMBIOSYS – Product – Service sYMBIOtic SYStem	
Date: 31/01/2016	D6.3 – Real World Event Driven Architecture and Events Processing First	

Executive Summary

Work package 6 deals with tools and platform to leverage the potential of Internet of Things for the sake of manufacturing intelligence and the integration of these tools into PSYMBIOSYS SOA-EDA platform. Within this context, this work relates to tools and platforms suitable for real time analysis of data streams for manufacturing intelligence. The goal is being able to generate, process, and analyze events produced by Internet of Things sensors in order to extract meaningful knowledge for a real time and well-founded decisional support.

In this report we accompany the delivery of the PSYMBIOSYS WP6.2 architecture and components based on the open source PROTON complex event processing tool developed and implemented by partner IBM.

Large space has been provided to the scenario demonstration (refer to Section 3.3) in order to show the capabilities of the tool and how to technically implement the demonstration on top of the software. The scenario is based on IoT for manufacturing intelligence in the context of a monitoring scenario in which factors such as noise, pressure, and temperature given by sensors at a workstation are analysed in order to design an improved environment.

Next steps in the scope of WP6 is the integration in the overall platform of WP6 in WP6.4 by M12 (D6.7). WP6 platform will be then integrated into the overall platform in WP9.2 and into end-user environments in WP10 to be used and validated by the use cases owners. The next and final version of the document is planned to be provided by M27 (D6.4).

1 Introduction

1.1 Objectives of the deliverable

D6.3 “Real World Event Driven Architecture and Events Processing First” objective is to deliver the first prototype for Real World Event Driven Architecture and Events Processing. This report is accompanying by a software prototype. The delivery is based on the IBM PROactive Technology ONline (PROTON) event processing tool that was developed by partner IBM and released as open source in the scope of the FIWARE FI-PPP project¹. Specific attention is devoted on a demonstrator based on an event driven application for monitoring environmental factors related to workstations in furniture projects and alerting in case of non-recommended situations at the workstation environment.

1.1 Relationship with other documents

D6.3 is related to D8.3 “Stream Data Analysis and Processing Platform First” that deals with the applicability to real-time analysis on Big Data as one of the key technologies is complex event processing. While both reports relate to real-time analysis of data and apply the same scenario, D6.3 focuses on the core technology and its applicability to IoT, where D8.2 proposes an architecture and a platform to support Big Data as part of PSYMBIOSYS overall platform and tools. For the sake of clarity and completeness, we bring in both reports the full requirements of the application and the background. The difference in both applications resides on the implementation side. D6.3 implements the stand alone version of the event processing engine while D8.3 implements the distributive platform adequate for Big Data applications.

This deliverable is also related to D2.2 “User Requirements Specifications and Engineering First” as we demonstrate our proposed platform through one of the project use cases.

1.2 Structure of this deliverable

This report is organized as follows:

- Section 2 provides information on the analysis phase, introduces complex event processing and describes its role in the context of manufacturing intelligence including basic terms and concepts;
- Section 3 is the core of the deliverable and provides information about the software released including: software description, technical information, and user manual. A specific effort has been devoted in chapter 3.3 to describe a comprehensive configuration and usage of the software in a realistic demonstrator.

¹ <https://www.fiware.org/>

2 Analysis phase

In this chapter, the analysis done in the context of the workpackage is reported. The analysis is the starting point for the software selection, understanding, editing, and provision in the context of the PSYMBIOSYS project.

2.1 Complex event processing background

2.1.1 Motivation

In the past decade, there has been an increase in demand to process continuously flowing data from external sources at unpredictable rate to obtain timely responses to complex queries. Traditional Data Base Management Systems (DBMSs) require data to be (persistently) stored and indexed before they could be processed, and process data only when explicitly asked by the users, that is, asynchronously with respect to their arrival. These requirements led to the development of a number of systems specifically designed to process information as a flow according to a set of pre-deployed processing rules. Two models have emerged: the data stream processing model [1] and the complex event processing model [2].

Data Stream Management Systems (DSMSs) differ from conventional Data Base Management Systems (DBMSs) in several ways: (a) as opposed to tables, streams are usually unbounded; (b) no assumption can be made on data arrival order; and (c) size and time constraints make it difficult to store and process data stream elements after their arrival, and therefore, one time processing is the typical mechanism used to deal with streams. Users of DSMS install standing (or continuous) queries, i.e., queries that are deployed once and continue to produce results until removed. Standing queries can be executed periodically or continuously, as new streams items arrive. As opposed to DBMSs, users in DSMSs do not have to explicitly ask for updated information; rather the system actively notifies it according to installed queries. DSMSs focus on producing queries results, which are continuously updated in accordance to the constantly changing contents of their input data. Detection and notification of complex patterns of elements involving sequences and ordering relations are usually out of the scope of DSMSs. DSMSs mainly focus on flowing data and data transformation, but only a few allow the easy capture of sequences of data involving complex ordering relationships, not to mention taking into account the possibility to perform filtering, correlation, and aggregation of data directly in-network, as streams flow from sources to sinks.

Complex Event Processing (CEP) systems adopt an opposite approach. They associate a precise semantics to the information items being processed: they are notifications of events which happened in the external world and were observed by sources, also called event producers. The CEP engine is responsible for filtering and combining such notifications to understand what is happening in terms of higher-level events (a.k.a. complex events, composite events, or situations) to be notified to sinks, called event consumers. CEP systems put emphasis on the issue that represents the main limitation of DSMSs, that is, the ability to detect complex patterns of incoming items, involving sequencing and ordering relationships. An example for a situation is a *Suspicious account* which is detected whenever there are at least three large cash deposits within 10 days for the same account. Event processing is in essence a paradigm of reactive computing: a system observes the world and reacts to events as they occur. It is an evolutionary step from the paradigm of responsive computing, in which a system responds only to explicit service requests. Event processing has evolved in the past years departing from traditional computing architectures which employ synchronous, request-response interactions between client and servers to reactive application, in which decisions are driven by events.

CEP [3] is a technique in which incoming data about what is happening (event data) is processed more or less as it arrives to generate higher-level, more-useful, summary information (complex events). Event processing platforms have built-in capabilities for filtering incoming data, storing windows of event data, computing aggregates and detecting patterns. In a more formal terminology, CEP software is any computer program that can generate, read, discard and perform calculations on events. A complex event is an abstraction of one or more raw or input events. Complex events may signify threats or opportunities that require a response from the business. One complex event may be the result of calculations performed on a few or on millions of events from one or more event sources. A situation may be triggered by the observation of a single raw event, but is more typically obtained by detecting a pattern over the flow of events. Event processing deals with these functions: get events from sources (event producers), route these events, filter them, normalize or otherwise transform them, aggregate them, detect patterns over multiple events, and transfer them as alerts to a human or as a trigger to an autonomous adaptation system (event consumers). An application or a complete definition set made up of these functions is also known as an Event Processing Network (EPN).

As aforementioned, the goal of a CEP engine is to notify its users immediately upon the detection of a pattern of interest. Data flows are seen as streams of events, some of which may be irrelevant for the user's purposes. Therefore, the main focus is on the efficient filtering out of irrelevant data and processing of the relevant ones. Obviously, for such systems to be acceptable, they have to satisfy certain efficiency, fault tolerance, and accuracy constraints, such as low latency and robustness.

CEP platforms required a new type of architecture. Conventional architectures are not fast or efficient enough for some applications because they use a "save-and-process" paradigm in which incoming data is stored in databases in memory or on disk, and then queries are applied. When fast responses are critical, or the volume of incoming information is very high, application architects instead use a "process-first" CEP paradigm, in which logic is applied continuously and immediately to the "data in motion" as it arrives. CEP is more efficient because it computes incrementally, in contrast to conventional architectures that reprocess large datasets, often repeating the same retrievals and calculations as each new query is submitted.

2.1.2 Complex event processing and the Internet of Things

Internet of Things (IoT) has become pervasive in our daily life. In fact, according to Gartner "The Internet of Things has the potential to transform industries and the way we live and work." [9].

A recent report by McKinsey Global Institute states that IoT has a total potential economic impact of \$3.9 trillion to \$11.1 trillion a year by 2025 [6]. Gartner forecasts that by 2020 there will be 25 billion connected IoT devices with a compound annual growth rate of 35% [7]. This tremendous growth is reflected in every industry and sector. One of these industries is manufacturing.

As pointed out by Gartner analyst Roy Schulte, event processing is the primary method to extract information value from event streams². One of our main goals in the project is to demonstrate how we can apply CEP to the scenarios in the project and leverage the power of streaming data from IoT sensors.

² <http://www.complexevents.com/2012/07/25/does-anyone-care-about-event-processing/>

There are already examples, reported in the literature, of event-driven applications in the manufacturing intelligence arena that apply CEP. David White in his post from April 20, 2015³ brings examples of companies that are using complex event processing for continuous process control. “One example is related to manufacturing intelligence and the Internet of Things (IoT). Schwering & Hasse, a company based in Germany that runs 400 production lines round the clock to manufacture enough copper magnet wire every day to wrap around the earth’s circumference three and a half times. Copper magnet wire, coated with a thin yet precise layer of insulation, is a vital component in many electrical products, such as transformers and motors. Made to fine tolerances, the wire is embedded within other components, so failure, product recalls, etc., are very expensive for their customers to manage. Quality is everything - if quality is poor, they can get dropped as a supplier. This demand for high quality means monitoring the production process in real-time, about 20,000 measurements per second across the factory. There are roughly 20 different feeds - oven temperatures, speed of the feed, rpm of the cooler, air monitoring - and so on. In addition, the quality of the insulation is checked once every inch. All of this data is fed into a complex event processing engine to provide real-time process control and alerting. This ensures that production quality remains high and has changed the way the factory works. Without such rigorous real-time production monitoring, there was a possibility that a whole spool of wire might need to be scraped if it didn't meet the quality standards. Continuous process control helps to reduce scrap, shipments are on-time, and service level agreements kept”.

This is only one example, but it strengthens the big potential of applying complex event processing technology for manufacturing intelligence in the light of IoT. As Gartner points out “CEP is also essential to future Internet of Things applications where streams of sensor data must be processed in real time” [5]. Furthermore “much of the growth in CEP usage during the next 10 years will come from the Internet of Things, digital business and customer experience management applications” [8].

2.1.3 Complex event processing platform in PSYMBIOSYS

In the PSYMBIOSYS project the complex event processing component is built on and extends the IBM PROactive Technology ONline (PROTON) research asset. This asset has become open source in the scope of the FIWARE FI-PPP project⁴ (PROTON being the CEP Generic Enabler in the FIWARE/FICORE platform). Open source code as well as technical documentation regarding PROTON can be found in⁵. PROTON comprises an authoring tool, a run-time engine, and producers and consumers adapters. Specifically, it includes an integrated run-time platform to develop, deploy, and maintain event-driven applications using a single programming model. In real-time PROTON executes a JSON (JavaScript Object Notation) definition file that includes all definitions for a specific event-driven application. The JSON file essentially represents the event processing network for a specific application (see Section 2.2).

2.2 Preliminaries

Each complex event processing engine uses its own terminology and semantics. We follow the semantics presented in Etzion’s and Niblet’s book [4] and applied in PROTON.

³ <http://www.arcweb.com/Lists/Posts/Post.aspx?List=28fb754f-5e14-4e59-8ca4-e1df494fe098&ID=456&Web=a946f105-faeb-4d28-b3a9-cc264012f95e>

⁴ <https://www.fiware.org/>

⁵ <https://github.com/ishkin/Proton/>

Henceforth we briefly present main concepts and building blocks in our terminology. The terms definitions are copied from [4] and brought here for the sake of clarity. For further details refer to [4].

2.2.1 Event Processing Network (EPN)

An **Event Processing Network (EPN)** is a conceptual model, describing the event processing flow execution. An EPN comprises a collection of event processing agents (EPAs), event producers, events, and event consumers (Figure 1). The network describes the flow of events originating at event producers and flowing through various event processing agents to eventually reach event consumers. For example, in Figure 1, events from Producer 1 are processed by EPA 1. Events derived by EPA 1 are of interest to Consumer 1 but are also processed by EPA 3 together with events derived from EPA 2.

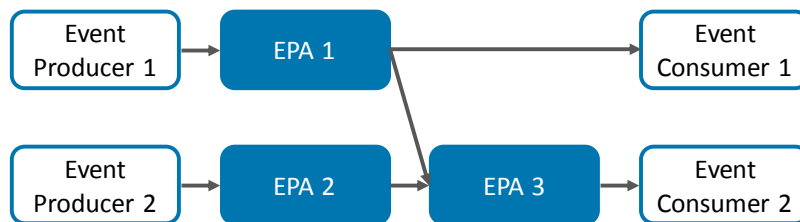


Figure 1: Illustration of an event processing network

2.2.2 Pattern matching process

An EPA performs three logical steps, a.k.a. **pattern matching process** or **event recognition** (see Figure 2).

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

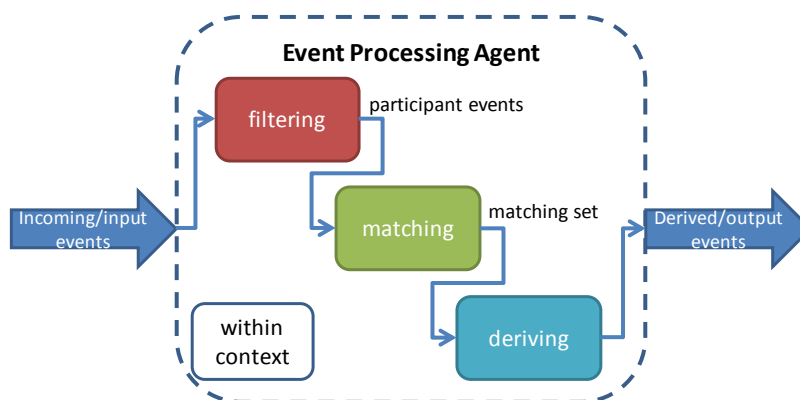


Figure 2: Event recognition process in an EPA

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

- **Sequence**, means that at least one instance of all participating event types must arrive in a specified order for the pattern to be matched.
- **Count**, means that the number of instances in the participant event set satisfies the pattern's number assertion.
- **All**, means that at least one instance of all participating event types must arrive for the pattern to be matched; the arrival order in this case is immaterial.
- **Trend**, events need to satisfy a specific change (increasing or decreasing) over time of some observed value; this refers to the value of a specific attribute or attributes.
- **Sum**, means that the value of a specific attribute, summed up over all participant events, satisfies the sum threshold assertion.
- **Average (AVG)**, means that the value of a specific attribute, averaged over all participant events, satisfies the average threshold assertion.

Note that the first two steps are optional but a derivation must take place (even if it is merely copying values from the input events to the derived/output event).

2.2.3 Pattern policies

A *pattern policy* (or simply *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. We distinguish among four types of pattern policies:

- **Evaluation policy** – This policy relates to the point(s) in time the matching sets are produced. The EPA can either generate output incrementally (in this case the evaluation policy is called *Immediate*) or at the end of the temporal context (called *Deferred*).
- **Cardinality policy** – This policy relates to the number of matching sets produced. Cardinality policy helps limiting the number of matching sets generated, and thus the number of derived events produced. The policy type can be *single*, meaning only one matching set is generated (for the entire temporal window); or *unrestricted*, meaning there are no restrictions on the number of matching sets generated.
- **Repeated/Instance Selection type policy** – what happens if the matching step encounters multiple events of the same type? The *override* repeated policy means that whenever a new event instance is encountered and the participant set already contains the required number of instances of that type, the new instance replaces the oldest previous instance of that type. The *every* repeated policy means that every instance is kept, meaning all possible matching sets can be produced. *First* means that every instance is kept, but only the earliest instance

of each type is used for matching. *Last* is the same as first, but the latest instance of each type is used for matching. This policy applies for the SEQUENCE and ALL patterns only.

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

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

2.2.4 Context initiator policies

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

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

3 PSYMBIOSYS CEP component (standalone version)

This chapter focuses on the description of the software component released. The section starts summarising the overall information about the software released (description, overall data, functionalities and architecture), after that technical information are reported about architectural stack, technical manual for installation and licensing (including third parties components). Finally the user manual and the conclusions and future steps closes the chapter.

3.1 Software Description

3.1.1 Overall Data

Item	Value
Component Name	IBM PROactive Technology ONLINE (PROTON)
Software version	Version 4.4.1
Reference workpackage	WP6.2 and WP8.2
Responsible Partner	IBM
Contact person	Fabiana Fournier (fabiana@il.ibm.com)
Source control	https://github.com/ishkin/Proton
Short Description	PROTON is an integrated platform to support the development, deployment, and maintenance of event-driven and complex event processing (CEP) applications.

3.1.2 Purpose of the tool

The purpose of the tool is to enable the design and deployment of event driven applications, using a friendly programming model. In the context of PSYMBIOSYS, the CEP component enables the development and deployment of manufacturing intelligence event driven applications for some use cases scenarios in the project. The CEP component enables the analysis and processing of input events in real-time and the detection of situations of interest for operational decision making, by using a set of building blocks and a declarative language.

3.1.3 Summary of Functionalities

PROTON supports the following features:

- Several 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.
- Easy development using web-based user interface, point-and-click editors, and list selections. PROTON uses a declarative language intended for non-programmer users.
- A comprehensive event-processing operator set, including joining operators, absence operators, and aggregation operators.

Technical highlights:

- Is platform-independent, uses Java throughout the system.
- Comes as a J2EE (Java to Enterprise Edition) application or as a J2SE (Java to Standard Edition) application.
- Based on a modular architecture.

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

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

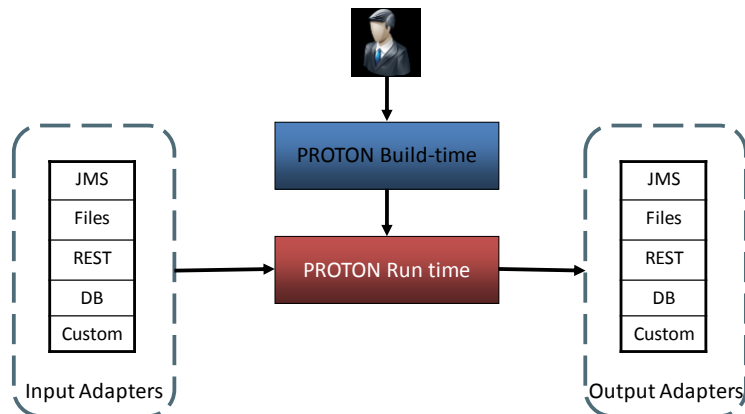


Figure 3: PROTON interfaces

The application definitions, i.e. the EPN, are written by the application developer during the build-time. The definitions output in JSON (JavaScript Object Notation) format is provided as configuration to the CEP run-time engine. At run-time, the standalone CEP engine receives incoming events through the input adapters, processes these incoming events according to the definitions, and sends derived events through the output adapters (see Figure 3). Figure 4 shows PROTON high level overview including the authoring tool and the run time engine.

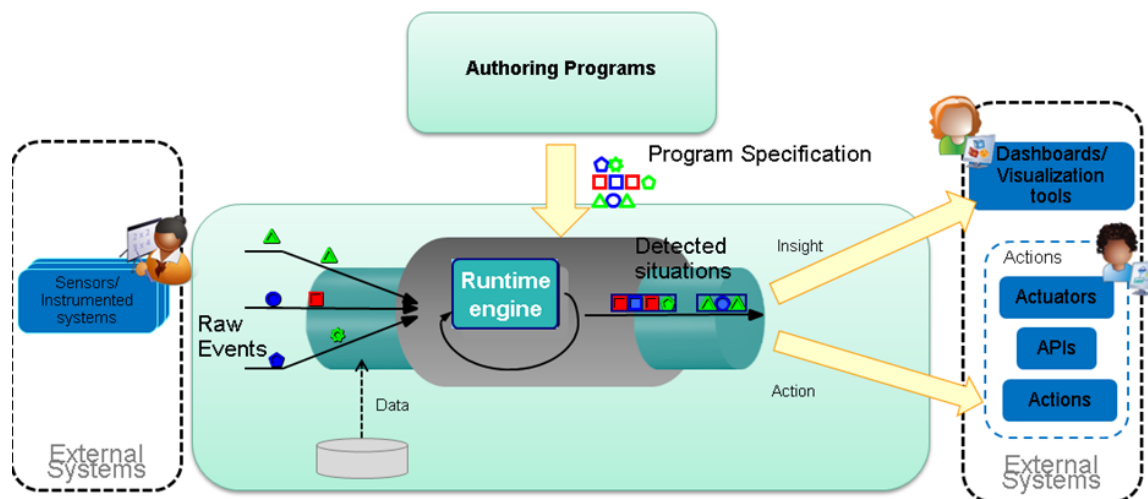


Figure 4: PROTON high level overview

3.2 Technical Information

3.2.1 Internal Architecture

PROTON architecture consists of a number of functional components and interaction among them, the main of which are (see Figure 5):

- Adapters – communication of PROTON with external systems
- Parallelizing agent-context queues – for parallelization of processing of single event instance, participating in multiple patterns/context, and parallelization of processing among multiple event instances
- Context service – for managing of context’s lifecycle –initiation of new context partitions, termination of partitions based on events/timers, segmenting incoming events into context groups which should be processed together.
- EPA manager –for managing EPA instances per context partition, managing its state, pattern matching and event derivation based on that state.

When receiving a raw event, the following actions are performed:

1. Look up within the **metadata**, to see which context effect this event might have (context initiator, context terminator) and which pattern this event might be a participant of
2. If the event can be processed in parallel within multiple contexts/patterns (based on the EPN definitions), the event is passed to **parallelization queues**. The purpose of the queues:
 - a. Parallelize processing of the same event by multiple unrelated patterns/contexts at the same time keeping the order for events of the same context/pattern where order is important
 - b. Solve out-of-order problems – can buffer for a specified amount of time
3. The event is passed to **context service**, where it is determined:
 - a. If the context is an initiator or a terminator, new contexts might be initiated and or terminated, according to relevant policies.
 - b. Which context partition/partitions this event should be grouped under
4. The event is passed to **EPA manager**:
 - a. Where it is passed to the specific EPA instance for the relevant context partition,
 - b. Added to state of the instance
 - c. And invokes pattern processing
 - d. If relevant, a derived event is created and emitted

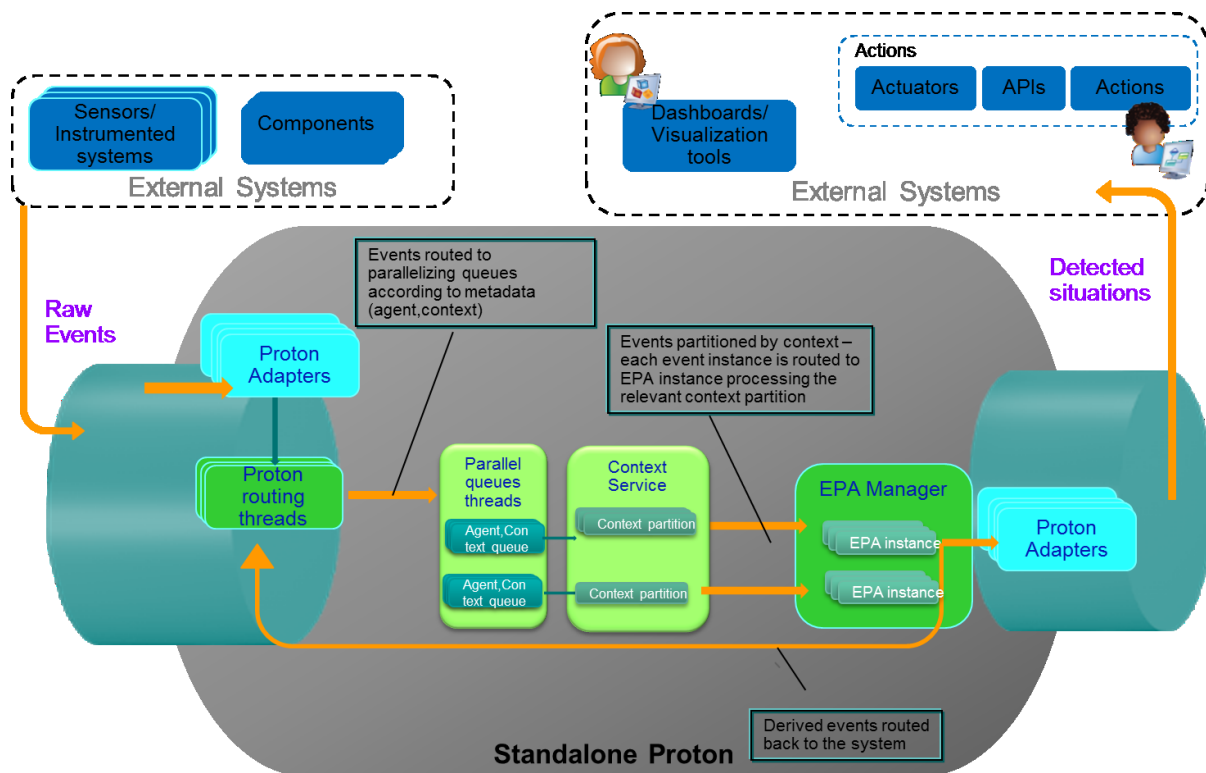


Figure 5: PROTON standalone high level architecture

PROTON's logical components are illustrated in Figure 6. The queues, the context service, the EPA manager are purely java-based. They utilize dependency injection to make use of the infrastructure services they require, e.g. work manager, timer services, communication services. These services are implemented differently for the J2SE and J2EE versions.

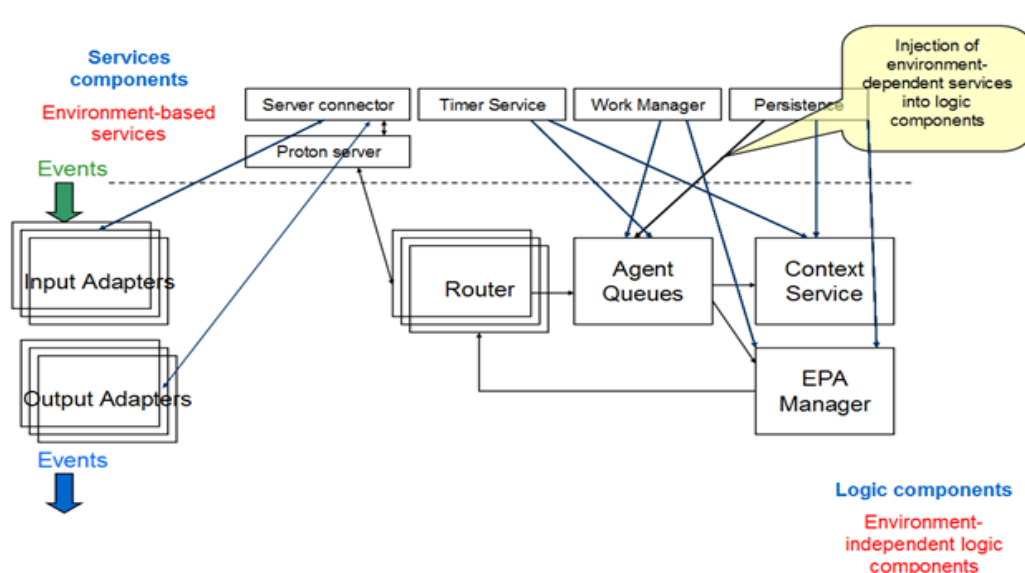


Figure 6: PROTON's components

3.2.2 Technological stack

Item	Value
Nature	Web application
Programming Language	Java
Development Tools	Eclipse
Additional Libraries	commons-httpclient (for RESTful interface) slf4j package for logging
Application Server	tested with but not limited to Apache Tomcat
Databases	N/A

3.2.3 Technical Manual

In PROTON, the JSON CEP application definitions file can be created in three ways:

1. Build-time user interface – By this, the application developer creates the building blocks of the application definitions. This is done by filling up forms without the need to write any code. The file that is generated is exported in a JSON format to the CEP run-time engine.
2. Programming – The JSON definitions file can alternatively be generated programmatically by an external application and fed into the CEP run-time engine.
3. Manually – The JSON file is created manually and fed into the CEP run-time engine.

The created JSON file comprises the following definitions:

- Event types – the events that are expected to be received as input or to be generated as derived events. 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

Project ID 636804	PSYMBIOSYS – Product – Service sYMBIOtic SYStem	
Date: 31/01/2016	D6.3 – Real World Event Driven Architecture and Events Processing First	

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

Instructions for installation, configuration, and administration of the software can be found at:

<http://proactive-technology-online.readthedocs.org/en/latest/Proton-InstallationAndAdminGuide/index.html>

3.2.4 Licensing

Apache 2

3.3 User Manual and use-case

PROTON's user manual can be found at: http://proactive-technology-online.readthedocs.org/en/latest/ProtonUserGuide_FI_WAREv4_4_1/index.html

Specifically, the configuration file for the AIDIMA event-driven application (see following sections) can be found at: <http://demos.txt.it:8096/intranet/wp6/m12-deliverables/aidima-usecase-application-defs-and-execution-instructions>

3.3.1 Description of the scenario use case

The idea is to provide a comprehensive workplace monitoring and renovation service, as a bundle with the manufactured furniture. The functionalities of the service will allow the definition of optimised and fully customer-oriented renovation project proposals to the customer.

The scenario is based on a furniture renovation project of working environments. In this scenario we aim to measure both physiological and emotional factors at the office, in order to design an improve project proposal for the customer. Focusing on the physiological side, we are interested in a monitoring scenario in which factors about the use of the furniture by the worker are analysed. The system collects information from sensors regarding workstation conditions; however this information is not leveraged to produce real-time alerts about non-recommended situations at the office environment. The aim of the CEP tool in this use case is to detect potential situations that trigger real time alerts due to physiological factors. To this end, a first EPN has been created with the collaboration of the use case owner with the goal of having something meaningful and representative, yet doable to be achieved in the first year of the project. The outcome is an EPN consisting of five EPAs shown in Figure 7 and detailed in the following sections. For the sake of simplicity we only show the EPAs and the events flow in the network.

Specifically, in the current EPN we want to fire situations in the following cases (for detailed descriptions of each EPA see Section 3.3.3)

- Temperature at the workplace might cause discomfort as it is too high (*HighTemperatureAlert*)
- Noise level might cause discomfort as it is too high (*NoiseLevelAlert*)
- A worker is sitting at the same workplace for too long (*UserSittingMoreThanOneHourAlert*)

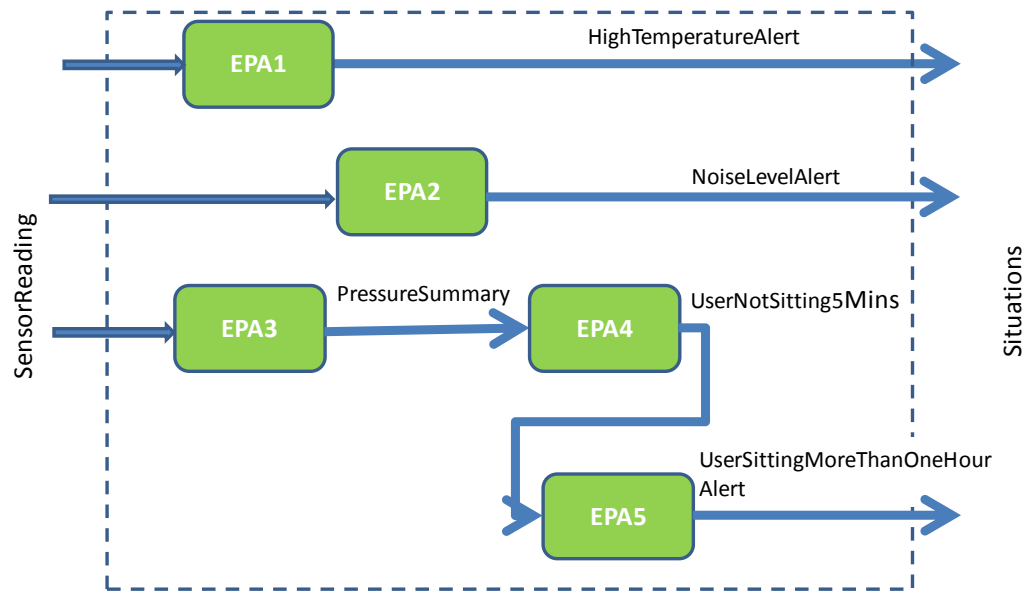


Figure 7: EPN

3.3.2 Define Event types

Six event types have been defined so far that comprise the event inputs, outputs/derived, and situations as shown in Table 1. For the sake of simplicity we only show the user-defined attributes or the event payload and not the metadata of the event.

There is only one raw event, *SensorReading* that holds the information regarding environmental factors at the workplace. Currently, *SensorReading* events are written to a DB and exposed through the following service: <http://62.15.170.194/PSYReport/getEventServlet>. The goal is to get the raw events directly from the sensors in real-time.

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

Table 1: EPN

Event name	SensorReading
Payload	sensorID; sensorType; workstationID; userID; sensorValue
Event name	HighTemperatureAlert
Payload	workstationID
Event name	NoiseLevelAlert
Payload	workstationID
Event name	UserSittingMoreThanOneHourAlert
Payload	workstationID ; userID
Event name	PressureSummary
Payload	workstationID; ; userID; PressureSum
Event name	UserNotSitting5Mins
Payload	workstationID ; userID

Examples of *SensorReading* events for the three types of input measurements:

- 03/12/2015 10:00,ST01,temperature,WP022,u005,22 – This raw event indicates a measurement of type “temperature” for sensor “ST01” with a value of 22 degrees for worker/user “5” at workstation “22” on “Dec. 3, 2015 at 10:00”. Note that the event timestamp is omitted from the event type attributes in Table 1 as this denotes the *Detection time* built-in metadata attribute.
- 03/12/2015 10:03,NS01,noise,WP022,u005,1 – This is a raw event of type “noise” for sensor “NS01” at workstation “22” for worker “5”. Note that for noise measurements we only get values of “1” to indicate a high-level value but not the specific measurement.
- 03/12/2015 10:05,PS33,pressure,WP040,u014,0 – A raw event for “pressure” is received with value of ‘0’ for sensor “PS33”, workstation “40”, and user “14”.

Different *SensorReading* raw events for temperature are illustrated in Figure 8

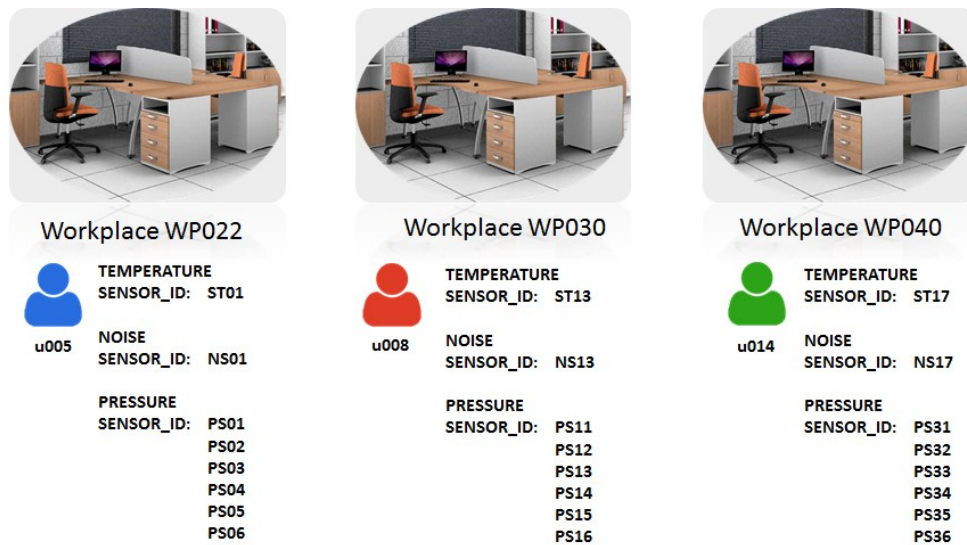


Figure 8: SensorReading temperature raw events examples

3.3.3 Event processing agents

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

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

EPA1: HighTemperature

Motivation: Check that the temperature stays “high”, i.e., ≥ 27 °C for a period of 20 min (alternatively, check after receiving the first “high temperature” measurement that there is no temperature measurement < 27 °C in the coming 20 min).

Event recognition process

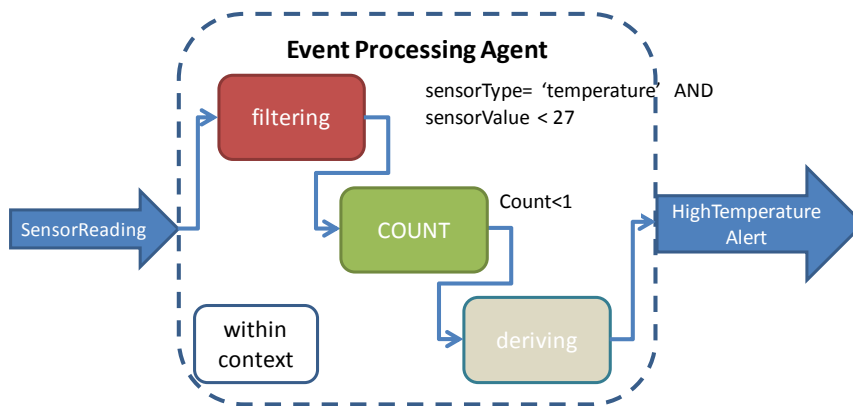


Figure 9: Event recognition process for HighTemperature EPA

Note that the pattern COUNT sums the number of the input event occurrences that pass the filter condition (in our case temperature measurements with values < 27), while *count* is the assertion value for the COUNT pattern. In our case we are looking for zero temperature measurements < 27 .

We apply the *deferred* policy, i.e. we check only at the end of the temporal window (20 min) whether the pattern is satisfied.

Pattern policies

Evaluation	Cardinality	Repeated	Consumption
DEFERRED	SINGLE	FIRST	CONSUME

Context

Segmentation: by workstationID (workspace)

Temporal window:

- Initiator: sensorType='temperature' AND sensorvalue ≥ 27
- Terminator: +20 min

Initiator policy: ADD

Meaning: For each *workplaceID*, the temporal window opens every time a new “temperature” SensorReading with temperature value ≥ 27 °C comes in (as the policy is ADD). For example, in Figure 10, there are three overlapping sliding windows as there are three input events with temperature ≥ 27 °C. We derive two *HighTemperatureAlert* events for the first two windows but there is no derivation at the end of the third window since before this ends there is one input event with temperature < 27 so the pattern matching is not satisfied.

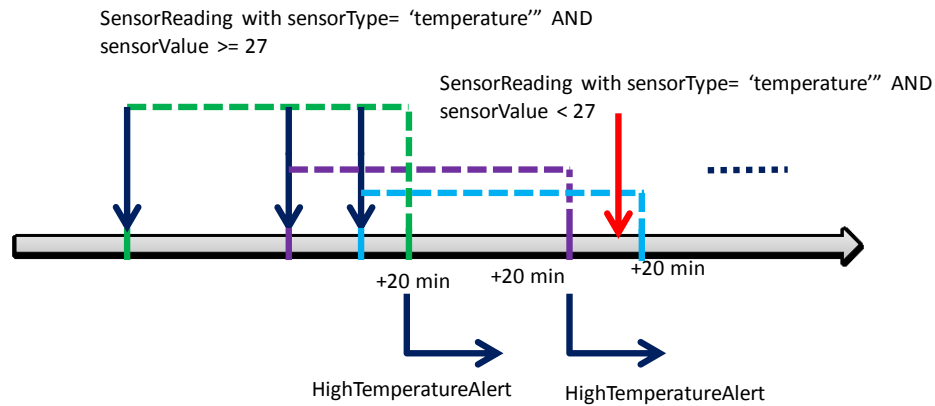


Figure 10: Context for HighTemperature EPA

EPA2: NoiseLevel

Motivation: Check that the number of *SensorReading* events with high level of noise at a workspace in 5 min is above 4.

Note that we get a “noise” *SensorReading* event with sensorValue = “1” every time there is a high noise level per workstation.

Event recognition process

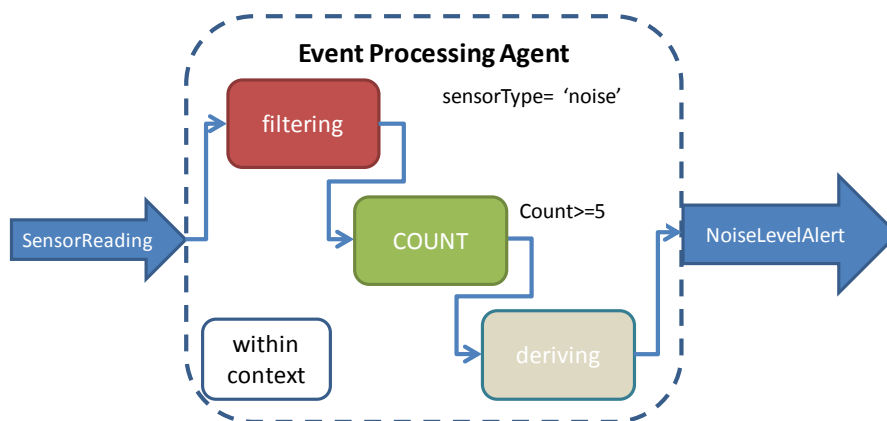


Figure 11: Event recognition process for NoiseLevel EPA

Pattern policies

Evaluation	Cardinality	Repeated	Consumption
IMMEDIATE	SINGLE	FIRST	REUSE

That is, we derive only once as the pattern matching is met.

Context

Segmentation: by workstationID

Temporal

- Initiator: sensorType='noise'
- Terminator: +5 min

Context policy: ADD

Meaning: A new temporal windows opens with a new incoming of a *SensorReading* of type “noise” and closes after 5 min. In Figure 12, three temporal windows are shown. For the first two a derived event is fired since the number of “noise” *SensorReading* input events is at least 5, while for the third one is only 4 therefore no derivation is made.

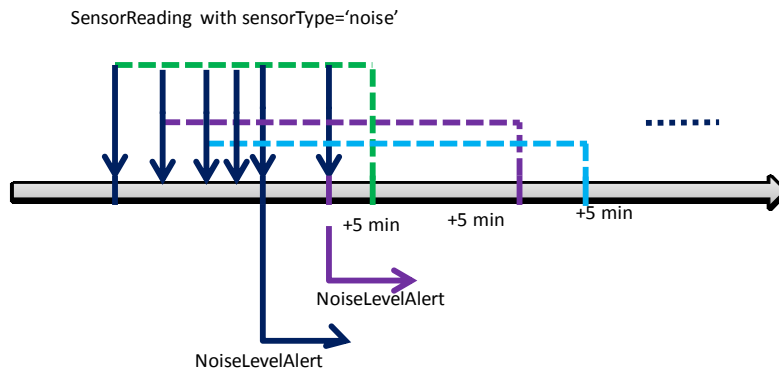


Figure 12: Context for NoiseLevel EPA

The third situation we want to detect is (*UserSittingMoreThanOneHourAlert*) in which we alert if a user doesn’t “get up” from a working station in one hour for more than 5 min.

In order to do so we need three EPAs:

- **EPA3 PressureSummary:** For each workstation every minute sums up the value of the 5 pressure sensors in the workstation and emits the sum value.
- **EPA4 UserNotSitting5Min:** For each workstation emits a derived event if the user is not sitting for 5 min.
- **EPA5 UserSittingMoreThanOneHour:** For each workstation alerts if the user doesn’t get up for more than 5 min in an hour of being sitting (the situation).

EPA3: PressureSummary

Motivation: Check every minute the sum of the 5 pressure sensors for a workstation. Note that if the workstation is idle (no user is sitting) then $PressureSum == 0$.

Note that the pattern SUM sums the *sensorValue* values of the input events that pass the filter condition, while *Sum* is the assertion (the value to be exceeded).

Event recognition process

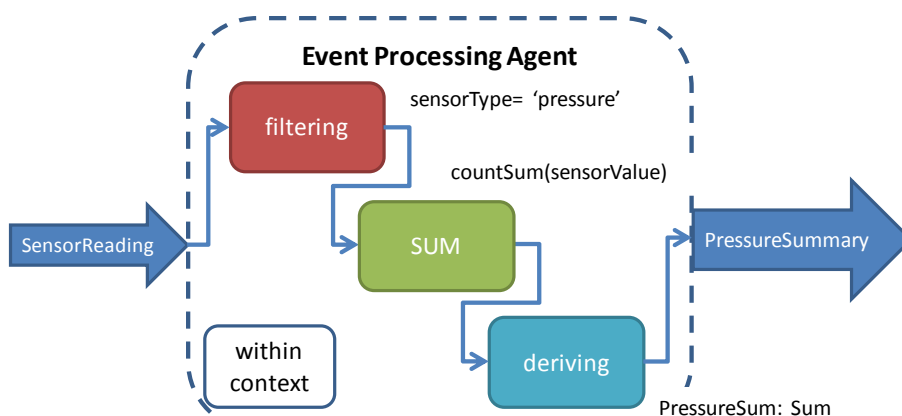


Figure 13: Event recognition process for PressureSummary EPA

Pattern policies

Evaluation	Cardinality	Repeated	Consumption
DEFERRED	SINGLE	FIRST	CONSUME

Context

Segmentation: by workstationID

Temporal

- Initiator sensorType='pressure'
- Terminator: +1 min

Context policy: IGNORE

Meaning: A new temporal windows opens with a first incoming of a SensorReading of type “pressure” and closes after 1 min. As the context policy is IGNORE and the pattern policy is DEFERRED then we derive a *PressureSummary* event at the end of the minute.

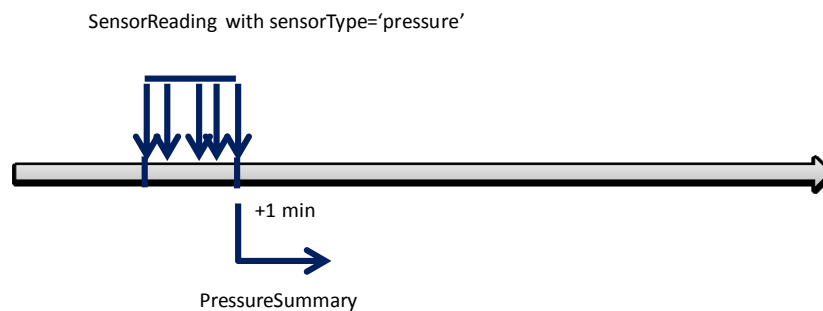


Figure 14: Context for PressureSummary EPA

EPA4: UserNotSitting5Mins

Motivation: For each workstation emits a derived event if the user is not sitting for 5 min (alternatively, there is no input *PressureSum* event with value>0 in the period of 5 min). Note that the input event to this EPA is *PressureSummary* derived from EP3.

Event recognition process

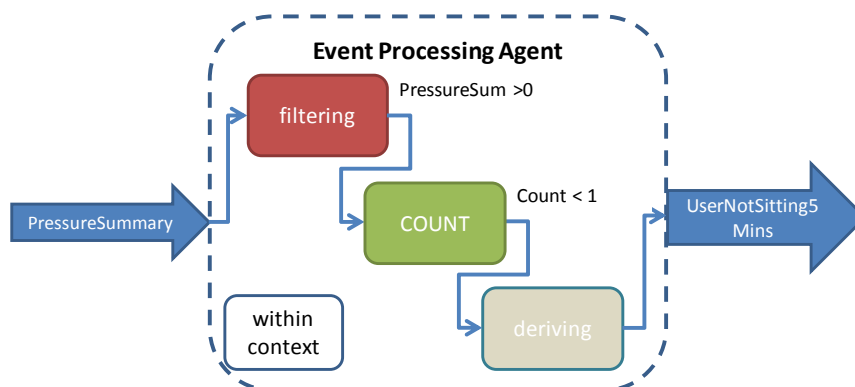


Figure 15: Event recognition process for UserNotSitting5Mins EPA

Pattern policies

Evaluation	Cardinality	Repeated	Consumption
DEFERRED	SINGLE	FIRST	CONSUME

Context

Segmentation: by workstationID

Temporal

- Initiator: PressureSum==0
- Terminator: +5 min

Context policy: IGNORE

Meaning: A new temporal windows opens with a first incoming of a PressureSummary event with PressureSum==0 and closes after 5 min.

To illustrate the context we distinguish between two cases: first case in which we have a derived event at the end of the temporal window, and second case in which no derivation takes place (Figure 16 and Figure 17 accordingly).

Case 1: As there are no input events with PressureSum>0 then we conclude that the worker/user is not sitting during the 5 min of temporal window.

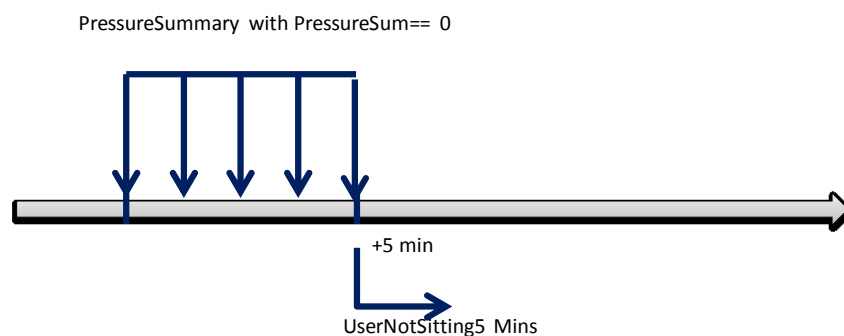


Figure 16: Context for UserNotSitting5Mins EPA (case 1)

Case 2: As there are two input events with PressureSum>0 then we conclude that the worker/user is sitting part of the time window and there is no derivation at its end.

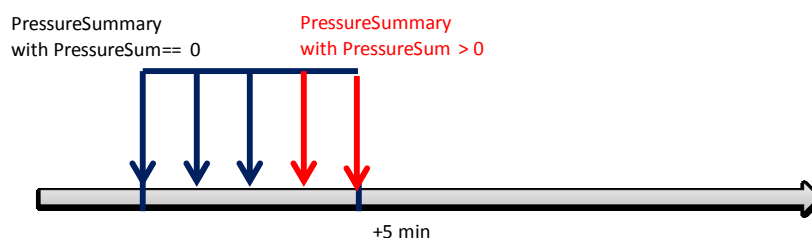


Figure 17: Context for UserNotSitting5Mins EPA (case 2)

EPA5: UserSittingMoreThanOneHour

Motivation: For each workstation alerts if the user doesn't get up for more than 5 min in an hour of being sitting. This is the alert or situation to be notified and uses outputs from the previous two EPAs.

Event recognition process

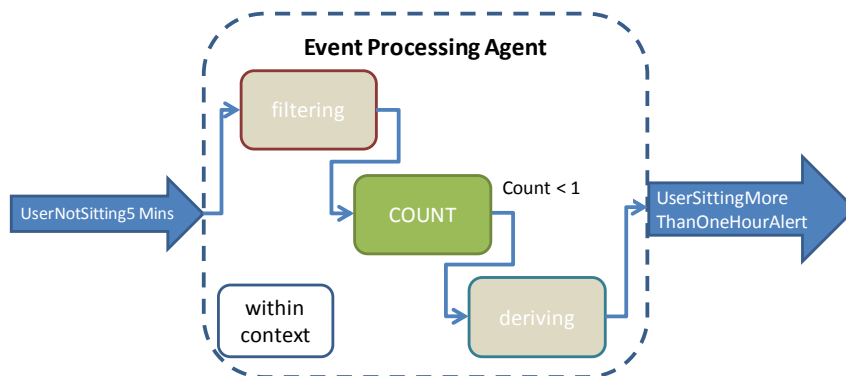


Figure 18: Event recognition process for UserSittingMoreThanOneHour EPA

Pattern policies

Evaluation	Cardinality	Repeated	Consumption
DEFERRED	SINGLE	FIRST	CONSUME

Context

Segmentation: by workstationID

Temporal

- Initiator: PressureSum > 0
- Terminator: +60 min

Context policy: ADD

Meaning: A new temporal windows opens with a new incoming of a *PressureSummary* event with PressureSum ==0, and closes after 60 min. In Figure 19, three temporal windows are shown. For the first one a derived event is fired since there are no input events of type *UserNotSitting5Min* while in the two consecutive windows there is one input event therefore the COUNT pattern is not satisfied and there is no derived event.

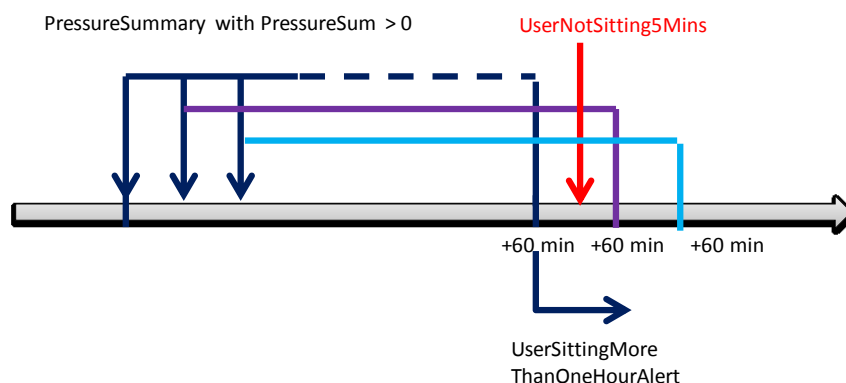


Figure 19: Context for UserSittingMoreThanOneHour EPA

3.3.4 Visualisation

The application uses an input file as the events producer and an output file as the events consumer as explained henceforth.

The events input file were stored as a CSV file and injected into PROTON. The output events along with the detected situations (alerts) are sent and shown in PROTON's dashboard that serves as the event consumer (see a screenshot of this dashboard in Figure 20). The goal is that in the future, the sensors will be connected and the processing of the sensor events will be done in real-time instead of storing them before injecting them into the CEP engine. The consumer can be any of the project dashboards or UI selected to show alerts in real-time.

The input file includes the readings on temperature, noise, and pressure sensors of three different workspaces over the span of 8 hours, an overall 4700 *SensorReadings* events.

The noise readings are provided only when a peak in noise (value == 1) is perceived, the pressure readings come from 6 sensors per workspace and are provided every minute, and the temperature readings are provided every 5 mins.

After testing the logic of the patterns, we run the application over a span of 2 hours on *SensorReadings* raw events received between 11:28 to 13:29. The truncated data file includes 2300 sensor readings. This truncated data file has been injected into the PROTON as explained before.

We have detected *NoiseLevelAlert* for WP030, and multiple *NoiseLevelAlerts* for WP022, multiple *HighTemperatureAlerts* for WP022, and *UserSittingMoreThanOneHourAlert* for users u014 and u005 (in WP040 and WP022 respectively).












PROTON DASHBOARD							
 NoiseLevelAlert		<input type="text" value="Search"/>					
_protonId		a22503a7-10a3-4756-a2f1-249c2255f93d					
OccurrenceTime		06.01.2016 AD at 15:38:10 IST					
EventSource							
Certainty		1					
workstationID		WP030					
	Name	sensorID	sensorType	workstationID	userID	sensorValue	readingDate
	NoiseLevelAlert	WP030
	NoiseLevelAlert	WP030
	SensorReading	PS05	pressure	WP022	u005	600	03.12.2015 AD at 11:31:00 IST
	SensorReading	NS13	noise	WP030	u008	1	03.12.2015 AD at 11:31:00 IST
	SensorReading	NS13	noise	WP030	u008	1	03.12.2015 AD at 11:31:00 IST
	SensorReading	PS06	pressure	WP022	u005	0	03.12.2015 AD at 11:30:00 IST
	SensorReading	NS13	noise	WP030	u008	1	03.12.2015 AD at 11:30:00 IST
	SensorReading	NS13	noise	WP030	u008	1	03.12.2015 AD at 11:30:00 IST
	SensorReading	NS13	noise	WP030	u008	1	03.12.2015 AD at 11:29:00 IST
	SensorReading	PS06	pressure	WP022	u005	600	03.12.2015 AD at 11:29:00 IST

Figure 20: Screenshot of the application dashboard

3.4 Conclusions and Future plans

In this report we accompany the delivery of the PSYMBIOSYS WP6.2 architecture and components based on the open source PROTON complex event processing tool developed and implemented by partner IBM. Large space has been provided to the scenario demonstration () in order to show the capabilities of the architecture, how to technical implement the demonstration on top of the software and all the interfaces composing the release. The scenario is based on IoT for

Project ID 636804	PSYMBIOSYS – Product – Service sYMBIotic SYStem	
Date: 31/01/2016	D6.3 – Real World Event Driven Architecture and Events Processing First	

manufacturing intelligence in the context of a monitoring scenario in which factors such as noise, pressure, and temperature given by sensors at a workstation are analysed in order to design an improved environment

Next steps in the scope of WP6 is the integration in the overall platform of WP6 in WP6.4 by M12 (D6.7). WP6 platform will be then integrated into the overall platform in WP9.2 and into end-user environments in WP10 to be used and validated by the,. The next and final version of the document is planned to be provided by M27 (D6.4).

4 References

- [1]. Babcock B., Babu S., Datar M., Motwani R., and Widom J. 2002. Models and issues in data stream systems. In *Proceedings of the 21st ACM SIGMOD/PODS Symposium on Principles of Database Systems (PODS'02)*. ACM, New York, NY, 1–16.
- [2]. Luckham D. 2001. The power of events: an introduction to complex event processing in distributed enterprise systems. Addison-Wesley Longman Publishing Co., Inc.
- [3]. Shulte R. 2014. *An Overview of Event Processing Software* (August, 25, 2014), [Online]. At: <http://www.complexevents.com/>
- [4]. Etzion O. and Niblett P. 2010. Event Processing in Action. Manning Publications Company.
- [5]. LeHong H. and Velosa A. 2014. *Hype Cycle for the Internet of Things*. Gartner report G00264127. Published: 21 July 2014.
- [6]. Manyika J, Chui M., Bisson P., Woetzel J., Dobbs R., Bughin J., and Aharon D. 2015. *Unlocking the potential of the Internet of Things*, McKinsey Global Institute, Published: June 2015.
- [7]. Lheureux B.J., Velosa A., Friedman T., Pezzini M., Forsman J., Schulte W.R., Sallam R.L., Perkins E., Thomas A., Cantara M., and Guttridge K. 2015. *Best Practices in Exploring and Understanding the Full Scope of IoT Solutions*. Gartner report #G00274014. Published: 26 March 2015.
- [8]. Thompson J. 2014. *Hype Cycle for Application Infrastructure, 2014*. Gartner report # G00263159. Published: 29 July 2014.
- [9]. Velosa A., Schulte W.R., and Lheureux B.J. 2015. *Hype Cycle for the Internet of Things, 2015*. Gartner report #G00272399, Published: 21 July 2015