FInest – **F**uture **In**ternet **e**nabled optimi**s**ation of **t**ransport and logistics networks

D6.3

Initial technical specification of event processing component

Project Acronym	FInest	
Project Title	Future Internet enabled optimisation of transport and logistics networks	
Project Number	285598	
Workpackage	#WP6 Proactive event driven monitoring	
Lead Beneficiary	IBM	
Editor	Fabiana Fournier	IBM
Contributors	Ella Rabinovich	IBM
	Guy Sharon	IBM
Reviewers	Rod Franklin	KN
	Clarissa Marquezan	UDE
	Andreas Metzger	UDE
Dissemination Level	Public	
Contractual Delivery Date	30/9/2012	
Actual Delivery Date	30/9/2012	
Version	V1.0	

The research leading to these results has received funding from the European Community's Seventh Framework Programme [FP7/2007-2013] under grant agreement no. 285598

Abstract

This report presents the third deliverable of work package 6 "Proactive Event Driven Monitoring". Work package 6 is accountable for the Event Processing Module (EPM), one of the four core technical modules of the FInest platform. The role of this module within FInest is to provide event-driven monitoring capabilities to facilitate the end-to-end visibility of logistics processes and real-time information regarding actual execution.

This document describes the initial technical specification of the EPM module in FInest and its interactions (in terms of interfaces) with other modules in the platform as well as with generic enablers offered by FI-WARE. FInest EPM can act in both reactive and proactive modes. FInest EPM proactive capabilities differentiate its run-time engine from currently available complex event processing engines, and position it in the technological and research frontage of event processing systems.

Document History

Version	Date	Comments
V0.1	20/8/12	First draft
V0.2	30/8/12	Updates after project technical meetings
V0.3	1/9/12	Updates after IBM internal review
V0.4	7/9/12	Updates after project face-to-face meeting
V0.5	20/9/12	Updates after project review
V1.0	30/9/12	Submitted version

Table of Contents

Abstract 2

Document History 3

Table of Contents 4

List of Tables 5

List of Figures 6

Acronyms 7

1. Introduction 8

2. Background 8

2.1. Event producers and consumers 9

2.2. Event processing run-time engine 10

2.2.1. Pattern detection 11

2.2.2. Conditions 12

2.2.3. Actions 13

2.2.4. Event processing agents 13

2.2.5. Context 14

2.3. Proactive event-driven computing 15

2.3.1. The proactive principle 15

2.3.2. Forecasted events 16

2.3.3. Proactive event processing agents 16

3. Methodology 17

4. FInest EPM architecture overview 18

4.1. Event types and attributes 18

4.1.1. Proactive attributes 19

4.2. Interfaces 20
4.2.1. Input and output adapters 20
4.3. Definition of FInest complex event processing application 22
4.4. FInest EPM instantiation 23
4.5. Interaction between EPM and other modules in FInest platform 23
5. FInest EPM technical specification 26
5.1. EPM high level architecture 26
5.2. EPM complex event processing run-time technical specification 27
5.3. EPM interfaces with other components 28
5.4. FInest EPM with FI-WARE GEs 30
5.5. Interfaces data types and definitions 31
6. Summary 33
7. References 33
Appendix A – Event processing network for the FISH use case 35

List of Tables

Table 1: Methods definitions by interfaces 32
Table 2: (partial) List of events for the FISH use case 35
Table 3: (partial) Rule set for the FISH use case 39
Table 4: Event processing network for the FISH use case 43
Table 5: Proactive event processing rules for the FISH use case 44

List of Figures

Figure 1: Functional view of event processing [1] 9

Figure 2: Illustration of an event processing network 11

Figure 3: Event processing agent types 12

Figure 4: Event processing agent architecture 13

Figure 5: Proactive event driven computing principle 15

Figure 6: WP6 D6.3 methodology 18

Figure 7: EPM interfaces 20

Figure 8: EPM interactions with other components 25

Figure 9: EPM high level architecture 26

Figure 10: EPM complex event processing run-time architecture 27

Figure 11: Flow of events into, within, and out of the complex event processing engine 28

Figure 12: EPM interactions with other components 29

Figure 13: CEP engine set-up 29

Figure 14: FInest EPM with FI-WARE GEs 30

Figure 15: Interfaces and data types 31

Figure 16: EPM interfaces and methods 31

Acronyms

Acronym	Explanation
BCM	Business Collaboration Module
CEP	Complex Event Processing
ECA	Event-Condition-Action
ECM	E-Contracting Module
EPA	Event Processing Agent
EPM	Event Processing Module
EPN	Event Processing Network
GE	Generic Enabler
FInest	Future Internet enabled optimisation of transport and logistics networks
JMS	Java Message Service
JSON	JavaScript Object Notation
IoT	Internet of Things
PRA	PRoactive event processing Agent
TCP	Transport Chain Plan
TEP	Transport Execution Plan
TPM	Transport Planning Module
WP	Work Package

1. Introduction

Complex Event Processing (CEP) is the analysis of event data in real-time to generate immediate insight and enable instant response to changing conditions. Some functional requirements this technology addresses include event-based routing, observation, monitoring, and event correlation. The technology and implementation of CEP provide means to expressively and flexibly define and maintain the event processing logic of the application. At runtime it is designed to meet all functional and non-functional requirements without taking a toll on application performance, removing one issue from the application developer's and system manager's concerns.

The Event Processing Module (EPM) is one of the four core technical modules in the FInest (Future Internet enabled optimisation of transport and logistics networks) project. Its role is to collect events from various sources and perform complex event processing on them in order to detect situations of interest; that is, of relevant meaning to the consumer of the event enabling them to react or make use of the event appropriately. In FInest, these *detected situations* (aka *derived events*) are notified to the Business Collaboration Module (BCM), or to the frontend of FInest, so appropriate actions can be taken, e.g. triggering re-planning of the transport plan.

This report refines the conceptual design of the EPM delivered in Month 12 of the project and delivers the initial technical specification of the FInest EPM Module, including its interactions with other core modules of FInest, and the potential use of FI-WARE Generic Enablers (GEs). It should be noted that work on "preliminary conceptual prototypes" (initially planned to be submitted in Month 18 as part of this deliverable) has already been successfully delivered as part of deliverable 6.2 "Conceptual design of event processing component" at M12, and therefore it is omitted from this report.

This document is organized as follows: Section 1 introduces the report scope and main goals. Section 2 gives some background regarding event processing necessary to understand the terminology used in the work. Section 3 describes the methodology that has been followed. Section 4 gives an overview of the FInest EPM while Section 5 details the proposed architecture and interfaces. The report concludes with a brief summary.

2. Background

The designer of event processing logic is responsible for creating event specifications and definitions, including information concerning the sources of the events. The designer is also responsible for discovery and understanding of existing event definitions.

Generally speaking, an *event* is an occurrence within a particular system or domain; it is something that has happened, or is contemplated as having happened in that domain. The word "event" is also used to mean a programming entity that represents such an occurrence in a computing system. In the latter definition, an *event* is an object of an *event type*. Events are actual *instances* of the event types and have specific values. For example, the event "today at 10 PM a customer named John Doe booked a new order" is an instance of the Order event type.

The EPM includes the following two conceptual components:

- Tools to define event processing applications on data interpreted as events, either manually or programmatically.

- Execution services that process events as they occur and generate derived events accordingly.

These functions align with the functional architecture view produced by the Reference Architecture Work Group of the Event Processing Technical Society (EPTS) [1], depicted in Figure 1. From an architect and developer perspective, the architecture includes three main functions: development, run-time, and administration. The run-time involves two additional functions: producers and consumers, and the event processing run-time engine.

Henceforth, we present main concepts and constructs related to event processing tools and applications relevant to the work in FInest. The terminology used in this report is based on, and follows, the event processing language presented in [2].

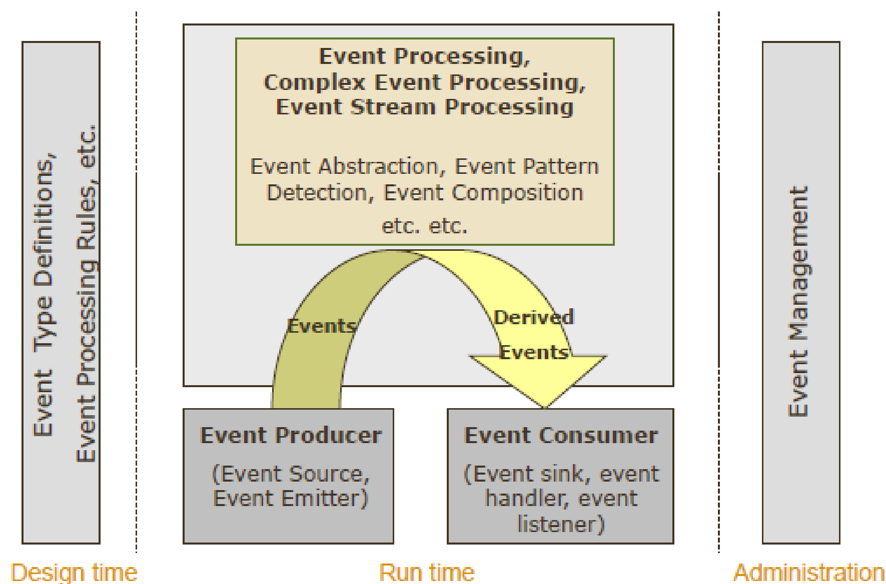

Figure 1: Functional view of event processing [1]

2.1. Event producers and consumers

Entities connected to the CEP engine can play two different roles: the role of *event producer*s or the role of *event consumers*. Note that nothing precludes a given entity from playing both roles.

An event producer is an entity at the edge of an event processing system that introduces events into the system. Event producers are the source of events for event processing. They can provide events in two modes:

- *"Push" mode*: The event producers push events into the complex event processing module by means of invoking a standard operation the module exposes (see Section 4.2).

- "*Pull*" mode: The event producer exports a standard operation that the complex event processing module can invoke to retrieve events.

An event consumer is an entity at the edge of an event processing system that receives events from the system. Event consumers are the sink point of events. Following are some examples of event consumers:

- *Dashboard:* a type of event consumer that displays alarms defined when certain conditions hold on events related to some user community or produced by devices of interest.
- *Applications:* a type of event consumer if it consumes events for its own processes.
- *Publish/Subscribe:* a type of event consumer that forwards the events it consumes to all interested applications based on a subscription model.

In the FInest EPM, we identify three event producers,

1. *BCM (Business Collaboration Module):* events related to the actual execution of a transport plan.
2. *IoT (Internet of Things):* events received from sensors like RFID tags or GPS systems.
3. *Backend systems:* events originated from systems external to FInest, such as booking systems.

and two consumers of FInest EPM events,

1. *BCM:* events related to actual execution plans, such as pickup arrival, change in shipment status, and schedule delay. The BCM, in turn, might transmit some of these events as notifications to the user via FInest frontend.
2. *FInest frontend:* proactive notifications regarding future probable events not directly related to the actual/real-time execution of the transport plan (see Section 2.3 for a description of proactive event-driven computing).

2.2. Event processing run-time engine

The CEP run-time engine in FInest implements event processing functions based on the design and execution of *Event Processing Networks* (EPNs). Processing nodes that make up these networks are called *Event Processing Agents* (EPAs). 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 2, events from Producer 1 are processed by Agent 1. Events derived by Agent 1 are of interest to Consumer 1 but are also processed by Agent 3 together with events derived from Agent 2. Note that the intermediary processing between producers and consumers in every installation is made up of several functions and often the same function is applied to different events for different purposes at different stages of the processing. The EPN approach allows dealing with this in an efficient manner, because a given agent may receive events from different sources. Another benefit in representing event processing applications as networks is that entire networks can be nested as

agents in other networks allowing for reuse and composition of existing event processing applications.

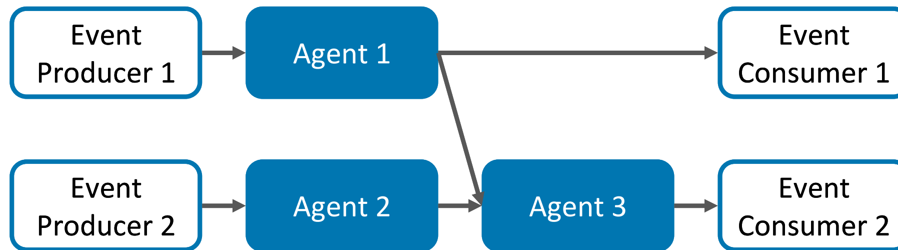

Figure 2: Illustration of an event processing network

The event processing network can be programmatically developed and deployed on the fly at execution, or it can be developed by applying form based tools. In the latter case, the network constructs can be manually deployed at execution, or on the fly.

The event processing agents (see detailed description in Section 2.2.4) and their assembly into a network is where most of the functions of the EPM in FInest are implemented. The behavior of an event processing agent is specified using a rule-oriented language that is inspired by the ECA (Event-Condition-Action) [8] concept and can be described as Pattern-Condition-Action. Rules in this language consist of three parts:

- A pattern detection that makes a rule of relevance
- A set of conditions (logical tests) formulated on events as well as external data
- A set of actions to be carried out when all the established conditions are satisfied

In the following sections, we describe the capabilities to be supported in each part of the rule language. A description of how such rules can be assembled is given in the event processing agent section (Section 2.2.4.).

2.2.1. Pattern detection

In the pattern detection part, the user may define patterns over selected events within an event processing context (such as a time window or segmentation, see Section 2.2.5). Only if the pattern is matched, the rule is of relevance, and according to its conditions, the action part is executed. A pattern matching process takes the participant events as an input and creates a pattern matching set, consisting of the event instances that satisfy the pattern.

Pattern detection is a type of EPA (see Section 2.2.4) that enables the analysis of collections of events and the relationship between them. Pattern detection is one of the EPA types as depicted in Figure 3. Informally, we say that a conditional combination of events matches a pattern if this combination satisfies the particular pattern definition. Pattern matching EPA examples are: *all*, *sequence*, *absence*, *any*, *and trend* (see brief explanations below).

Note that processing of a rule (established by the EPA type) can be *stateless* or *stateful*. The stateless EPA induces that the processing applies to a single event and is only formulated over properties of the event, while stateful processing applies to multiple events within a certain time frame. Most pattern detection EPAs are stateful, involving both detection of multiple event combinations and temporal semantics. Other EPA types (except *Aggregation*) usually represent

stateless EPAs, and thus include simpler functions. Examples for stateless EPAs are *Filter (aka Threshold)*, *Split*, and *Compose* (for a full description of EPA types refer to [2]).

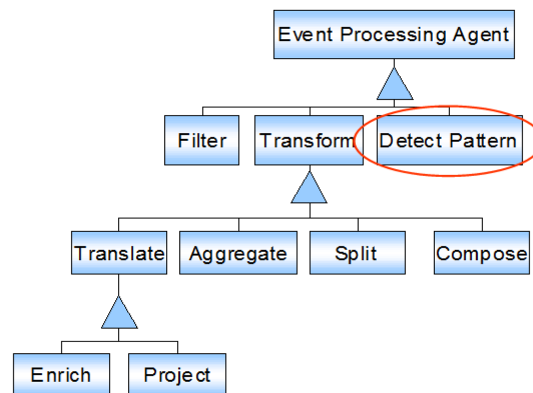

Figure 3: Event processing agent types

Examples for pattern matching EPAs (see Table 3 in Appendix A for examples of rules applying pattern matching EPA in the FISH use case) are:

- *Sequence*, means that at least one instance of all participating event types must arrive in a specified order for the pattern to be matched.
- *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 over time of some observed value; this refers to the value of a specific attribute or attributes.
- *Absence*, a specified event(s) must not occur within a predefined time window. The matching set in this case is empty.
- *Any*, the participant event set contains an instance of any of the event types in the relevant event types list. A matching set for the any pattern contains just one member.

2.2.2. Conditions

Each EPA, in particular pattern detection EPA, may incorporate two types of conditions:

1. Simple conditions, which are established as predicates defined over single events of a certain type. Such a condition is also known as an *event threshold* condition.
2. Complex conditions, which encapsulate several event types, or several event instances of the same type, and are denoted by "cross-event" conditions.

A single EPA can encapsulate both simple and complex conditions, for example:

All(A,B) where A.price > 1000 and A.amount > B.amount

This agent is looking for the arrival of at least one instance of event type A and event type B with simple condition (*A.price > 1000*) and complex condition (*A.amount > B.amount*).

Conditions on events are expressed based on a number of predefined operators applicable over:

- Values of event data fields.
- Values of other properties inherent to an event (e.g., lifetime of the event).
- External functions the CEP engine can invoke and to which event data field values or event property values can be passed.

2.2.3. Actions

Action is usually a user-defined operation that is triggered upon detection of a certain pattern. Examples for actions are: sending an email or SMS, database update, or invocation of external services that allows achieving some desired effect in the overall system.

Action execution is external to the CEP engine, however, an action can result in a return code that is of interest for further processing – the return code, for instance, can later be referred to by an event processing agent. Action definition in event processing applications may include parameters needed for action execution.

2.2.4. Event processing agents

To simplify the specification of an event processing agent, a framework based on a number of building blocks is provided. This allows the user to express their intent without the need to write logical statements that can be sometimes long and difficult to understand. Users can define EPAs by specifying three building blocks that make up an agent, as well as input and output terminals that specify the type of events to be considered in a rule and the type of events that may be derived by the rule respectively. This is depicted schematically in Figure 4.

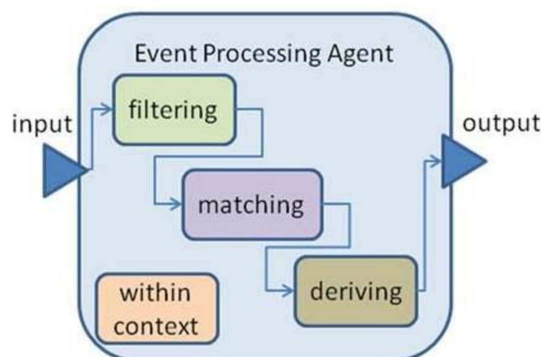

Figure 4: Event processing agent architecture

Only events of the type that have been specified in the input terminals of the agent will be considered for the rule. Then, the following building blocks are defined:

- The first building block is the filter block, an optional block, where filters can be applied on individual input events in order to decide whether to consider them or not in further execution of the rule. For example, in FInest, we can isolate those events that may affect a specific shipment; for example, filter traffic reports from the specific highway numbers that are relevant to the shipment. The filter is referred to as a simple condition in Section 2.2.2.

- The second building block is the matching block, an optional block, where the events are matched against a specified pattern, and where matching sets are created. This is the process where pattern detection (Section 2.2.1) is done.
- The final building block is the derivation block, the only mandatory block, where matching sets, filtered events, or direct input events are used to compose one or more derived events according to conditions and expressions. The *derived events* are specified in the output terminals to be selected as inputs by other agents. Note that derived events are emitted from one EPA to another in the EPN. When these events are emitted to a consumer, they are called *detected situations*.

The different types of EPAs may implement subsets of the components depicted in Figure 4. Pattern detection is the only EPA type that implements the three parts, i.e. filtering, matching, and deriving. Others may have filtering and deriving, or even only deriving component (the only obligatory one). For example, Filter (Threshold) EPAs implement filtering and deriving components, Enrich and Aggregation EPAs implement (optionally) filtering and deriving (refer to [2]). The entire rule may be executed in a processing context that is specified at the level of the EPA (see Section 2.2.5 below). In order to specify EPAs programmatically, the syntax of the building blocks is given and the artifacts are deployable through APIs to the processing engine.

2.2.5. Context

Event processing context is defined as a named specification of conditions that group event instances so that they can be processed in a related way. It assigns each event instance to one or more *context partitions*. A context may have one or more context dimensions and can give rise to one or more context partitions. A context dimension states what aspect of the event is used to do the grouping. Four context dimensions are identified: temporal, spatial, state-oriented, and segmentation-oriented. A context can also be a composite context, i.e., one made up of other context specifications.

A *temporal context* consists of one or more time intervals, possibly overlapping. Each time interval, or window, corresponds to a context partition, which contains events that occur during that interval. An example can be an EPA that raises an alert if someone attempts to make more than three withdrawals from an ATM machine within a single day.

A *spatial context* groups event instances according to their geospatial characteristics. This type of context assumes that the event contains an attribute that assigns a location to the event.

A *state-oriented* context is determined by the state of some entity that is external to the event processing system. For example, an airport security system could have threat level statuses taking values green, blue, yellow, orange, or red. Some events may need to be monitored only when the threat level is orange or above whereas other events may be processed differently in different threat levels.

A *segmentation-oriented* context is used to group event instances into context partitions based on the value of an attribute or collection of attributes in the instances themselves. As a simple example, consider an EPA that takes a single stream of input events, in which each event contains a customer identifier attribute. The value of this attribute can be used to group events so there is a separate context partition for each customer. Each context partition contains only events related to that customer so the behavior of each customer can be tracked independently of the other customers.

2.3. Proactive event-driven computing

Event driven architectures, and conceptual models that support them, have evolved in the last few years. These architectures depart from traditional computing architectures that employ synchronous, request/response interactions between client and servers. This is a paradigm shift in two senses: first, event driven architectures support applications that are reactive in nature, in which processing is triggered in response to events, contrary to traditional responsive applications, in which processing is done in response to an explicit request. Second, event driven architecture adheres to the decoupling principle in which there are event producers, event consumers and event processing agents that are mutually independent. Proactive event-driven computing can be considered as the next phase of this evolution [3, 5]. In fact, the FInest consortium has already been able to demonstrate the potential benefits and opportunities of proactive event-driven computing in the setting of transport and logistics (see [9]).

2.3.1. The proactive principle

We refer to proactive event-driven computing as the ability to mitigate or eliminate undesired states, or capitalize on predicted opportunities—in advance. This is accomplished through the online forecasting of future events, the analysis of events coming from many sources, and the application of online decision-making processes.

The proactive principle can be illustrated through the scenario depicted in Figure 5: For a certain application we define the set of acceptable/desirable states (scoped by the closed curved line in the figure). At time (t - Δ) we forecast that the system is moving towards the red state (an undesirable situation) that it will reach at time (t). Therefore, we change the course of the states (by making a real-time decision), and proactively change the outcome towards the green state. We have a window in which to act ((t-Δ) to t) in order to change the course of states/events.

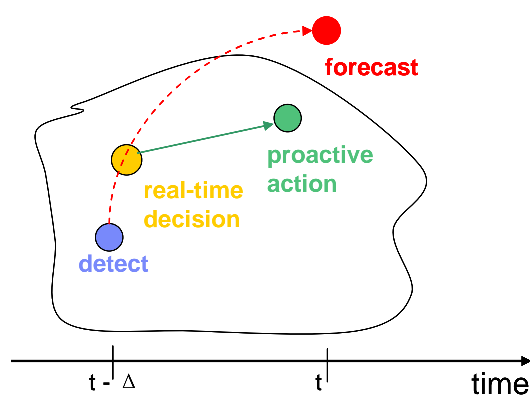

Figure 5: Proactive event driven computing principle

For example, let's assume that we have a passenger flying from A to C via B. At 9:00AM our passenger has to leave airport A. At 8:20AM there are reports regarding severe weather conditions at airport B, which can lead to an airport closure with probability 0.6. At 8:30 AM, our passenger gets a text message regarding potential closure of airport B along with a reschedule of the flight itinerary (via D) without affecting the arriving time at C.

Proactive Event-Driven Computing is a new paradigm where a decision is neither made due to explicit users' requests nor as a response to past events, but is autonomously triggered by forecasting future states, either desired or undesired. The decisions and actions are often real-time in the sense that they are done under time constraints.

The proactive principle extends the reactive pattern known as sense-and-respond or detect-and-act pattern [7] to a novel pattern consisting of four stages:

1. *Detect* events and situations using enhanced event processing techniques.
2. *Forecast/predict* unexpected future events and states before they occur. Forecasting is done by observing ongoing events, analyzing them, and using predictive analytics techniques.
3. *Decide*, within time constraints, how to handle forecasted events and states using real-time optimization techniques.
4. *Act* in the best way possible by adapting (e.g., re-planning/optimizing) the operational system to either mitigate the problem or reach a desired opportunity.

Proactive capabilities require going beyond the state-of-the-art in several areas. First in event processing engines, as there is a need to extend event processing to support the concept of future events (*forecast*); secondly in real-time optimization approaches, as there is a need to make real-time decisions based on events (*decide* and *act*).

The underlying motivation of proactive computing stems from social and economic factors, and is based on the fact that prevention is often more effective than cure [6]. Proactive event-driven applications can be applied to scenarios in a wide range of fields, including medicine, finance, transportation, and security to predict and prepare for future events.

2.3.2. Forecasted events

As mentioned before, proactive capabilities require extending event processing engines to cope with forecasting. That is, the CEP engine needs to analyze on-going events and, based on causality models, to forecast events with a future, and therefore probabilistic, occurrence time. In the FInest CEP engine, forecasted events are expressed by extending the event metadata specification to include three additional attributes. For the list of attributes please refer to Section 4.1. In order to process future events and make real-time decisions, a new building block is introduced to the EPN: *proactive event processing agents (PRA).*

2.3.3. Proactive event processing agents

Proactive event processing agents (PRAs), similar to their (reactive) EPA counterparts, analyze incoming events and, based on some defined rules, derive events. As mentioned before, these are future events with some kind of probability. Therefore, proactive event processing agents involve predictive as well as real-time optimization techniques. In our previous example, the relevant proactive agent detects incoming events regarding potential severe weather conditions, forecasts that there is a chance of 0.6 that the original itinerary cannot be completed, decides on a rescheduling, and acts by sending notification to the passenger along with the new flight itinerary.

3. Methodology

The main purpose of this report is to provide an initial specification of the event processing module in the FInest platform. While this primarily concerns the technical architecture of the proposed module, we do believe that the architecture should be checked against business scenarios and be acceptable by the domain partners as well as by the technical people. Therefore, during the preparation of this report, we worked in close collaboration with the technical team as well as with the domain partners of the project. Specifically we followed the guidelines below, illustrated in Figure 6:

- Regular weekly technical conference calls organized and led by the technical coordinator of the project, with action items and milestones to be achieved towards Month 18 deliverables.
- Conference calls with domain partners involved in the FISH use case, since this was selected by us to be our first scenario for building an event processing application (initial list for events, rules, and event processing network for the FISH use case can be found in Appendix A). This scenario helped us build and validate the EPM and the way the modules interact.
- Ad-hoc technical collateral meetings with other technical WPs (i.e. WP5 and 7) to close any open issues regarding expected interfaces. Specifically, many discussions have been carried out with WP5 since the BCM is the main interaction entity with the EPM. Also, conversations with WP7 regarding the content of the TCP (Transport Chain Plan) and TEP (Transport Execution Plan), and the ways to get the rules required by the CEP engine. The results of these meetings can be summarized in Figure 8, which describes the interactions among the EPM and other modules. One of the main questions answered was which module is responsible for defining the set of relevant rules and their parameters for a new transport plan, and how these will be passed to the event processing module. Feedback from these collateral meetings served as input for (ongoing) refinements of the technical specifications.

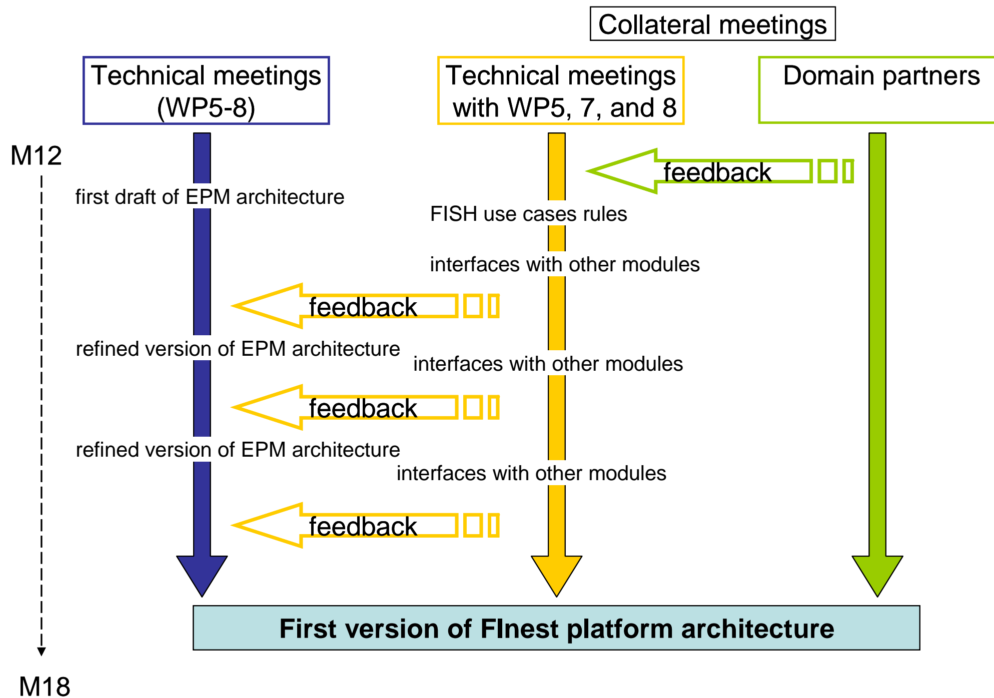

Figure 6: WP6 D6.3 methodology

4. FInest EPM architecture overview

The event processing module in FInest comprises a run-time engine, producers, and consumers with the characteristics and capabilities described in the background section (Section 2). Specifically, the CEP engine of FInest includes an integrated run-time platform to develop, deploy, and maintain reactive and proactive applications using a single programming model.

With regards to proactivity capabilities, the EPM employs a basic model [3], by incorporating PRAs to the EPN (see Section 2.3.3) and sending proactive notifications to the FInest front-end regarding possible future detected situation of interest to the user (see Table 5 as an example for the FISH use case).

Section 4 gives an overview of FInest EPM architecture including event types and interfaces definitions, EPM instantiation, and interactions with other modules in FInest platform.

4.1. Event types and attributes

Events enter the CEP engine during run-time. They carry information about things that happen externally to the engine (raw events) or as a result of processing in the engine (derived events). An *event* is an object of an *event type* and its *attributes* are defined based on the event type. Events are actual instances of the event types and have specific values. For example, the event "today at 10 PM a customer named John Doe booked a new order" is an instance of the Order event type.

We distinguish between two types of event: raw events and derived events.

1. *Raw event*: A raw event is an event that is introduced into an event processing system by an event producer. The definition of a raw event relates only to its source and not to its structure; a raw event may or may not be composed of other events.
2. *Derived event*: A derived event is an event that is generated as a result of event processing that takes place inside an event processing system. When a derived event is emitted outside the event processing system and consumed by a consumer, it is called a *detected situation*.

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

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

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

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 (event payload) and built-in (event header) attributes. When derived events are emitted to consumers they are called *detected situations.*

4.1.1. Proactive attributes

Future events in the FInest CEP engine are expressed by setting the event a future occurrence time with an optional distribution. Its occurrence certainty can be expressed using the *certainty attribute*. The following attributes are added to the event type built-in attributes:

- *Certainty* – a built-in double attribute that stores the certainty of this event. An event has a default certainty value equal to 1, while it can have any value between 0-1.
- *Cost* – the cost of this future event occurrence. The cost is negative if this is an opportunity. This attribute is used by the proactive algorithm to decide on mitigation actions to prevent the occurrence of a predicted event.

- *ExpirationTime* - the time until which the cost and certainty parameters are valid, and until which an activation of proactive action is considered by the proactive algorithm (beyond this time there is no point in changing any course of action).

4.2. Interfaces

The CEP runtime engine has three main interfaces with its environment as depicted in Figure 7.

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

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 sent to the CEP run-time engine. At run-time, the CEP receives incoming events through the input adapters, it processes these incoming events according to the definitions, and sends derived events through the output adapters.

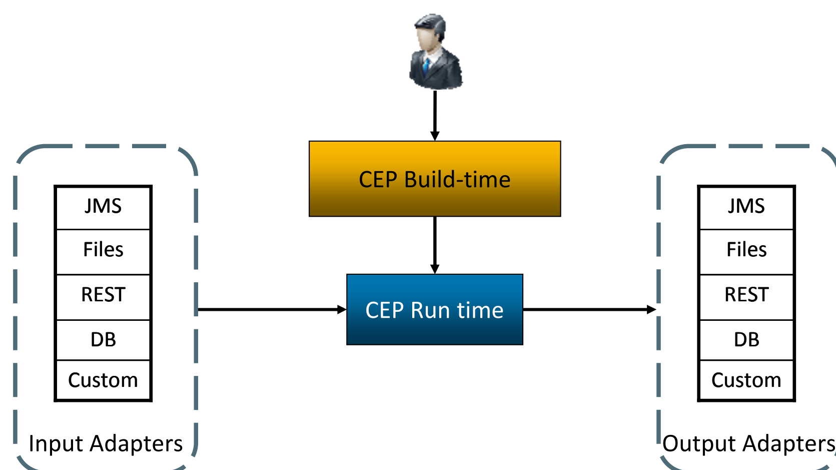

Figure 7: EPM interfaces

4.2.1. Input and output adapters

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

The consumers and their respective output adapters operate in a push mode – each time an event is published by the runtime it is pushed through environment-specific server connectors to the appropriate consumers, represented by their output adapters, which publish the event in the appropriate format to the designated resource.

In FInest, the sources of events can be backend systems (such as AIS or booking systems) or external sensors (e.g., RFID tags) or events originated in FInest during execution and propagated to the CEP module by the BCM. Consumers in FInest are the BCM module (e.g. notification regarding change in shipment status) or the FInest frontend for proactive notifications (see Section 2.3 on proactive event-driven computing). The set of rules applicable to the specific TCP along with corresponding parameters are passed to the CEP engine during execution by the BCM, which in turn gets its information from the TPM module (see Section 4.5 for interactions among the different modules in FInest).

4.2.1.1. EPM RESTful API

As mentioned previously, the CEP has three main interfaces: one for getting input events using input adapters, a second for sending output events using output adapters, and a third for getting application specific definitions. In the case of a RESTful API for the first two interfaces, the CEP engine can activate RESTful services of external applications for receiving input events and for sending output events using its input and output adapters.

The EPM module will build upon FI-WARE CEP GE [4]. The run-time engine supports a RESTful, resource-oriented API accessed via HTTP that uses either XML-based, JSON-based, or tag-delimited format representations for getting input events and for sending derived events. The CEP engine uses a RESTful client input adapter that is capable of accessing a RESTful web service and pulling input events using its GET method and a RESTful client output adapter, which is capable of accessing a RESTful web service and pushing output events using its PUT method.

The CEP GE uses a REST input adapter that activates a REST service as a client, allowing the CEP GE REST input adapter to access the REST web service declared by the event producer and pull events using the GET method. The CEP GE uses a REST output adapter that activates a REST service as a client, allowing the CEP GE REST output adapter to access the REST web service declared by the event consumer and push events using the PUT method.

In terms of representation format, the CEP REST adapter supports XML-based, JSON-based or tag-delimited formats for the received input event. The format is specified in the CEP producer definition (for pulling input events from it) and in the CEP consumer definition (for pushing output events to it) and is passed as part of the requests using the Content-Type header.

Operations

Getting Events API

The CEP activates a REST API for getting incoming events in a pull mode. The CEP plays as a REST client in this API:

Verb	URI example	Description
GET	/application-name/producer	Retrieve all available incoming events

/application-name/producer

Retrieve all available incoming events from a producer.

In tag-delimited format:

```
GET /application-name/producer
Accept: text/plain
```

Sample result:

```
Name=ShipPosition;ShipID=RTX33;Long=46;Lat=55;Speed=4.0;Time=1333033200;
Name=ShipPosition;ShipID=JF166;Long=47;Lat=55;Speed=2.0;Time=1333033260;
```

Sending Events API

The CEP activates a REST API for sending output events (in a push mode). The CEP plays as a REST client in this API:

Verb	URI example	Description
PUT	/application-name/consumer	Send an output event to a consumer

/application-name/consumer

Send an output event to a consumer

In tag-delimited format:

```
PUT /application-name/consumer
Content-type:text/plain

Name=ShipStopped;ShipID=RTX33;Long=46;Lat=55;Speed=4.0;Time=1333033200;
```

4.3. Definition of FInest complex event processing application

The CEP 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.

Currently in FInest, option number 3 is used, i.e., the definitions file is created manually. For Phase II of the project a user interface (option 1) is proposed that will enable creating the rules definitions without programming.

The building blocks of a CEP application are:

- Event type – the events that are expected to be received as input or to be sent as output. An event type definition includes the event name and a list of its attributes.

- Producers – the event sources and the way CEP gets events from those sources.

- Consumers – the event consumers and the way they get derived events from the CEP.

- 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.

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

4.4. FInest EPM instantiation

In FInest a set of rules is defined and provided to the run-time engine during the build-time. At execution time, the instantiation of the relevant EPAs is done by specifying for each Transport Execution Plan (TEP) in a single TCP (Transport Chain Plan) the set of rules from the rules definition that apply to the specific TEPs along with the corresponding parameter values. This set of rules is transferred to the EPM by the BCM, which in turn gets its information from the TPM (see Figure 8). The rules activator component in the EPM is responsible for selection and activation of rules relevant for each TEP based on the information received from BCM (see Section 5.3). An initial list of events and rules for the FISH use case in FInest is provided in we denote by "To be filled in real time by BCM" attributes that are not part of the initial TEP, but their values are obtained as a result of the execution of a TEP at run-time.

Table 2 and Table 3 in Appendix A.

Upon the completion of each TEP within a TCP, the temporal window (temporal context) associated with this TEP is terminated, which causes termination of all EPA instances associated with this context, i.e. all EPA instances related to this TEP.

4.5. Interaction between EPM and other modules in FInest platform

As previously defined in the Month 12 deliverables, the EPM will receive events from the BCM and directly from IoT and backend systems. However, there were some issues to be resolved for M18, such as:

- How the EPM gets the relevant set of rules with appropriate parameter values for the specific TCP/TEP?
- What is the right granularity for execution? A TCP or a TEP?
- What is the information existing in the TCP XML file and what additional (if at all) information is required in order to monitor for detected situations?
- When should the BCM update its information model regarding changes in attributes as the result of events during run-time?
- When should the ECM gets notified of statuses regarding execution of a transport plan?

All these issues were discussed and solved during the collateral meetings (see Figure 6) between WP6 and work packages 5 and 7, especially with WP5, since most of the interaction of the EPM is with the BCM. As a result, refinements of the architecture diagrams (see Section 5) have been made.

Agreements (in no specific order):

- We kept the principle that all information regarding execution will be maintained and taken care by the BCM and the EPM. Therefore, statuses regarding completion (whether successful or not) will be transmitted by the BCM to the other modules at the end of execution.
- At this stage of the project work, a transport replanning will be initiated by the user upon an event notification and will be treated as a new request for planning to the TPM. We will consider more sophisticated mechanisms in the future, in which replanning can be automatically issued by the EPM as a result of certain detected situations (e.g., delays in schedule, change in cargo amounts).
- The TCP/TEP XML should encompass all information required by the EPM. In the FISH use case, attributes required for the events and rules have been mapped to attributes in the XML file. Although most of the information exists, discrepancies have been identified, and it has been decided that some additional information will be included in the XML file.
- Initialization of the rule set for execution by the EPM:
    - The concept is that the TPM has a predefined set of transportation configurations (truck, air, ship...) that have a (more or less) constant set of rules related to them, i.e., "for an overseas shipment you need an import license X time before loading". In that case, once a TCP is constructed with its corresponding TEPs, they induce a list of rules for this transport plan. Parameter values for some of the rules should also be set by domain experts, so that the entire rules configuration is generated by the TPM and is sent to the BCM which, in turn, sends it to the EPM.
    - Based on input from the BCM at the initialization of each TCP, the EPM activates/deactivates rules accordingly.
    - The rules and parameters from the BCM are sent in a predefined structure so that it will be possible to automatically convert them to the required CEP engine format rules (i.e., JSON).
    - The EPM activates relevant rules for the entire TCP and each TEP upon TEP initiation events, and deactivates them upon TEP termination events.
- The BCM will keep up-to-date both its data and process models with regards to events and detected situations (e.g., change in an attribute value, achievement of a milestone, triggering a guard). Furthermore, all event attributes will be stored in the BCM data schema.

Interactions between the EPM and other modules in the FInest platform are depicted in Figure 8.

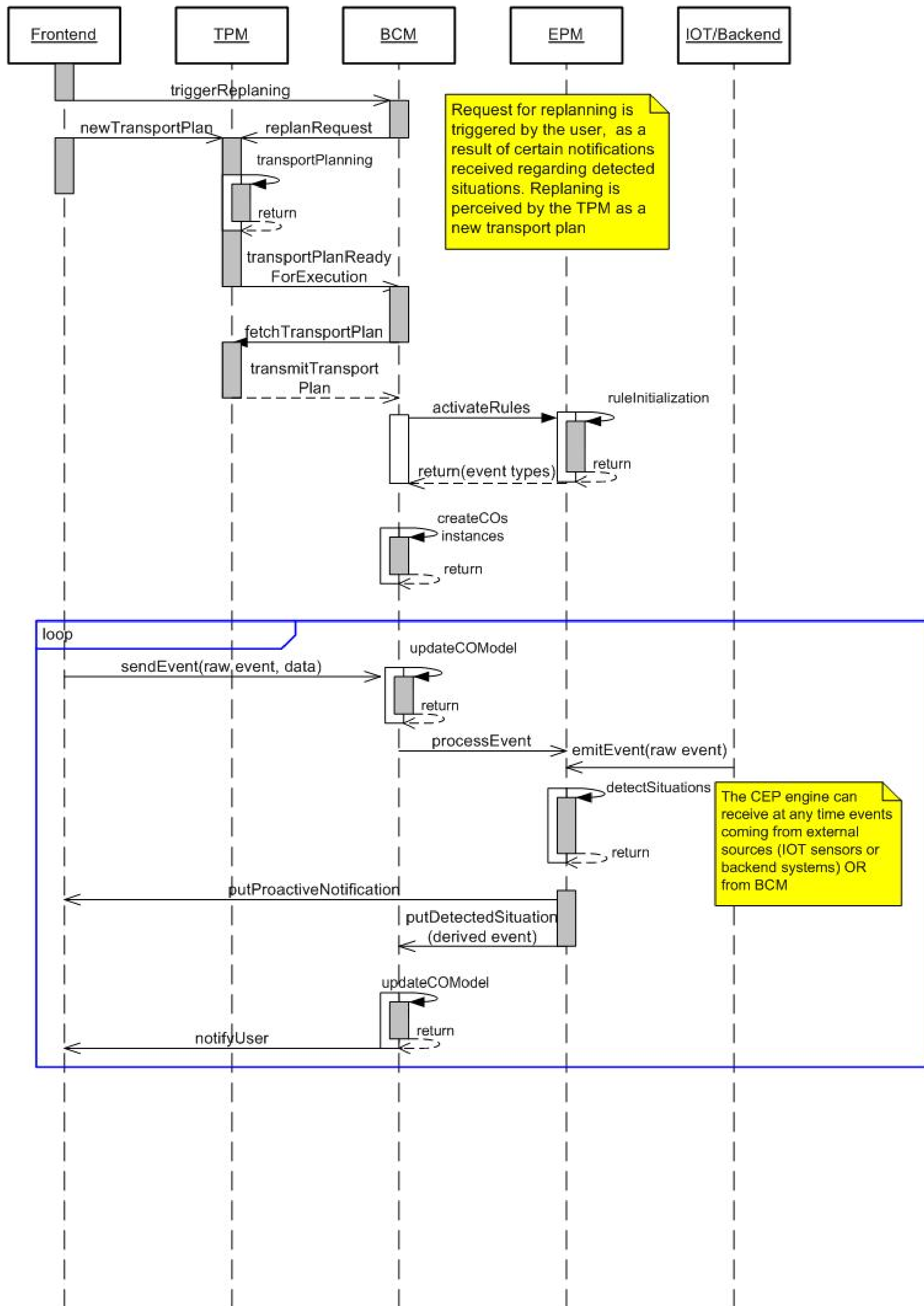

Figure 8: EPM interactions with other components

5. FInest EPM technical specification

In this section of the report we present the proposed technical architecture of the FInest event processing module that comprises the capabilities discussed so far. We start with a high level overview of the EPM module, followed by a more detailed architecture. Conceptually, the FInest CEP engine can be implemented based on the FI-WARE CEP GE, except for the proactive capabilities that differentiate the FInest CEP from currently available complex events engines (see Section 2.3 and Section 5.4). An example of events, rules, and the EPN definition for the FInest FISH use case can be found in Appendix A.

5.1. EPM high level architecture

The EPM high level architecture is depicted in Figure 9. The CEP runtime engine is the heart of the module – it receives events from producers, monitors and checks them for predefined patterns, and produces derived events. When the latter are emitted outside the module to consumers, they are called detected situations. The entire set of rules to monitor is part of the engine set up and they are defined in advance.

Producers: According to the CEP architecture, producers can produce multiple event types and a single event type can be produced by multiple sources. In FInest, we have identified three sources for events: IoT, backend systems, and the BCM.

Consumers: Detected situations are consumed by the BCM, which in turn sends notifications to users via the FInest frontend. Proactive notifications are directly sent to the FInest frontend (consumer) by the EPM since these events don't actually alter the execution of the BCM, but serve as meaningful notifications for a user so they can make better decisions (e.g., trigger replanning).

Events: Raw events are introduced into the module by the producers, while derived events are generated as a result of event processing that takes place inside the EPM. Derived events can either be emitted to consumers (detected situations) or can be input for further processing inside the CEP module by other event processing agents. Detected situations can be sent as notifications to users regarding meaningful situations they have subscribed to.

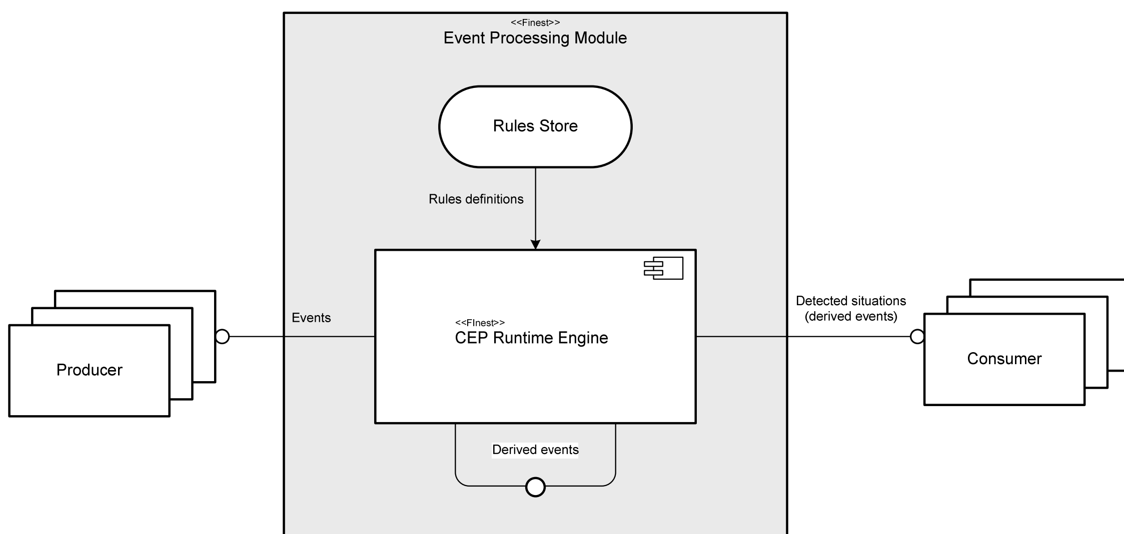

Figure 9: EPM high level architecture

5.2. EPM complex event processing run-time technical specification

Figure 10 shows the internal architecture of the CEP runtime engine.

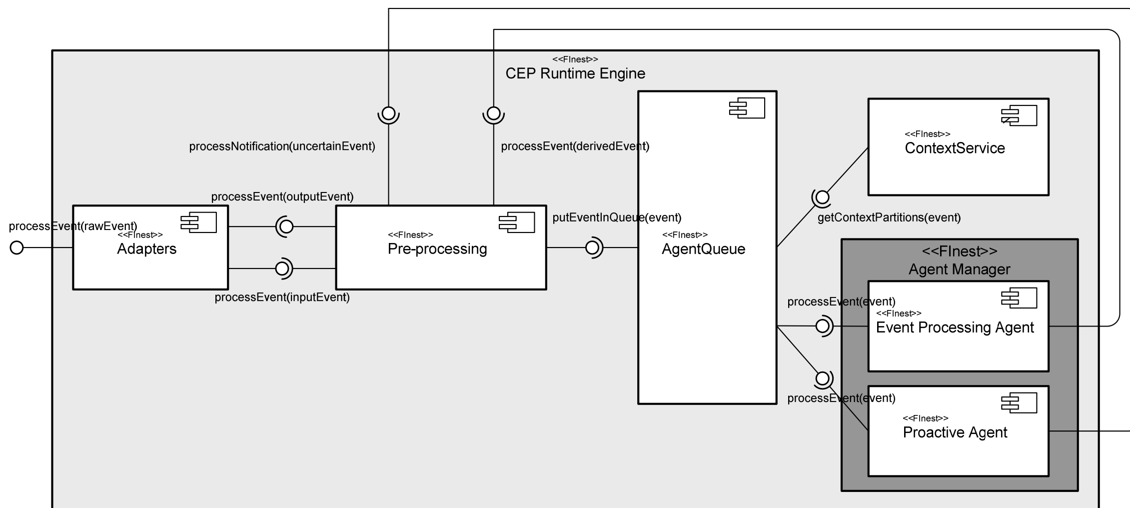

Figure 10: EPM complex event processing run-time architecture

The *adapters component* incorporates both input and output adapter sub-components (see Section 4.2).

- *Input adapters* are responsible for accepting raw events from producers, e.g., from file, JMS (Java Message Service) or REST, converting them into the format that is required by the engine and sending them for preprocessing.

- *Output adapters* receive derived events and proactive notifications and send them to consumers, e.g., to file, JMS, REST.

The *Preprocessing* component is responsible for decisions related to further routing of the accepted event. An event can be routed to a specific agent queue (in case it is an input event) or it can be routed to output adapter (in case it is a derived event) or both. This component preprocesses derived events and proactive notifications and decides whether they will be further processed by the engine (feedback loop) or will be forwarded to output adapters.

The *AgentQueue* component maintains event queues for the different agent types and contexts that these agents are associated with. For each event in a queue, a list of relevant context partitions is retrieved from the *ContextService* (which manages all active context partitions), then the event is sent to event processing/proactive agent instances that are associated with these partitions.

These steps are described in the sequence diagram of Figure 11.

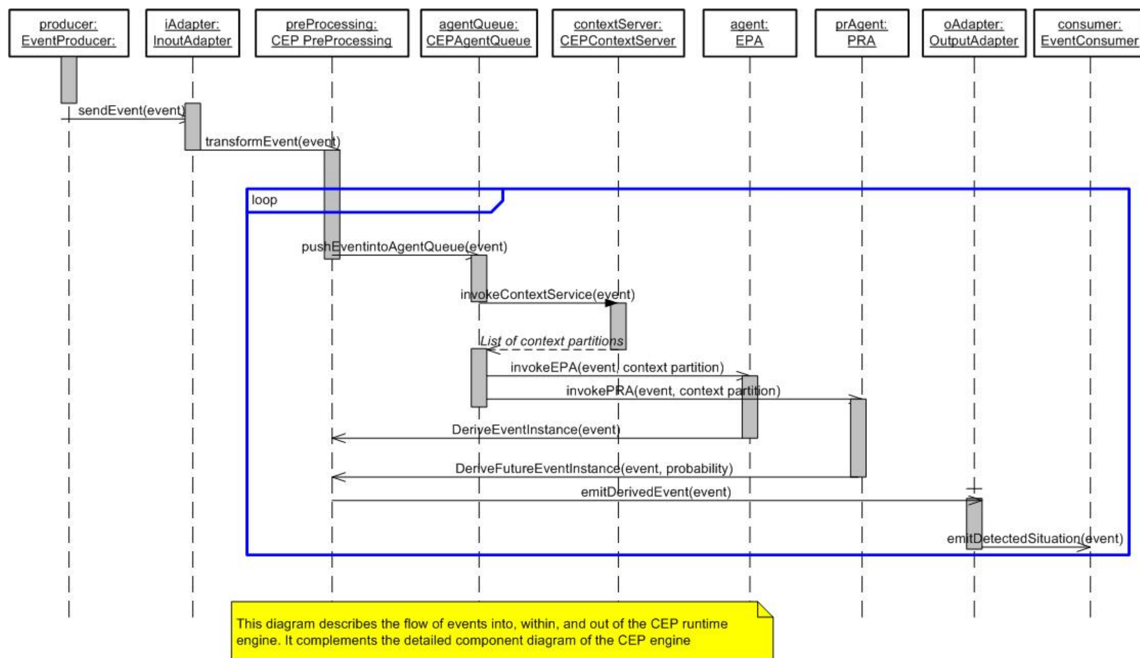

Figure 11: Flow of events into, within, and out of the complex event processing engine

5.3. EPM interfaces with other components

Figure 8 describes how interactions among the modules are carried out by the FInest platform. Figure 12 presents how the FInest architecture, focused on how the EPM realizes these interactions.

The BCM is responsible for sending the entire set of rules to the EPM. This is performed for each TCP upon shipment initiation. Both rules and events are sent by the BCM to the EPM via a REST API. The EPM activates relevant rules for the entire TCP and each TEP upon TEP initiation event, and deactivates them upon a TEP termination event.

The *Rules activator* is responsible for selection and activation of rules relevant for each TEP based on information received from the BCM.

The CEP engine produces detected situations and proactive notifications to the BCM and the FInest frontend correspondingly.

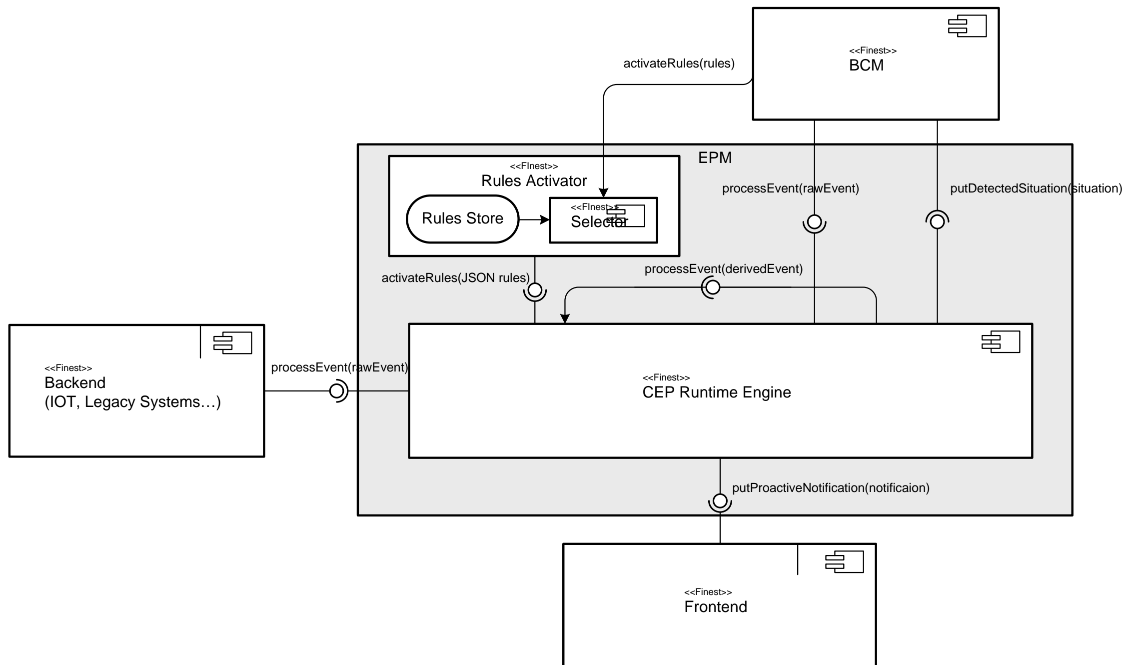

Figure 12: EPM interactions with other components

The steps carried out by the CEP engine upon an arrival of a new set of rules (i.e. initialization of new EPN constructs) is illustrated in Figure 13.

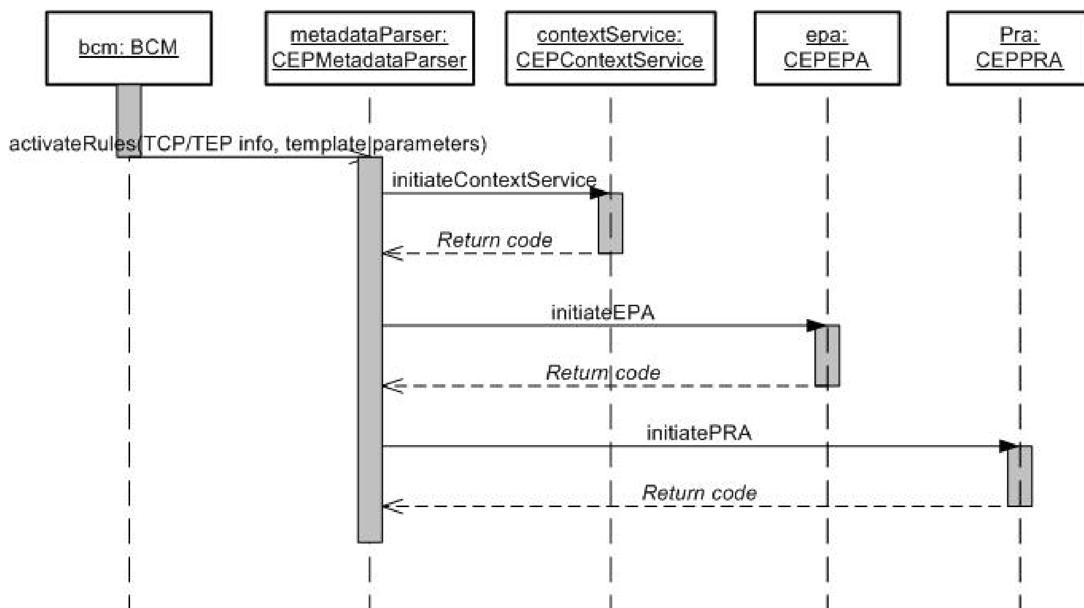

Figure 13: CEP engine set-up

5.4. FInest EPM with FI-WARE GEs

Conceptually, the FInest CEP engine can be implemented based on the FI-WARE CEP GE, except for the proactive capabilities that differentiate the FInest CEP from currently available complex events engines (see Section 2.3).

Figure 14 depicts the FInest EPM with the implementation of FI-WARE GEs. We note that at this stage we envisage the implementation of two FI-WARE GEs: the CEP engine and the PUB/SUB engine (see definitions of these GEs in [4]). The Pub/Sub GE will provide the mechanism to get events from external and IoT systems.

As discussed previously, the CEP GE engine will provide the necessary event processing capabilities except for proactivity. This latter capability will be realized by a specific component that will encompass the specific proactive agents (see Proactive Agent component in the diagram).

The *(FI-WARE) CEP GE* is currently a standalone application (not a web application) that can invoke other components via a REST API for putting produced events; however, other applications can not push events into it in a RESTful manner. The *events repository* component bridges this gap – the BCM pushes events to this repository and they are later pulled by the CEP GE in order to process them.

The *Rules Activator* is also a FInest specific component that is responsible for translating a set of rules from the BCM to a format required by the CEP GE (see also diagram in

Figure 12).

The *Proactive Agent* is a piece of software accountable for processing future events and forecasting detected situations (refer to Section 2.3.3)

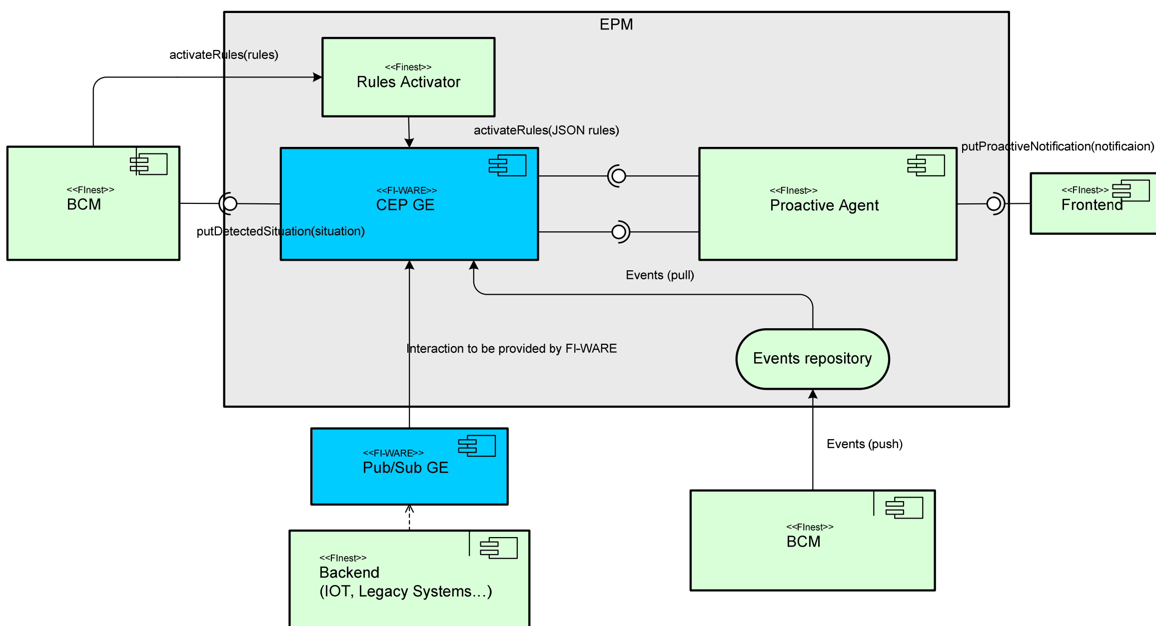

Figure 14: FInest EPM with FI-WARE GEs

5.5. Interfaces data types and definitions

Interactions between various components carry data types as detailed in Figure 15.

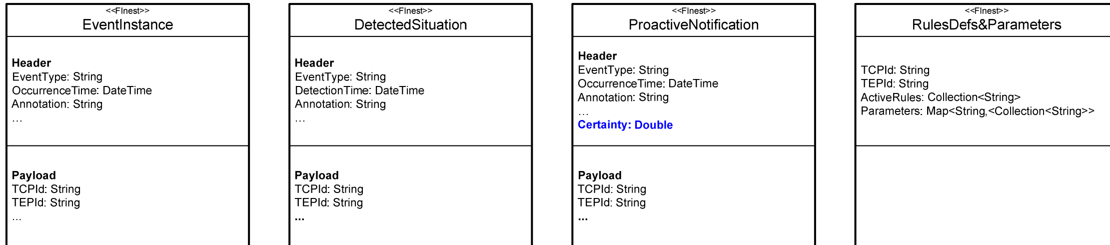

Figure 15: Interfaces and data types

Note that while the EventInstance and DetectedSituation data types have the same structure, the ProactiveNotification also includes the Certainty attribute. ProactiveNotification encloses information regarding an uncertain future event, as described in Section 2.3. The *Certainty* attribute is a value in the [0,1] range representing the probability of this event happening (see Section 4.1.1). The ProactiveNotification is an output of the PRA and the Certainty attribute is calculated by predictive models used by this PRA.

In general, the EPM offers three main services or interfaces (the interfaces with its corresponding methods are shown in Figure 16):

- The *EventSourcing interface*, that allows connecting event sources to the EPM;
- The *RulesActivation interface*, that allows activating pre-defined event types and rules for a specific transport instance; and
- The *EventNotification interface*, that allows being notified about identified detected situations.

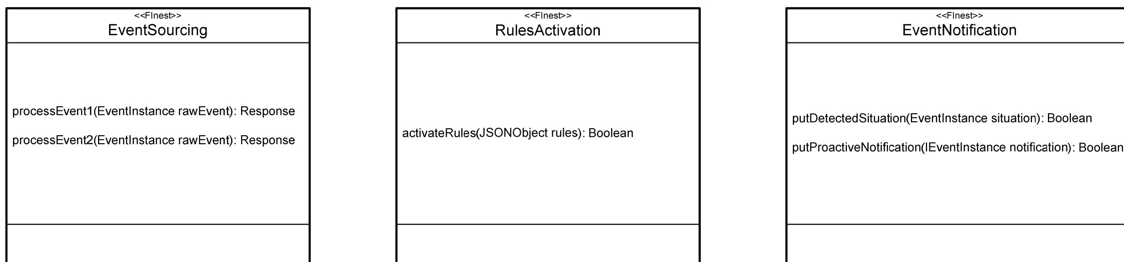

Figure 16: EPM interfaces and methods

The methods/operations grouped by interface are detailed in Table 1 and are shown in the architecture diagrams in Figures 9, 10, 12, and 14. Note that the *source* and *target* columns in the table denote current connections as they appear in the architecture figures. Additional connections (e.g. for phase 2) are possible. For example, proactive notifications could also be connected to other modules in the platform in addition to the frontend.

Table 1: Methods definitions by interfaces

Method name	Source	Target	Description	Data
Interface: EventSourcing				
processEvent1	BCM	EPM	Process raw event (event arriving from BCM), e.g., cancellation of booking, deviation in cargo or schedule, delivery pickup delay etc.	EventInstance - header: EventType, OccurrenceTime, Annotation… - payload: event instance specific attributes
processEvent2	Backend	EPM	Process raw event (event arriving from Backend – IoT and legacy systems), e.g., sensors measurements for container with cargo etc.	EventInstance - header: EventType, OccurrenceTime, Annotation… - payload: event instance specific attributes
Interface: RulesActivation				
activateRules	BCM	EPM	Activate set of rules associated with a specific TCP (and its TEPs). This method is invoked once upon TCP initiation, and it incorporates the entire set of rules to activate + parameters for templates.	RulesDefs&Parameters A list of rules for a specific TCP, including all its TEPs. For templates - parameters values are also provided. This list should be well structured, so that it will be possible to automatically translate rules to (JSON) definition which is an input to CEP engine.
Interface: EventNotification				
putDetectedSituation	EPM	BCM	Put a notification about detected situation to BCM, e.g., a significant deviation in cargo or schedule occurred, a delivery was not picked up as scheduled, import	DetectedSituation - header: EventType, OccurrenceTime, Annotation… - payload:

			license did not arrive for a shipment etc.	Detected situation specific attributes
putProactiveNotification	EPM	Frontend	Put a proactive notification about (uncertain) future event to Frontend, e.g., possible future significant schedule deviation due to high probability to miss a flight.	ProactiveNotification - header: EventType, FututreOccurrenceTime, Certainty, Annotation - payload: Proactive notification specific attributes

6. Summary

The event processing module of FInest enables the real-time monitoring and analysis of event data regarding monitored processes to generate immediate insight and enable instant response to changing conditions.

Deliverable 6.3 details the event processing engine's characteristics, describes the first version of the FInest EPM architecture, its interactions with other core modules of the offered platform, and the potential use of FI-WARE Generic Enablers (GEs). In essence, the FInest CEP engine can be implemented based on the FI-WARE CEP GE, except for proactive capabilities that differentiate the FInest CEP from currently available complex events engines, and position it in the technological and research frontage of event processing systems.

For the next six months, we intend to refine the proposed architecture and the partial EPN for the FISH use case, and to demonstrate EPM capabilities in the module proof-of-concept to be submitted at Month 24.

7. References

[1] A. Paschke and P.Vincent, *Event processing reference architecture*, DEBS 2011 tutorial, available at: http://www.slideshare.net/isvana/epts-debs2011-event-processing-reference-architecture-and-patterns-tutorial-v1-2

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

[3] Y. Engel, O. Etzion, and Z. Feldman, *A Basic Model for Proactive Event-Driven Computing*, Proceedings of the sixth ACM conference on Distributed Event Based Systems (DEBS'12). July 2012

[4] http://forge.fi-ware.eu/plugins/mediawiki/wiki/fiware/index.php/FIWARE.ArchitectureDescription.Data.CEP and

http://forge.fi-ware.eu/plugins/mediawiki/wiki/fiware/index.php/FIWARE.ArchitectureDescription.D

ata.PubSub

[5] Y. Engel and O. Etzion, *Towards proactive event-driven computing*, Proceedings of the fifth ACM conference on Distributed Event Based systems (DEBS'11), DEBS 2011:125-136

[6] M. Fleming, "Business Navigation Systems Combine CEP with BPM", IDC report, August 2009.

[7] W. R. Schulte, "How to choose design patterns for event-processing applications", Gartner report #G00214719, August 2011.

[8] D. Luckham, "The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems", Addison-Wesley, Boston, 2002.

[9] A. Metzger, R. Franklin, and Y. Engel, "Predictive monitoring of heterogeneous service-oriented business networks: The transport and logistics case (Best Paper Award: Service Engineering Innovation & Quality)," in SRII 2012 Global Conference, ser. Conference Publishing Service (CPS), R. Badinelli, F. Bodendorf, S. Towers, S. Singhal, and M. Gupta, Eds. IEEE Computer Society, 2012.

Appendix A – Event processing network for the FISH use case

As previously described, building a CEP application requires the definition of the EPN constructs, namely the event processing network (EPN). As part of the methodology guidelines we followed (see Section 3), we worked on the technical architecture of the EPM module in parallel to validate it through one of the FInest use cases (the FISH use case). As mentioned throughout this report, events producers can be: the BCM, IoT (e.g., TemperatureRead(s) in Table 2), and the FInest backend. All detected situations are sent to the BCM, except proactive rules since these are not part of the actual execution plan. In the case of proactive rules, notifications regarding possible future events are sent to the FInest frontend.

Table 2 shows a partial list of events defined for the FISH use case. Table 3 shows a partial list of rules that have been identified, while Table 4 describes the corresponding event processing network.

Events related to the FISH use case

Note that we denote by "To be filled in real time by BCM" attributes that are not part of the initial TEP, but their values are obtained as a result of the execution of a TEP at run-time.

Table 2: (Partial) List of events for the FISH use case

#	Event name	Attribute name	Attribute in TEP
1.	PickupCargo	TEPId	TransportExecutionPlan. ID
		DeliveryNumber	Consignment.Shipment. ID
		PickupScheduledTime	Consignment. PlannedPickupTransportEvent. Period
		PickupLocation (address)	Consignment. PlannedPickupTransportEvent. Location
		…	
2.	LoadingCargo	TEPId	TransportExecutionPlan. ID
		DeliveryNumber	Consignment.Shipment. ID
		LoadingScheduledTime	Consignment. MainCarriageShipmentStage. LoadingTransportEvent. Period

		LoadingLocation (address)	Consignment. MainCarriageShipmentStage. LoadingTransportEvent. Location
		…	

3.	UnloadingCargo	TEPId	TransportExecutionPlan. ID
		DeliveryNumber	Consignment.Shipment. ID
		UnloadingScheduledTime	Consignment. MainCarriageShipmentStage. DischargeTransportEvent. Period
		UnloadingLocation (address)	Consignment. MainCarriageShipmentStage. DischargeTransportEvent. Location
		…	

4.	CargoArrival	TEPId	TransportExecutionPlan. ID
		DeliveryNumber	Consignment.Shipment. ID
		ArrivalScheduledTime	Consignment. PlannedDeliveryTransportEvent. Period
		ArrivalLocation (address)	Consignment. PlannedDeliveryTransportEvent. Location
		…	

5.	ScheduleDeviation (unplanned)	TEPId	TransportExecutionPlan. ID
		DeliveryNumber	Consignment.Shipment. ID
		ScheduledTime	Consignment. MainCarriageShipmentStage. PlannedDeparture/Arrival/Waypoint

			TransportEvent. Period
		ActualTime	To be filled in real time by BCM
		Deviation (automatic)	
		…	

6.	CargoDeviation (unplanned)	TEPId	TransportExecutionPlan. ID
		DeliveryNumber	Consignment.Shipment. ID
		PlannedCargo	Consignment. TotalTransportHandlingUnitQuantity
		ActualCargo	To be filled in real time by BCM
		Deviation (automatic)	
		…	

7.	ImportLicenseArrival (to the Alesund terminal)	TEPId	TransportExecutionPlan. ID
		DeliveryNumber	Consignment.Shipment. ID
		…	

8.	BookingSchedule Modification	TEPId	TransportExecutionPlan. ID
		DeliveryNumber	Consignment.Shipment. ID
		OriginalTime (e.g., for pickup)	
		NewTime (e.g., for pickup)	
		…	

9.	BookingCargo Modification	TEPId	TransportExecutionPlan.ID
		DeliveryNumber	Consignment.Shipment.ID

		OriginalAmount	
		NewAmount	
		…	

10.	BookingCancellation	DeliveryNumber	Consignment.Shipment. ID
		…	

11.	TemperatureRead (RFID event)	TEPId	TransportExecutionPlan. ID
		DeliveryNumber	Consignment.Shipment. ID
		Timestamp	To be filled in real time by IoT
		Value	To be filled in real time by IoT
		Location	To be filled in real time by IoT
		…	

12.	TEPInitiation	TEPId	TransportExecutionPlan. ID
		DeliveryNumber	Consignment.Shipment. ID
		…	

13.	TEPTermination	TEPId	TransportExecutionPlan. ID
		DeliveryNumber	Consignment.Shipment. ID
		…	

Rule set defined for the FISH use case

In Table 3 we describe rules in a formal and informal way, the type of event processing agent that will be associated with a rule and what are the derived events of each rule. As mentioned in Section 5, the subset of relevant rules will be set up upon the arrival of a new TCP for each TEP. For template rules, parameters should be given actual values. For the types of operators, please refer to Section 2.

The entire rule set is relevant within the time period between TEPInitiation and TEPTermination events. Also, the BookingCancellation event causes termination of execution of all rules related to the relevant shipment.

Table 3: (Partial) Rule set for the FISH use case

Rule name	MissingImportLicense
Rule (informal definition)	Import license was not received at the Alesund terminal X time units before planned loading
Rule definition	Absence(ImportLicenseArrival) X time units before CargoLoadingTime Assumption: CargoLoadingTime stays up-to-date (e.g., is updated according to ScheduleDeviations)
Operator	Absence
Participating events	ImportLicenseArrival
Derived events (notifications)	MissingImportLisence notification
Notification consumer	BCM
Template rule	Yes
Template parameters	X time units before planned loading

Comment: the same rule can be created for LoadingListArrival event.

Rule name	DelayedPickup
Rule (informal definition)	Cargo pickup did not happen as scheduled
Rule definition	Absence(PickupCargo) X time units after CargoPickupTime Assumption: CargoPickupTime stays up-to-date (e.g., is updated according to ScheduleDeviations)
Operator	Absence
Participating events	PickupCargo
Derived events (notifications)	DelayedPickup notification
Notification consumer	BCM

Template rule	Yes
Template parameters	X time units after scheduled pickup

Comment: the same rule can be created for loading, unloading, arrival etc. phases.

Rule name	ScheduleDeviationObserved
Rule (informal definition)	Schedule deviation occurred
Rule definition	Existence(ScheduleDeviation) if (ScheduleDeviation.Deviation > X% OR ScheduleDeviation.Deviation < X%)
Operator	Threshold
Participating events	ScheduleDeviation
Derived events (notifications)	ScheduleDeviation notification
Notification consumer	BCM
Template rule	Yes
Template parameters	Deviation of X% or time units from schedule that should be reported

Rule name	CargoAmountDeviationObserved
Rule (informal definition)	Cargo amount deviation occurred
Rule definition	Filter(CargoDeviation) if (CargoDeviation.Deviation > X% OR CargoDeviation.Deviation < X%)
Operator	Threshold
Participating events	CargoDeviation
Derived events (notifications)	CargoDeviation notification
Notification consumer	BCM
Template rule	Yes

Template parameters	Deviation of X% from cargo amount that should be reported

Rule name	BookingCancellationObserved
Rule (informal definition)	Booking cancellation occurred
Rule definition	Existence(BookingCancellation)
Operator	All
Participating events	BookingCancellation
Derived events (notifications)	BookingCancellation notification
Notification consumer	BCM
Template rule	No
Template parameters	-

Rule name	TemperatureAnomaly
Rule (informal definition)	A temperature read in the cargo storage area exceeds/goes under the allowed range
Rule definition	Filter(TemperatureRead.Value) if (NOT WITHIN [X,Y])
Operator	Threshold
Participating events	TemperatureRead
Derived events (notifications)	TemperatureAnomaly notification
Notification consumer	BCM
Template rule	Yes
Template parameters	Allowed temperature range (from X to Y)

Rule name	TemperatureIncrease

Rule (informal definition)	Steady temperature increase is observed in cargo storage area
Rule definition	Trend(TemperatureRead.Value) is increasing
Operator	Trend
Participating events	TemperatureRead
Derived events (notifications)	TemperatureIncrease notification
Notification consumer	BCM
Template rule	No
Template parameters	

Proactive rule set defined for the FISH use case

Two examples of candidate proactive rules that relate to the entire TCP are given below. Assuming there are various transport modalities for different TEPs in a TCP, we can define the following rules:

For TCP including track, ship and air transportation

In case a ScheduleDeviation occurred in the TEP involving track transportation, produce a notification with probability for significant delay (due to the possibility to miss the flight). This notification may trigger a shipment replanning action.

For TCP including ship, air and track transportation

In case a CargoDeviation occurred at the beginning of the delivery, produce a notification with probability for significant cargo deviation for the TEP involving track transportation (due to the need for additional trucks). This notification may also trigger replanning action.

Event processing network

Table 4 describes the event processing network specified for the FISH use case, which encompasses the rule set given in Table 3. Input and output events are described in we denote by "To be filled in real time by BCM" attributes that are not part of the initial TEP, but their values are obtained as a result of the execution of a TEP at run-time.

Table 2. For an explanation on context and EPA types please refer to Section 2.

Table 4: Event processing network for the FISH use case

Event processing agent name	EPA Type	Input event types	Output event types	Context
MissingImportLicense	Absence	ImportLicenseArrival	MissingImportLicenseNotification	EventInterval (TEPInitiation → X time units before loading)
DelayedPickup	Absence	PickupCargo	DelayedPickupNotifications	EventInterval (TEPInitiation → X time units after planned pickup)
ScheduleDeviationObserved	Threshold (filter)	ScheduleDeviation	ScheduleDeviationNotification	EventInterval (TCPInitiation → TCPTermination)
CargoAmountDeviation	Threshold (filter)	CargoDeviation	CargoDeviationNotification	EventInterval (TCPInitiation → TCPTermination)
BookingCancellationObserved	All (existence)	BookingCancellation	BookingCancellationNotification	EventInterval (TCPInitiation → TCPTermination)
TemperatureAnomaly	Threshold (filter)	TemperatureRead	TemperatureAnomalyNotification	EventInterval (TEPInitiation → TEPTermination)
TemperatureIncrease	Trend	TemperatureRead	TemperatureIncreaseNotification	EventInterval (TEPInitiation → TEPTermination)

Table 5 describes the proactive event processing rules specified for the FISH use case examples given above. Input and output events are described in Table 2. For an explanation on proactive agents please refer to Section 2.3.3

Table 5: Proactive event processing rules for the FISH use case

Proactive Agent Name	Description	Input Event Types	Output Notifications	Context
SignificantScheduleDeviation	In case ScheduleDeviation occurred in a TEP involving track transportation, produce a notification with probability for significant delay (due to the possibility to miss a flight).	ScheduleDeviation	SignificantScheduleDeviationNotification (probability) This notification may trigger a shipment replanning action.	EventInterval (TEPInitiation → TEPTermination)
SignificantCargoDeviation	In case CargoDeviation occurred at the beginning of the delivery, produce a notification with probability for significant cargo deviation for a TEP involving track transportation (due to the need in additional tracks).	CargoDeviation	SignificantCargoDeviationNotification (probability) This notification may also trigger replanning action.	EventInterval (TEPInitiation → TEPTermination)