ICT-2011.8
GET Service Project
2012-318275

Deliverable D6.2: Conceptual architecture specification of information aggregation engine

14 August 2014
Public Document

GET SERVICE

The GET Service project (http://www.getservice-project.eu) has received funding from the European Commission under the 7th Framework Programme (FP7) for Research and Technological Development under grant agreement n°2012-318275.

Project acronym:	GET Service
Project full title:	Service Platform for Green European Transportation
Work package:	6
Document number:	D6.2
Document title:	Conceptual architecture specification of information aggregation engine
Version:	1.0
Delivery date:	28 February 2014 (M17)
Actual publication date:	13 August 2014
Dissemination level:	Public
Nature:	Report
Editor(s) / lead beneficiary:	HPI
Authors(s):	A. Baumgrass, R. Breske, C. Cabanillas, C. Di Ciccio, R. Eid-Sabbagh, M. Hewelt, A. Meyer, A. Rogge-Solti
Reviewer(s):	R. Dijkman, M. van der Velde

History

Version	Changes	Authors
0.1	Created structure	A. Baumgrass
0.2	Updated structure and added short description to sections	A. Baumgrass
0.21	Updated structure based on discussion with M. Hewelt and R. Eid-Sabbagh	A. Baumgrass, M. Hewelt and R. Eid-Sabbagh
0.22	Updated structure based on discussion with C. Cabanillas; added high-level task list	A. Baumgrass, C. Cabanillas
0.23	Discussion of evaluation criteria, merged some of the text from wiki	M. Hewelt
0.24	Added background section; updated structure;	A. Baumgrass
0.25	Added content to Section 4 (design of the service)	A. Baumgrass
0.26	Added remaining CEP Systems descriptions and Evaluation Summary	R. Eid-Sabbagh
0.27	Added content to sections 3.1 (potential terms) and 3.2.	C. Cabanillas
0.28	Added definitions of some language functionalities.	C. Cabanillas
0.29	Added Sections 4.4 (on the cooperation of stream processing and external classifiers).	C. Di Ciccio
0.30	Added content to sections 3.1 and 3.3	M. Hewelt
0.31	CEP Systems Evaluation table completed, Description of Esper edited, and Evaluation summary section added	R. Eid-Sabbagh, M. Hewelt
0.32	Discussions and updates of Section 2 and 4	A. Rogge-Solti, A. Baumgrass
0.33	Added content to section 4	A. Rogge-Solti
0.34	Added content to section 5	R. Breske, A. Baumgrass
0.35	Section 3.1 restructured and definitions added. Section 3.2 extended.	C. Cabanillas
0.36	Aligned tables, structure, and fonts	A. Baumgrass
0.37	Updates on Sections 3.1, 3.2, 3.3 and 3.4.	C. Cabanillas
0.38	More detailed description of the interfaces of the architecture	R. Breske, A. Baumgrass
0.39	Added sections in introduction incl. some content; updated fonts, tables, and some minor corrections throughout the text	A. Baumgrass
0.40	Section 3.3 reworked. Features and adapters list changed into text. Evaluation results aligned to querying languages evaluation in section 3.2.	R. Eid-Sabbagh, M. Hewelt
0.41	Section 3.3 reworked. Slight changes added. Evaluation of querying languages in Section 3.2 aligned to evaluation of CEP systems in 3.3.	R. Eid-Sabbagh, M. Hewelt
0.42	Added content to architecture design	A. Baumgrass
0.43	Updates on Sections 1.2 and 1.3, fixed references in Section	C. Di Ciccio

	4.4	
0.44	Updates on Sections 1.3 and 1.5, and Executive Summary	C. Di Ciccio
0.45	Added conclusion (Section 6)	C. Di Ciccio
0.461	Revised Section 4	A. Baumgrass
0.462	Revised Section 5	A. Baumgrass, A. Meyer
0.47	Spellchecking, references, aligned fonts, figures and more	A. Baumgrass
0.48	Executive Summary and Sections 1 and 2 modified. Comments by Marten van der Velde considered.	C. Cabanillas
0.49	Sections 3.1 and 3.2 improved	C. Cabanillas
0.50	Sections 3.3 improved with regard to reviewers comments	R.-H. Eid-Sabbagh
0.51	Merged all documents and comments from the reviewer	A. Baumgrass
0.52	Executive Summary + Sections 4.4, 6 proof-read, corrected and improved w.r.t. reviewers' comments.	C. Di Ciccio
0.53	Comments for 3.3.-3.4 included	R. Eid-Sabbagh, M. Hewelt
0.54	Updated architecture	R. Breske
0.55	Reworked comments on Section 4.2 and 4.3	A. Rogge-Solti
0.56	Reworked mostly Section 4 and 5; added comments and changed in between where necessary	A. Baumgrass
0.57	Sections 3.1 and 3.2 updated according to reviewers' comments. Minor comments left to be addressed.	C. Cabanillas
0.58	All comments addressed in Section 3.2 and all the references related to query languages added.	C. Cabanillas
0.581	Reworked 4.4	C. Di Ciccio
0.59	Proofreading complete text, removed comments marked as done	M. Hewelt
0.60	Addressed comments from the reviewer till Section 3 (excluded)	A. Baumgrass
0.61	Applied changes to address Marcin's comments in Section 3.2. Consolidation v0.60 and v0.61.	C. Cabanillas
0.62	SiddhiQL added to Section 3.2.	C. Cabanillas
0.63	Table 6 (on recommendation of A. Raptopoulos), referenced in Section 5 , last comments addressed	A. Baumgrass
0.64	Challenges in conclusion (based on A. Raptopoulos comments)	A. Meyer
0.99	Finalized for submission	A. Baumgrass, A. Meyer
1.0	Final version	A. Baumgrass

Executive Summary

The first deliverable of WP6, in D6.1, has established the foundational concepts in the domain of transportation-related events. Their processing is crucial as it bridges the planning and enactment of transportation processes to the analysis and monitoring of the real-world context evolution, in which the GET Service platform operates. In the preceding deliverable, fundamental considerations have been drawn, with respect to the representation of events, their relation to the conceptual objects representing the GET Service domain, and their categorisation according to impact and expectedness. Building upon it, D6.2 shifts the focus, from the concepts that are handled, to the design of the component that is meant to manage them. To this end, the requirements with which such component must comply are delineated. The conceptual architecture is thereby formally defined, taking into account the gathered requirements as design constraints.

The evolution of the context in which the transportation processes are carried out, and with which GET Service platform interacts, can be intercepted through the analysis of the information stemming from various sources. Such sources are heterogeneous as they include, e.g., weather forecast services, traffic information reports, transponders installed on ships and antennas intercepting the position of aeroplanes. Each source of information contributes to depicting the development of the scenario, by offering partial views on it. Furthermore, the originators of such descriptive *streams of events* represent the reported information according to different standards. Therefore, an aggregation service is indispensable to offer a holistic and uniform picture. Deliverable D6.2 deals with this essential functionality.

After an insight on the fundamental concepts of event stream processing, an investigation on the state of the art in that field is reported. In particular, a comparative analysis of the current event processing languages is conducted. Their expressive power is assessed on the basis of the support they offer with respect to the scenarios depicted in deliverable D1.1. Such study is preparatory to the upcoming realisation of the core component, because it drives the future adoption and adaptation of those specific standards that are deemed more applicable to the needs of GET Service. To the same extent, an evaluation of the current major stream processing systems is made, considering proprietary and open source systems, both academic and commercial.

Thereupon, a close examination of the requirements that the event processing component must meet is provided. Among others, these requirements include: the correlation of events to transportation processes; the implementation of a notification mechanism that allows for the selective subscription to specific kinds of events; the usage of static information to enrich and interpret the dynamic flow of events; the integration with automated classifiers, in order to identify the dynamics of potentially anomalous events at run-time, preferably before the transportation process gets irremediably disrupted.

Finally, a detailed description of the architecture is given. The integration of the component with the rest of the GET Service platform is examined. Thereafter, the software interfaces to be exposed are detailed and the functionalities to offer are bound to the realising modules in accordance with WP2.

Contents

List of Figures

List of Tables

List of Terms and Abbreviations

Term	Meaning
CEP	Complex Event Processing
EDIFACT	Electronic Data Interchange For Administration, Commerce, and Transport
EPL	Event Processing Language
EPA	Event Processing Agent
EPN	Event Processing Network
LSP	Logistics Service Provider
TSP	Transportation Service Provider
RPC	Remote Procedure Calls
Notification	Message containing information about the occurrence of an event and its circumstances.

1 Introduction

This deliverable presents the design of the architecture for aggregating sensory events to transportation-related events in the GET Service project. This section provides the background to this deliverable, by presenting the goal of the project as a whole, the goal of the work package of which the deliverable is a part, and the goal of the deliverable itself. Finally, it presents the structure of the remainder of the deliverable.

1.1 Project Goal

The GET Service platform provides transportation planners with the means to plan transportation routes more efficiently and to respond quickly to unexpected events during transportation. To this end, it connects to existing transportation management systems and improves on their performance by enabling sharing of selected information between transportation partners, logistics service providers and authorities. In particular, the GET Service platform consists of components that: (i) enable aggregation of information from the raw data that is shared between partners and transportation information providers; (ii) facilitate planning and re-planning of transportation based on that real-time information; and (iii) facilitate real-time monitoring and control of transportation, as it is being carried out by own resources and partner resources. By providing this functionality, the GET Service platform aims to reduce the number of empty miles that are driven, improve the modal split, and reduce transportation times and slack, as well as response times to unexpected events during transportation. Thus, it reduces CO_2 emissions and improves efficiency.

1.2 Work Package Goal

WP6 aims at providing the foundation for the planning and service composition work packages by providing accurate state information of service composition instances, with the help of event processing (GET Service, 2012). Therefore, its main objectives include the identification, capture, and dissemination of events that can occur during the transportation of goods. This involves operations in charge of processing the raw events captured from devices (e.g. locations as longitude and latitude), from the environment (e.g., weather conditions), or emitted by other systems (e.g. sensors attached to trucks), for example aggregating events to provide them with a specific meaning, and correlating events to specific activities that are carried out in the transportation.

In the GET Service project, WP6 has connections to several work packages, which deliver, specify, or consume transportation-related information and events (cf. (GET Service, 2012)). WP1 is the requirements analysis work package that delivers information about the scenarios and use cases that are relevant for the project. WP2 to WP7 are the development work packages. WP2 defines the structure of the GET Service architecture, including a standardisation of the interfaces (data flows) between the different components. This architecture serves as basis to integrate the implementation of the aggregation service developed in WP6. In WP3, PC-based and mobile device-based user interfaces for end-user services will be developed for both transportation management and route planning that support run-time aggregated transportation planning and control. The interaction with the user that is enabled through these services can also be captured as events and needs to be processed in the aggregation service. WP4 is aimed at defining a domain-specific modelling language that allows the representation of transportation processes and the coordination of all the parties involved in the platform at run-time. As a consequence, events must be associated to the activities defined in the process models so that they can be automatically monitored together with the other elements. Regarding WP5, planning systems require run-time information about transportation, e.g. current train schedules, available assets or capacities, or infrastructures. This information might be captured as events or as persistent information that needs to be aggregated and correlated with events, e.g. to ensure the assets and capacities are actually available as shown in the GET Service platform. Furthermore, re-planning might be required when unexpected events are detected, e.g. a terminal closed due to construction sites for an unlimited period of time. Therefore, providing accurate event definitions as early as possible is of utmost importance to execute proper re-planning operations. Finally, WP7 requires the specification of events and the

aggregation service to support the orchestration of transportation-specific control structures, defined using the concepts developed in WP4. This orchestration should provide novel reconfiguration techniques to support adaptions of running transportation-specific control structures. Therefore, the processing activities mentioned above will be represented in the process models developed in WP4 or required for the purpose of tasks in WP7.

The deployment of the aggregation service requires the other components to specify the structure of events and information that the user interface forwards and that the transport service, the run-time aggregated planning and the orchestration service require.

1.3 Deliverable Goal

Events are known to be detected by different sensors and reported by several information sources. Due to the dynamic nature of the context domain, such information is intrinsically meant to change over time. Therefore, the GET Service core module that is in charge of extracting relevant information on the current development of transportation processes, deals with concurrent event streams stemming from various originators. The information coming from the collection and comparison of the events in the flow of updates has to be interpreted, in order to detect and possibly foresee the development of the transportation process, given the history of its enactment and the context it is carried out within.

Deliverable D6.1 (Baumgrass, Cabanillas, Di Ciccio, Meyer, & Schmiele, 2013) defined *events* in the context and scope of GET Service, i.e., considering only *transportation-related events*. Their relation with concepts concerning the development of a transportation process has been modelled as taxonomy. On the ground of this conceptual model, a machine-readable format has been proposed to represent events (Event Objects). Events have also been categorised into the classes of expected and unexpected events.

D6.2 aims at defining the architecture of the information aggregation and provisioning engine, in the context of the GET Service platform. Its relevance lies in the fact that it determines the design choices of one of the core components of the project, which will affect the following steps of implementation and reverberate on the inter-component communication means. The managed data is defined according to the conceptual foundations of D6.1 (Baumgrass, Cabanillas, Di Ciccio, Meyer, & Schmiele, 2013). Basing upon this deliverable, deliverable D6.3 will present the implementation of the architecture. To this end, a comparative analysis of the state of the art in stream processing languages and systems is also provided in D6.2, as a basis for the future improvement and integration within GET Service platform. Afterwards, D6.4 will depict the extension of the prototype able to (semi-)automatically support the scenarios introduced in (Treitl, et al., GET Service Project – Deliverable 1.1: Use Cases, Success Criteria and Usage Scenarios, 2013). Furthermore, the examples given in this document take inspiration from the use cases that deliverable D1.1 describes in detail. Deliverable D2.1 (van der Velde, Rook , Saraber , Grefen, & Ernst, 2013) serves as the basis for the definition of the interfaces offered from the information aggregation and provision module. In this regard, its relevance for the topics covered in this deliverable resides also in the investigation conducted on the messaging standard for logistics.

1.4 Deliverable Structure

The remainder of this deliverable is structured as follows. Section 2 presents the background on event processing. In particular, Section 2.1 reintroduces the fundamental concepts of event processing networks (EPN), processing agents (EPA), source, consumer, object, and channel, already described in D6.1 (Baumgrass, Cabanillas, Di Ciccio, Meyer, & Schmiele, 2013) but reported here for the sake of clarity. Section 2.2 explains how such concepts come into play in the context of event processing. Furthermore, it outlines how aggregation and correlation patterns contribute to the gathering of knowledge regarding the evolution of transportation processes, out of event streams. Section 3 reports the results of a comparative analysis of current state-of-the-art event processing languages and systems. Specifically, Section 3.1 provides formal definitions for the fundamental terms which the further explanation are based on, such as event stream, event

query and event query language. Section 3.2 proposes a classification of event processing languages. In particular, a comparison is performed on the basis of their expressive power and, more specifically, their support for queries related to transportation-related events in the context of GET Service. The queries considered to this end are derived from the usage scenarios illustrated in (Treitl, et al., GET Service Project – Deliverable 1.1: Use Cases, Success Criteria and Usage Scenarios, 2013). Section 3.3 focuses on a comparative evaluation of event processing systems, i.e., software tools allowing for the processing of event queries. The analysis includes both open source and commercial tools. Then, Section 4 delves into the details of the functionalities that the information aggregation services must offer in the GET Service platform. The discussion is promoted to Section 5, where the architecture of the component offering those services is detailed, in conformance with the aforementioned criteria. Section 6 concludes this deliverable.

2 Background

This section summarizes the background on event processing, as well as the requirements gathered in previous deliverables that are necessary to design the event aggregation architecture in this deliverable.

2.1 Event Processing Infrastructure

Events that are of importance for tracing how a specific transportation process is executed, for coordinating the different parties involved and for making appropriate decisions in relation to re-planning and rescheduling are denoted as *transportation-related events* (Baumgrass, Cabanillas, Di Ciccio, Meyer, & Schmiele, 2013). Typically, events are produced and collected by different kind of systems spanning an *event processing network* (EPN) in which *event processing agents* (EPA) are linked by *event channels* to exchange events (Niblett & Etzion, 2011). Each EPA may act as an *event consumer* to receive *event objects*, and as an *event source* in case it observes *events* and publishes them in a machine-readable form as *event objects*. In this way, an EPA reacts to its input by processing events and outputs events that can be fed to other EPAs over event channels (Luckham, 2002). Figure 1 summarises these interactions and relations, which are also detailed in D6.1, cf. (Baumgrass, Cabanillas, Di Ciccio, Meyer, & Schmiele, 2013).

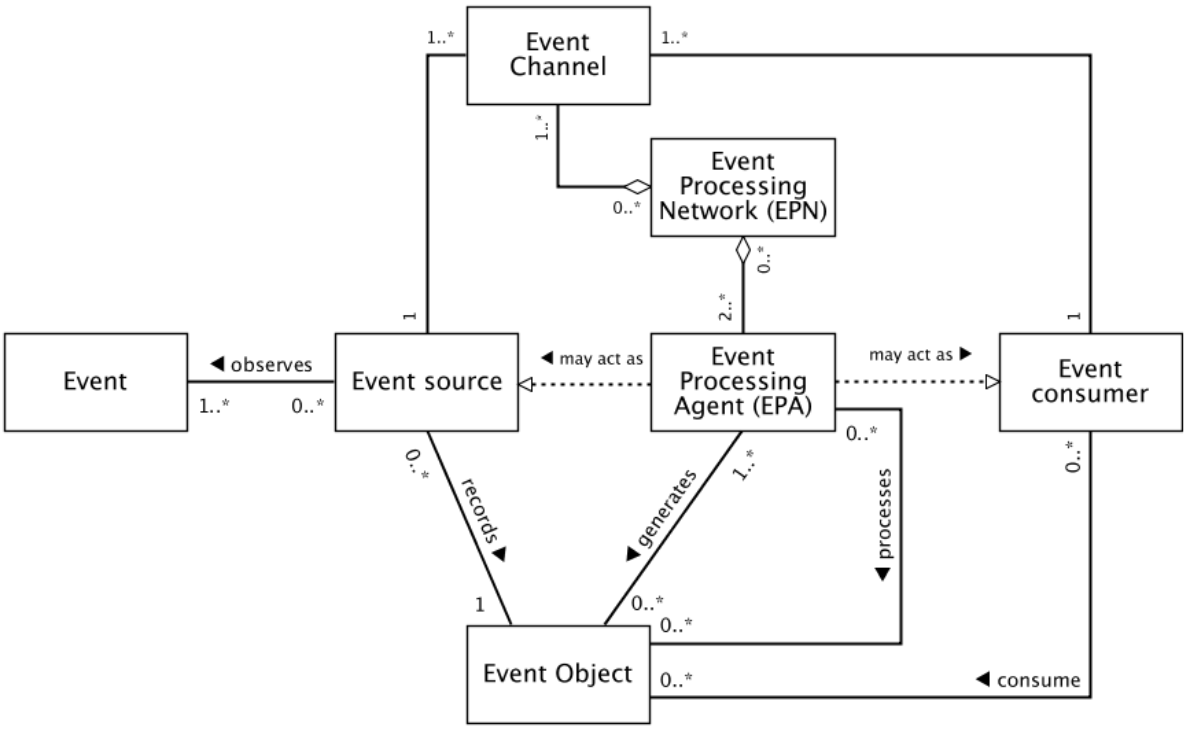

Figure 1 Event Processing Infrastructure

In the context of GET Service, the GET Service Platform should act as an event consumer to gather events from several event sources (e.g. driver's mobile devices and weather stations) and process them to generate transportation-related events, which might be provided to several consumers, e.g. logistic service providers (Baumgrass, Cabanillas, Di Ciccio, Meyer, & Schmiele, 2013).

2.2 Event Processing Mechanism

In accordance with (Treitl, et al., GET Service Project – Deliverable 1.2: Requirements analysis, 2013), it is of high importance for GET Service that each event related to the transportation of goods and the corresponding transportation plan can be observed, processed to derive transportation-

related information, and published to all interested consumers. For this purpose, this deliverable delves into the mechanisms of event processing. Event processing is performing computing operations on events, including reading, creating, transforming, or discarding (Niblett and Etzion 2011). An EPA carries out these operations.

First, an EPA contains an event adapter, which transforms events into event objects (Luckham, 2002), e.g. for importing weather forecasts in XML format. The EPA receives the events from an event channel as input in order to process them. In the GET Service project, adapters are used to identify event objects published by different event sources, e.g. as input in the form of XML, EDIFACT, CSV, MXML, or XES.

Second, events are related to each other according to event relationships. Typically, events are related by time, causality, aggregation (Luckham, 2002), or correlation (Rozsnyai, Slominski and Lakshmanan 2011).

1. Time – relationship that orders events. For example, in Table 1 we can identify that event `Goods loaded` happened before event `Container mounted` given by the occurrence time of the events.
2. Causality – dependence relationship, e.g. event `Container mounted` caused event `Goods loaded`, or the events are independent if neither one caused the other. Furthermore, events can be related by a common cause.
3. Aggregation – abstraction relationship that signifies that an event consists of a set of events, e.g., `Truck ready for transportation` is an aggregation of events `Goods loaded` and `Container mounted`.
4. Correlation – relationship that links event objects based on their values, e.g. `Container mounted` and `Goods loaded` can be correlated to each other in case they happen at the same geographical location or are related to the same truck in a given time window. Events may also be correlated with persistent information, e.g. `Truck ready for transportation` is correlated with a certain *transportation order* and may also contain information about this order (cf. glossary in (Treitl, et al., GET Service Project – Deliverable 1.1: Use Cases, Success Criteria and Usage Scenarios, 2013)).

Table 1 shows an excerpt of how the event objects for the aforementioned events look like and their impact for the target, according to the event properties and concepts defined in D6.1 (Baumgrass, Cabanillas, Di Ciccio, Meyer, & Schmiele, 2013). As a reminder, the *originators* are the entities, which raised the events, and the *targets* are the elements that can be involved in the event. If an event has the GET Service platform as originator this means that the event has been created by the system, probably as a result of previous event processing.

Event patterns are used to specify these relationships and identify them in an event stream considered by an event processing system. For example, only if `Goods loaded` has happened before `Goods unloaded` in a certain time window and both event objects refer to the same truck, they are related by a correlation relationship and can be aggregated to an event `Goods transported`. In this way, a correlation is specified through defining correlation attributes between event object types. Furthermore, event aggregation pattern can be used for recognizing or detecting a significant group of events from among a set of events, and creating a single event that summarizes in its data their significance. An overview of available event processing systems and their integrated EPLs as well as the capabilities of EPLs is provided in Section 3.

Third, event patterns are also used to forward events to interested consumers. For this purpose, an event consumer subscribes to an event processing system with a defined event pattern. Events that match this pattern are sent as notifications over event channels to the consumer. A notification contains data describing an event and may additionally carry information describing the circumstances of the event. In (Mühl, Fiege, & Pietzuch, 2010), the event processing infrastructure only represents the theoretical components and disregards the issues to be dealt with when implementing this infrastructure in practice, e.g. access control or messaging formats.

Table 1 Event examples to identify relationships among them.

	Goods loaded	Container mounted	Truck ready for transportation[1]
Description	New goods are loaded in Container1.	Container1 is mounted onto Truck1.	Truck1 is ready to start its transportation.
Occurrence time	January 2nd 2014 07:30	January 1st 2014 16:30	January 2nd 2014 07:30
Occurrence location	Harbour of Rotterdam	Harbour of Rotterdam	Port of Rotterdam
Originator(s)	Container1	Truck1, Container1	GET Service Platform
Impact	Truck may start the transportation.	Goods can be loaded in the container.	Truck may start driving and transport the loaded goods to their destination.
Target(s)	Container1	Truck1	TransportationOrder1

In the context of the GET Service project, and specifically in WP6, we aim to use the transportation context to identify events from several event sources, process them into transportation-related events, and forward them to interested parties. For this purpose, this deliverable specifies the architecture, components, functions, and interfaces, of a system for conducting event processing in logistics.

[1] For the sake of simplicity, the example is kept simple and abstracted from the real-world. For example, in order to have a truck ready for transportation other events might be relevant as well (e.g. documentation on board).

3 Classification of Event Processing Systems and Languages

This chapter presents a survey of complex event processing (CEP) systems and languages that HPI and WU conducted as part of WP6.

3.1 Terminology

As basic terms required to understand the functionalities of CEP were already introduced in Chapter 2, this section focuses on terms pertaining to event processing systems and languages.

Definition 1 (Event Processing System). An *event processing system* is a software component that is the endpoint of input streams to continuously process them, according to rules set by event queries formalized in an event query language (Bui, 2009).

Examples are all those systems described in Section 3.3.

A CEP system mainly consists of adaptors, a query editor and a CEP engine. *Adaptors* receive messages from event sources (e.g. Web services, an enterprise service bus, via JMS) and transform them into the internal representation used for events, i.e., event objects (cf. Figure 1). Some systems allow working with instantaneous and temporal events, but in some cases only instantaneous events are considered, e.g. SARI and SCEPter (cf. Table 2).

Definition 2 (Instantaneous and Temporal Events). An *instantaneous event represents* something that happens at a specific point in time. A *temporal event* has a lifespan within some time interval.

An example of instantaneous event is the reception of GPS coordinates from a truck, and examples of a temporal event are construction sites or traffic congestions, which have duration.

Many systems already provide a certain selection of adaptors for common event sources, which only need to be configured, e.g. with the address of a Web service. The OneTick system for example comes with preconfigured adaptors for often used stock tickers. Additionally adaptors also transform events that are to be forwarded to other systems (e.g. ESB, JMS, DB) into the external representation format.

Typically, events do not arrive to the system as individual occurrences from time to time, but on a regular (or continuous) basis in the form of a stream.

Definition 3 (Event Stream). An *event stream* is a collection of events, not necessarily being of a single event type. Event streams can (in theory) grow infinitely large; there is no size restriction (Bui, 2009).

An example is the continuous arrival of GPS coordinates for a road truck-based transportation.

Definition 4 (Event Query). An *event query* is a semantic function that takes input streams as arguments and processes the events in the input streams, that means, possibly creating complex events, sending (received or created) events through output streams, or calling other functions (not necessarily event queries) (Bui, 2009).

Examples are a function that prints a message on the screen whenever it finds a new event in its input stream; as well as another function that sends an alert event whenever it receives three failed login attempt events in a short time span.

The event objects already stored in the system must be then interpreted so that valuable information is obtained from them. Defining queries that allow a proper processing of the event objects is key for this purpose. To do so, the *query editor* provides a textual and/or graphical way to specify queries by relying on a specific *event processing language*, which sets the expressiveness boundaries for query formulation.

Definition 5 (Event Query Language). An *event query language* is a formal language that can be used to specify event queries (Bui, 2009).

Examples are all the languages included in Table 2.

All queries define some *conditions*. If these conditions are fulfilled by an event from the stream this event is said to match the query. A match might include also several events (a pattern). Depending on the engine and the language, *actions* can be performed on the matching events, e.g. aggregating them into one new event. Most languages also allow expressing *operations* on event streams like joining two streams. Different languages are able to express different conditions, actions and operations on event streams and thus, are suitable for specific scenarios but may not be applicable to more complex cases (see Section 3). Queries are registered in the engine and continuously compared against the incoming event streams.

The algorithms behind the event processing are usually not disclosed. TIBCO Business Events[2] and jBoss Drools[3] are known to use a variant of the Rete-Algorithm. Cayuga[4], Esper[5], and Siddhi[6] use variants of non-deterministic finite automata as their foundation. In the following section, we will first present a classification of event query languages and then a summary of event processing systems, the characteristics of which highly depend on the language they use.

3.2 Classification of Event Processing Languages

A set of languages that can be used to query event streams has been evaluated. The evaluation criteria have been obtained by extending the comparison framework described in (Bui, 2009) in two different ways.

On the one hand, the set of languages studied has been extended compared to the languages evaluated in (Bui, 2009), which comprised Continuous Query Language (CQL) (Arasu, Babu, & Widom, 2005), AmiT (Adi & Etzion, 2004), ruleCore (Seiriö & Berndtsson, 2005), SASE+ (Diao, Immerman, & Gyllstrom, 2007), Esper's Event Processing Language (EPL) (Inc., 2008), Cayuga (Brenna, et al., 2007), Drools (JBoss Community), and XChange^{EQ} (Eckert, 2008). The extension has been mainly focused on finding out more recent languages for stream querying (i.e., from 2009 to present), as CEP technologies are in constant evolution at the moment. This resulted in ten more languages being added to the list, namely Language-Integrated Query (LINQ) (Raizman, Ananthanarayan, Kirilov, Chandramouli, & Mohamed, 2010), StreamSQL (Kersten, 2007), Complex Event Detection and Response (CEDR) (Barga & Caituiro-Monge, 2006), Semantic CEP (SCEP) (Zhou, Simmhan and Prasanna 2012), SARI-SQL (Rozsnyai, Schiefer, & Roth, 2009), EP-SPARQL (Aninic, Fodor, Rudolph, & Stojanovic, 2011), TESLA (Cugola & Margara, 2012), MONINA (Inzinger, Satzger, Hummer, & Dustdar, 2013), Extended Rete (Walzer, Schill, & Löser, 2007), SiddhiQL[7], and

[2] http://www.tibco.com/

[3] https://www.jboss.org/drools/

[4] http://www.cs.cornell.edu/bigreddata/cayuga/

[5] http://esper.codehaus.org/

[6] http://siddhi.sourceforge.net/

[7] https://docs.wso2.org/display/CEP300/Siddhi+Language+Specification

EAExpression (Rozsnyai, Obweger, & Schiefer, 2011). Languages for which poor or inconsistent information was available have been left aside of the study, e.g. BEA (Oracle) or C-SPARQL. BEA is a language mentioned in (Bui, 2009) for which no information could be found. It was also excluded from Bui's evaluation because of being considered to have similar properties to the rest of languages classified in the same language category (see below), i.e., data stream query languages. In the case of C-SPARQL, information about the operators supported and their implementation found in different sources was contradictory, so no trustworthy conclusions could be made. Newer versions of the languages studied in Bui's work have been evaluated, when existing.

On the other hand, it has been analysed which properties a query language needs to have to express queries for the three use cases derived from the usage scenarios described in D1.1, namely (i) automatic prediction of aircraft diversion, (ii) automatic prediction of delays in inland-waterway transportation, and (iii) automatic prediction of delays in road transportation. From this at least six new functionalities were derived that were not considered in the queries defined in (Bui, 2009) but which are relevant in the context of GET Service. More details are provided below. In summary, as a result of the extension of the evaluation framework the criteria for the classification of query languages can be divided as follows:

1) Group: this property refers to the origin of the language, divided into 4 main types, namely:

    a) Data Stream Query Languages (DS) emerged for querying data streams where data are usually events. In most cases, the main functionalities of these languages rely on some relational query language, typically SQL, which is extended to provide features dealing with characteristics present in streams, mainly temporal issues, i.e., the fact that events change over time. When processing event streams, it is very important to notice that certain operators can only be applied to a finite set of events. Therefore, defining windows and/or limiting the scope of those operators are required. This category of languages is described in more detail in (Bui, 2009), and is widely acknowledged in the related literature. The DS query languages considered in the study are LINQ, StreamSQL, CEDR, SCEP, SARI-SQL, EP-SPARQL, CQL, Esper's EPL, Cayuga and SiddhiQL.
    b) Production Rule Languages (PR) do not operate on streams but on so-called *working memories*, mutable sets of objects capable of carrying data, called *facts* (Bui, 2009). The main idea is to define conditions that, when true, fire actions. In the case of applying such conditions over event streams, they act as the queries of a query language, which identify specific *patterns* or features in the stream. Then, the action might modify the stream by adding or removing facts, i.e., events. To this group, also described in (Bui, 2009), belong TESLA, MONINA, Extended Rete, and Drools.
    c) Composition-Operator Languages (CO), also explained in (Bui, 2009), were specifically designed with CEP in mind. The idea behind CO is to compose complex event queries by combining simple event queries (Bui, 2009). Their expressiveness is usually measured in terms of the operators they support. These languages include AMiT, ruleCore, and SASE+. Esper's EPL and Cayuga share some properties with CO languages.
    d) Logic-Based Languages (LO) are based on logical formulas, hence learning how to define queries with them may be tricky for people without some knowledge in logics. XChange^{EQ} is the only language we have identified for this group. In (Bui, 2009), the language was not classified in any of the previous categories, being placed in an "Others" category because of its singleton status.

2) Notation: languages can have a textual and/or a graphical representation.

3) Foundation: this property specifies whether the query language has been defined on the ground of a well-known formalism or another programming or query language, so that certain functionalities are shared with the foundation. This information was not found for every language reviewed.

4) Functionalities: these are the features of the language that are used to assess its expressiveness. The 10 first functionalities described below were introduced in (Bui, 2009) and support all the queries defined in that work, ranging from basic features such as aggregation (covered by most of the languages) to some advanced features such as integration with external data sources. Windows were also defined there, but the evaluation was constrained to tuple and time windows. The remaining functionalities have been included when studying the new query languages and the specific requirements for the three aforementioned use cases. In particular, the set of functionalities considered are the following:
   a) Disjunction: a complex event denoting a disjunction represents the occurrence of one of its members (Bui, 2009).
   b) Negation: it relates to the capability of searching for the non-occurrence of an event. It is usually used together with other operators for the purpose of counting.
   c) Conjunction: a complex event denoting a conjunction represents the happening of all of its members (Bui, 2009).
   d) Data extraction: the extraction and/or typecast of data to assign a value to a variable or to the output of the query.
   e) Integration with external data sources: it is sometimes necessary to compare event data with data from outside, that is, to include non-event data in the queries, such as data from a database (Bui, 2009).
   f) “Group by” aggregation: this feature is related to the grouping of events regarding specific criteria, usually as a way to prepare the outcome of the query.
   g) Aggregation: data can also be processed with functions, such as the average, maximum, minimum of an attribute (Bui, 2009). Notice that these functions do not work well with unbounded streams, as a finite set of events is required.
   h) Event instance selection: this function serves to filter events in an event stream, typically to consider only the events of a certain type, e.g. the last, or the last *n*, or the first (Bui, 2009).
   i) Sequences: a number of events that occur in a specific order or under certain conditions. There are different interpretations and implementations of this functionality. Sometimes, loose order is enough, e.g. an event B must occur eventually after an event A; whereas in other occasions the order is strict, e.g. B must occur directly after A in the event stream.
   j) Event instance consumption: this feature is used to restrict reuse of events for pattern detection by removing an event from an event stream once it has been used in a query.
   k) Renaming: renaming information obtained from events might be required in order to format the output of a query and/or to compare values of events coming from different sources.
   l) Nested queries: it might be required to define complex queries by combining the output of other queries. This feature is fundamental for CO query languages (see above).
   m) Error catching: some languages provide a way to detect problems during the execution of certain operators, which can be returned by the query. Note that it is up to the CEP system to provide exception handling functionalities in such situations.
   n) Parameterisation: in order to generalise the queries, it is convenient to allow defining parameters that will be customised for specific query instances, similarly to languages like SQL.
   o) Window types: in event streams, it is typically necessary to process only a subset of all the events being received. There are different types of windows for that purpose, mainly:
      i) Tuple window: a function that retrieves a specific number of events, e.g. the last 10 events in an event stream.
      ii) Time window: a function that retrieves events that occurred (or that were introduced in the system) within a specific time frame, e.g. the events captured in the last 20 seconds.
      iii) Sliding window: a window that continuously moves over time.
      iv) Field-value-based window: a window whose size and advance is determined by a tuple field (Kersten, 2007).
      v) Batch window: a window that posts collected events as a batch to listeners. Batch windows can be subdivided in tuple, time, or field-value-based windows.

p) Baskets: the subsets of events that need to be jointly processed for a specific query can be kept in main memory, thus making their processing faster. Such a subset of events is called basket, and events can usually be inserted into and removed from it while executing a query.
q) Geospatial information: in logistics scenarios it is typically necessary to track the positions of domain objects (e.g., transportation vehicles), and to properly locate occurring events. Specific operators and functions are required to support the analysis of geographical information in order to estimate, e.g., the proximity of two points in space, the geographical containment of points in areas, etcetera.

Table 2 shows the classification of the languages according to points 1 to 4, together with some examples of systems where these languages are used, in case evidence was found, e.g., they were explicitly mentioned in the scientific papers describing the languages, or some Web site was referenced where such information could be found. The information included in the table comes from scientific papers, manuals, tutorials and/or online information. The fields for which information could not be found have been left blank. The acronyms shown in the Group column correspond to the acronyms specified when defining the languages groups above.

Table 2 Classification of query languages

Language	Group	Notation	Foundation	Systems using it
LINQ	DS, LB	Textual	CEDR temporal algebra / Implemented in XML	Microsoft StreamInsight
StreamSQL	DS	Textual	SQL	Data Cell, TIBCO StreamBase
TESLA	PR	Textual		T-Rex
EAExpression		Textual		
CEDR	DS	Textual	CEDR temporal algebra	
SCEP	DS	Textual	SPARQL for semantics subpatterns	SCEPter
MONINA	PR	Textual		Indenica
Extended Rete	PR	Graphical	Rete Algorithm	
SARI-SQL	DS	Textual	SQL	SARI
EP-SPARQL	DS	Textual	SQL/RDF	
SiddhiQL	DS	Textual	SQL	WS02
CQL	DS	Textual		STREAM
AMiT	CO	Textual		IBM Active Middleware Technology
ruleCore	CO	Textual	ECA rules	

Language	Group	Notation	Foundation	Systems using it
SASE+	CO	Textual		SASE
Esper's EPL	DS, CO	Textual	SQL	Esper, Glutter
Cayuga	DS, CO	Textual		
Drools	PR	Textual		JBoss
XChange^{EQ}	LB	Textual		XChange

Table 3 shows which of the functionalities described above are supported by each query language. The information was gathered from the same type of sources as for Table 2. In addition, the languages that offered an open-source implementation and were kept up-to-date were also tested, e.g. Esper's EPL and EP-SPARQL. In the table, √ means that the feature is supported with the language operators, √* means that it is partially supported, ~ means that it is somehow supported by a combination of other operators, X means it is not supported at all, and blanks represents the information for which no evidence could be found. Some remarks about the content of the table are:

- All the languages are declarative.
- TESLA and CEDR implement not only the event instance consumption policy but also others.
- TESLA and CEDR refer to nested queries as query composition.
- LINQ defines sliding windows as snapshot windows, time windows as hopping windows, and tuple windows as count-based windows. In addition, it introduces tumbling windows but the definition is not clear in the documentation of the language.
- In the case of XChange^{EQ}, windows are not built-in but can be defined by combining different operators of the language.

For the purpose of the project, the languages that are used in open source systems or that are in a stable version and can be used as query language in any system are of interest. Hence, the languages proposed in scientific papers that are under evolution and the languages defined for commercial systems (e.g. LINQ, StreamSQL, Drools, AMiT) are disregarded. That reduces the set of interesting languages drastically to only Esper's EPL, CQL, SiddhiQL and EP-SPARQL. It is remarkable though that no language supports the use of geospatial information (cf. Table 3).

In Bui's survey (Bui, 2009) CQL and Esper's EPL were named to be the most expressive languages as they cover the basic functionalities of stream query languages. However, Esper's EPL expressiveness outperforms CQL as well as SiddhiQL characteristics, as can be seen in Table 3. Hence, Esper's EPL is slightly more powerful than the other two languages. The other language that competes with Esper's EPL in expressiveness is EP-SPARQL. They have different features due to their different nature, though. EP-SPARQL derives from SPARQL[8], a language for querying RDF triples that provide specific meaning about facts. Esper's EPL was explicitly developed for CEP as query language in the Esper system. Nonetheless, they share language category (in particular DSQLs), since both rely on relational database management and specifically use SQL as database query language.

[8] http://www.w3.org/TR/rdf-sparql-query/

Table 3 Functionalities provided by query languages

Language	Functionalities																
	Disjunction	Negation	Conjunction	Data Extraction	Integration with external data sources	Group By	Aggregation	Event instance selection	Sequences	Event instance consumption	Renaming	Nested queries	Error catching	Parameterisation	Window types	Baskets	Geospatial Information
LINQ	√	~	√	√	√	√	√	√	X		√	√	X	√	Sliding, time, tumbling, tuple	√	X
StreamSQL	√	√	√	√	√	√	√	√	√	√	√	√	√	√	Sliding, time, tuple, field-value-based	√	X
TESLA	X	√	X	√	X	X	√	√	√	√	X	√	X	√	Tuple, time, field-value-based	X	X
EAExpression	X	X	X	√	X		√	√	√	X		~	X	X	Tuple	X	X
CEDR		√		√				√	√	√		√	X	√	Time, sliding	√	X
SCEP	X	X	X	√		~	√	√		X	X	X	X	√	Time, sliding, data, batch	~	X
MONINA	X	X	X	X	X	X	√	√	X	X	√	~	X	√	Batch, time	~	X
Extended Rete	√	√	√		X	X	√	√	√	X	X	X	X	X	Time, sliding	X	X
SARI-SQL	√	~	√	√	X	X	√	√	X	X	X	√	X	√		X	X
EP-SPARQL	√	~	√	√	√	√	√	√	√	X	√	√	√	√	Tuple, time	√	X
SiddhiQL	X	X	√*	√	X	√	√	√	√		√	X	X	√	Sliding, time, tuple, batch	√*	X
CQL	√	√	~	√	√	√	√	~	~	~	√	X	X	X	Tuple, time	X	X
AmiT	√	√	~	X	X	√		√	√	√			X	X	Time	X	X
ruleCore	√	√	√*	X	X	√*		~	√*	√*			X	X	Time, sliding	X	X
SASE+	√	√	~		√	√	√		√	~			X	X	Tuple, time	X	X
Esper's EPL	√	√	√	√	√	√	√	√	√	~	√	√	√	√	Tuple, time, batch	~	X
Cayuga	√	√	X	X	√	√	√	X		~			X		Tuple, time	X	X

Language	Functionalities															
	Disjunction	Negation	Conjunction	Data Extraction	Integration with external data sources	Group By	Aggregation	Event instance selection	Sequences	Event instance consumption	Renaming	Nested queries	Error catching	Parameterisation	Window types	Baskets	Geospatial Information
Drools	√	√	~	~	~	√	√	√	√	√	~		X		Time	X	X
XChange^{EQ}	√	√	√	√*	~	√	√	X	√*	X	√*		X		Tuple, time	X	X

Esper's EPL covers all the basic functionalities of stream query languages, namely columns 1 to 11 describing functionalities in Table 3, and also most of the advanced features such as nested queries, which are required in some of the queries for the three uses cases described in the scope of the GET Service. However, baskets are supported in Esper only by means of the concept of batch window (thus in an indirect way), and the semantical interpretation of information is not covered, which in the context at hand might be necessary, e.g., to deal with the concept of physical proximity. On the contrary, EP-SPARQL supports these two aspects, but event instance consumption is not supported.

All in all, Esper's EPL is most likely suitable for the use cases considered in GET Service. However, either it would need to be combined with EP-SPARQL queries when semantics are required, or it should be extended to provide such functionalities by itself. Esper's EPL is extensible and, thus, the latter option would be feasible. In both cases, the handling of geospatial information is an issue that needs to be solved.

3.3 Classification of Event Processing Systems

To evaluate existing complex event processing (CEP) systems and frameworks we consider a set of criteria, which are described in the following.

CEP systems are used in many different application domains for several application scenarios. Also the deployment scenarios of the evaluated systems vary. Some evaluated systems are to be used as standalone systems, while others are optimized for grid or cloud scenarios, providing resilience and high availability.

First we distinguish between open source (OSS) and proprietary software. While proprietary software systems might be the right choice for companies, which only want to use the software, open source CEP engines make for easier integration into self-developed systems like the GET platform. As the GET project requires the integration of CEP with process management, it is likely that the used CEP system will have to be adapted. However, changes to proprietary systems are seldom realizable. In the case of OSS the specific license, under which it is published, is important to determine, whether the system can be modified and used as part of a commercial platform. While support in most cases is better for proprietary software, the community of an OSS can make up for the lack of official support. Many OSS projects have an active community of developers who answer questions and even incorporate feature requests. We found the overall quality of documentation to be high, both for OSS and proprietary offerings. In many cases, software products are developed as open source projects and freely available, however, there are many companies that provide extra support, extra services, maintenance, adaption, or SLA's for the product as a service.

A related distinction can be made between academic and commercial software systems. Academic CEP systems often explore and implement innovative concepts. On the other hand they often are unstable, lacking documentation and support, and their lifespan is short.

An important aspect of our evaluation is the language that is used to express queries. Most of the considered CEP systems use an extension of database query language SQL, adapted to potentially infinite event streams. This requires windowing mechanism to formulate queries over a subset of events. In our evaluation of CEP systems we found that all products support the common functionalities of filtering event streams based on conditions, joining of event streams, computation of aggregate values like sum or average, matching of patterns, and the definition of windows. Languages differ only subtly. A detailed evaluation of event query languages can be found in Section 3.2.

Another criterion is the possibility and ease with which to integrate the CEP engine with event sources. All evaluated CEP systems follow the concept of adapters, which transform incoming messages into the internal event representation. Some systems come equipped with pre-defined adaptors, some offer wizards for the definition of custom adaptors. Commercial CEP systems are integrated with the other enterprise applications of this vendor. Commonly pre-defined adaptors include the Java messaging system (JMS), JDBC-compliant databases, and web services (either SOAP or REST over JSON). The GET project has not formulated requirements for the communication infrastructure of the cooperating components yet; therefore we cannot check whether it is supported. Each system references the languages it uses from Section 3.2.

As GET Service aims at integrating CEP with BPM, we try to evaluate to which amount the reviewed systems support business processes. Unfortunately little information on this topic can be found in the documentation of the considered systems. Esper mentions in its documentation, that process analysis and monitoring are application scenarios for CEP but does not address this topic further.

3.3.1 Overview of considered CEP systems

For the review a list of 17 CEP systems was compiled, which is displayed in Table 4. Those 17 systems were reviewed cursorily and 10 systems were selected for a more detailed evaluation. Please, notice that some of the languages mentioned in Table 2 are not present in our evaluation, as we did not find any specific information about them further than the scientific papers where their query languages were described. On the other hand, some systems are included here, the language of which was not considered in Table 2. This is due to the fact that no academic papers were available on their query language and hence these languages could not be evaluated according to the standard of Section 3.2.

First, open source and proprietary systems under active development at the time of writing were considered. This excludes Aurora, Medusa and Borealis from our investigation. RuleCore also appears to be inactive.

The Fuseki component of the Apache Jena framework provides a SPARQL-server which allows querying RDF databases. Because Fuseki uses a database, it does not constitute a CEP system as defined above. The focus of Storm and Hadoop lies on distributed computation (MapReduce). Although they might be used to develop CEP applications, they are primarily designed for a different task. The same applies for OneTick, which specifically aims at financial institutions. Hence its provided adapters connect with stock exchange data. Because the focus of the GET project is very different, we excluded Fuseki, Storm, Hadoop, and OneTick from our evaluation.

EasierBSM is an experimental component for the Petals ESB framework. While the Petals framework aims at enterprise application integration, easierBSM deals with the monitoring and enforcement of service level agreements in web service environments. However, complex event processing is only supported indirectly via the monitoring of SLAs and service governance. Hence easierBSM was not analyzed further.

The Event Model (TEM) is currently under development but only in a prototypical stadium. It focuses on easy definition of queries, which is suitable for business people. Due to its early stage we excluded TEM from our survey.

The following sections describe the remaining CEP systems in more detail.

Table 4 Overview of Complex Event Processing Systems

Name	Description	Further information
TIBCO BusinessEvents	Integrated platform with ESB, BAM, CEP; rules engine	http://www.tibco.com/products/event-processing/complex-event-processing/businessevents
EasierBSM	Web service monitoring and SLA enforcement; research prototype	https://research.petalslink.org/display/easierbsm
Apache Jena Fuseki	SPARQL server	http://jena.apache.org/documentation/serving_data
Esper	CEP engine	http://esper.codehaus.org/
Aurora	Data-stream processing engine	http://cs.brown.edu/research/aurora/
Medusa	Extension to Aurora for the distribution of events	http://nms.csail.mit.edu/projects/medusa/
Borealis	Stream Processing Engine	http://cs.brown.edu/research/borealis/public/
RuleCore	CEP Server for real-time detection of complex event patterns from live event streams	http://rulecore.com/
StreamBase	CEP Engine; StreamBase Systems, Inc. has been acquired by TIBCO Software Inc.in June, 2013	http://www.streambase.com/products/streambasecep/
jBoss Drools Fusion	Business Logic Integration Platform with Rule engine, CEP engine, graphical editors for eclipse and jBPM integration	http://www.jboss.org/drools
SAP Sybase Event Stream Processor (ESP)	Platform for developing real-time event based applications	http://www.sybase.de/home
Storm	Free and open source distributed real-time computation system for streams of data	http://storm-project.net/
Oracle Event Processing (OEP)	Building applications to filter, correlate and process events in real-time	http://www.oracle.com/technetwork/middleware/complex-event-processing/overview/index.html
OneTick CEP	Finance sector, market data analysis	http://www.onetick.com/web1/onetick_cep.php
Hadoop	Parallel data processing, MapReduce	http://hadoop.apache.org/
StreamInsight	Effectively analyse large amounts of event data streaming from multiple sources	http://www.microsoft.com/en-us/sqlserver/solutions-technologies/business-intelligence/streaming-data.aspx
The Event Model (TEM)	Integration in WebSphere	http://www.youtube.com/watch?v=9zjy8wngy5Y

3.3.2 TIBCO Business Events

TIBCO Business Events (BE) is a commercial software platform for complex event processing, which can be integrated with other TIBCO applications. Extensive documentation is available, including an excellent "Getting started" guide. At its core BE uses a forward-chaining Rete algorithm to match incoming events against the rule set. The rule editor is implemented as an Eclipse plugin.

Conceptually a BE instance consists of three types of agents: Interference Agents, which perform the pattern matching, Caching Agents, which provide a cache for the inference, and Query Agents,

which perform queries. This allows for scalability, as one BE instance can contain several agents of each type. All logic is realized by rules. A rule has four parts: attributes of the rule e.g. priority, declaration of concepts and events, conditions that need to be fulfilled, and actions that are executed, if the rule matches. In contrast to rules, queries have no action part.

BE distinguishes between concepts, events, and channels. Concepts are data structures that represent existing knowledge to be compared with events. They can be created and/or manipulated as result of rule execution and can have an attached lifecycle. Events are also data structures; however, they live only for a single run of the Rete algorithm (except when the rule explicitly keeps the event). Events are transmitted through channels. A channel is BE's abstraction of communication and can be used either internally or bound to e.g. JMS. Channels know their destination, either source or sink, and can perform pre-processing of incoming messages, e.g. parse a XML document into the BE event representation (and the other way round).

3.3.3 Streambase CEP

Streambase was acquired by TIBCO and it is unclear whether the software will be integrated into TIBCO's other offerings or will continue to exist on its own. Streambase CEP provides Java, C++, and Python APIs to develop CEP applications. The event processing logic can be either defined graphically in an Eclipse-based IDE, the Streambase Studio, or textually in StreamSQL. Unfortunately, the languages are not congruent and some queries can only be expressed in one but not the other language. The IDE provides wizards for new projects and definition of adapters, extended testing and debugging facilities, and graphical query modeling. StreamSQL offers the usual stream operators, i.e. filtering, aggregating, windows, and joining, and allows querying both stored and streaming data. Instead of interpreting the query, it is compiled by the dynamic stream compiler (DSC). According to Streambase this provides high performance and low latency.

Data in a stream is called a tuple consisting of several fields defined by a schema. Processing can add, remove, or modify fields. Tuples are produced by adapters of which several are pre-defined e.g. financial market data streams, JMS messaging systems, JDBC-compliant databases, and Streambase's Chronicle data storage framework.

3.3.4 WSO2 CEP / Siddhi

"The WSO2 CEP Server monitors an SOA environment, and identifies meaningful patterns, relationships and data abstractions among apparently unrelated events, and fires an immediate response enabling enterprises to take immediate action." (WS02 Webpage, http://wso2.com/). The CEP server is a part of the rich WSO2 platform for business middleware, which among others include an ESB, message broker, application server, load balancer, and business activity monitoring. The complete platform is open source and extensive documentation including samples is available.

Event sources for the WSO2 CEP server can be connected via JMS, REST with JSON, SOAP, and email. While adapters are used to connect to sources, event builders are used to map the received events into the uniform WSO2event format. Both adapters and event builders can be configured through the management UI.

Processing of the events is carried out by the Siddhi engine, which is also available stand-alone. Esper and Drools Fusion can be used as alternative engines for the WSO2 server. Siddhi's querying language supports the usual operations: Filtering event streams with conditions, joining event streams, and detecting patterns of events. Windows can be used to limit processing to a certain subset of events. The CEP server is scalable and can use several engines. In this case, Hazelcast can be used as shared memory. The developers claim a higher performance than Esper (see the blogpost http://wso2.com/library/blog-post/2013/08/cep-performance-processing-100k-to-millions-of-events-per-second-using-wso2-complex-event/).

3.3.5 jBoss Drools Fusion

jBoss Drools is a platform for the integration of business logic. It started out as a rule engine, in which rules capture business logic. It consists of the modules Expert (rule engine), Guvenor (governance), Flow (Workflow engine, integrates with jBPM), Fusion (Complex Event Processing) and OptaPlanner (planning and resource allocation engine). Drools is an open source project with a strong community, extensive documentation and is written entirely in Java.

Fusion provides basic CEP functionalities: handling event streams and clouds, detecting, correlating, aggregating and composing / abstracting (produce complex event) events. Support for sliding windows, temporal orders and clocks / calendars exists. Pattern matching is provided by Drools Expert, all features of Fusion can be used in the Flow process module. What is interesting about Drools is that it includes also a workflow engine.

3.3.6 Esper

Esper is an open source component for event processing and event stream analysis. The Esper core module is a CEP engine that has been developed in Java but also provides a .NET distribution.

It can deal with streams with large amount of events or message independent from their nature being historical or real-time. It provides various techniques for filtering, aggregating, and analyzing and reacting to events and event streams. Esper allows assigning an event stream to each defined event type. Its tailored event processing language (EPL) is one of the most expressive querying languages for complex event processing. Beside a very active developers community, Esper has an extensive documentation for the development of own applications and adapters. It provides a well document API, and many features for pattern matching, event stream processing, input and output adapters, and event representation. Esper is highly scalable, memory-efficient, and capable of processing a high velocity and high variety data.

The Enterprise Edition also provides more features, e.g. a graphical query editor and a jquery API, among others. The core features for event stream processing include the following functions: sliding/tumbling/combined windows, familiar SQL-standard-based continuous query language, enumeration methods, date-time methods, Allan's interval algebra, declarative context dimensions, and a high degree of parallelization for processing the same statement as well as multiple statements. In regard to event pattern matching the core features include the following among others, logical and temporal event correlation, crontab-like timer 'at' operator, lifecycle of pattern, pattern-matched events provided to listeners. The basic input and output adapters provide the connection to the environment. Esper comes with the following adapters, CSV input adapter, JMS input and output adapter based on Spring JMS templates, DB output adapter for running DML and for keyed update-insert (aka. upsert), HTTP input and output adapter, socket input adapter.

3.3.7 APAMA

The Apama Event Processing Platform was bought by Software AG in June 2013. The platform is a proprietary product that encompasses a design and deployment environment for CEP applications. It provides also graphical design tools, research and back-testing utilities for CEP solutions. It is a core platform for developing event-driven applications.

The core functionalities are the monitoring of event streams, detection and analysis of event patterns, and according triggering of re-/action. The detection functionality can be configured to find time-based, attribute and location based relationships. Event stream correlators provide a multi-dimensional filtering mechanism for multiple data streams.

The Apama platform concept is based on three aspects; (i) external systems that provide events are connected to Apama with different adapters, e.g., real-time feeds, static data sources, (ii) the Apama correlator run-time engine, the central component of the Apama platform that consumes and processes the event streams triggers according actions, and (iii), the Apama platform provides a unified productivity tooling for fast development and testing of application.

The Apama Platform provides an Integration framework that consists of three layers for integration; transport layer, codec layer, and semantic mapper layer. The transport layer communicates with the source or sink of the event stream, usually through an adapter plugin written in C, Java or C++. The codec layers are written in C, C++ or Java and used to transform message from custom into a normalized form and vice versa. The semantic Mapper Layer transforms the normalized events into Apama events and vice versa. It uses XML for the transformation based on a set of mapping rules.

3.3.8 Oracle Complex Event Processing

The Oracle Complex Event Processing is a standalone event stream processing platform that provides filtering, correlation, and processing of events. It is also a component of oracle's SOA Suite (http://www.oracle.com/technetwork/middleware/soasuite/overview/index.html). It provides an open architecture for designing, defining, developing and implementing Oracle Event Processing applications.

OEP offers visual development tools as well as the possibility to develop an event stream processing application in Java. Oracle offers an Eclipse IDE plugin that provides a visual directed graph canvas and palette to easily build an event processing application. It considers data and event sources for processing. If needed, the visual development can be enriched with Java or Spring DM code.

OEP supports standards based continuous query execution. Querying is based on an in-memory execution model. The standard ANSI SQL 99 is used for querying. OEP is fully compliant to the ANSI SQL 99 Standard. Web applications can communicate with the event processing platform through Http and subscribe to application channel that pushes events to client. According to Oracle the platform can process more than 1.000.000 events per second.

3.3.9 Odysseus Data Stream Management Engine

Odysseus is an in-memory data stream management (DSMS) and complex event processing (CEP) system that can be plugged between any data sources and data target. Odysseus offers flexible adapters and plugins so that source can be manifold. Odysseus is flexible and customizable framework for processing continuous data and data streams. Odysseus is platform independent as it is developed in Java.

It offers filtering, correlating, enhancement or transformation, or fusing of events into complex events and other enriched information. Odysseus also provides optimization techniques for an efficient execution.

It is built on client (Odysseus Studio)/ server architecture (Odysseus Core). Odysseus Studio provides visual graph query creation that can be enriched with code. Queries can also be programmed in code. It uses Eclipse RCP, In-Memory Database technology and provides the usage of various query languages, which are Odysseus Script, Procedural Query Language (PQL), StreamingSPARQL, or StreamSQL (CQL) queries. It is based on Java.

Odysseus is a research prototype of the Information Systems Group (http://www-is.informatik.uni-oldenburg.de/) at the Department for Computer Science at the University of Oldenburg.

The core features of the system consist of reuse of similar queries (sharing), a multi-user environment, built-in optimization strategies, possible integration of new operators, built-in parser for mathematical expressions, different scheduling and buffer strategies, several levels of abstraction through query languages and query processing, and an Odysseus Script language for facilitating the control and execution of the system.

Odysseus core input and output adapters and feature for connecting the platform to its environment include among others, an access framework with various, combinable and extensible access protocols and mechanisms, and access via web, console or Java.

3.3.10 ETALIS - Event TrAnsaction Logic Inference System

ETALIS is an open source system for CEP. ETALIS stands for *Event TrAnsaction Logic Inference System*. It supports two languages, one for defining events, the ETALIS Language for Events, and one for defining queries, the Event Processing SPARQL (EP-SPARQL).

ETALIS also supports *reasoning* about events, context, and real-time complex situations (i.e., *Knowledge-based Event Processing*). ETALIS is implemented in Prolog and hosted as an open source project on http://code.google.com/p/etalis/. It is published under the GNU LGPL.

It appears that ETALIS is an exception among all the CEP Systems as it is the only system developed in Prolog, a declarative logic programming language. Its main features are support for the Event Processing SPARQL (EP-SPARQL) language, declarative rule-based language for event processing, detection of complex events and reasoning over states (with logic rules), support of classic (e.g., sequence, concurrent conjunction, etc.) and complex event operators (Allen's interval algebra), event aggregation/filtering/enrichment/projection/translation/multiplication, sliding windows, and the processing of out of order events as well as alarm events. A shared computation plan can be designed for the evaluation of complex event rules.

Due to the lack of detailed information, nothing can be said about adapters and interfaces, the connection to the environment of the system, i.e., the connection to event sources and event sinks.

3.3.11 SAP Sybase Event Stream Processor (ESP)

SAP Event Stream Processor (ESP) is a high-performance complex event processing engine. It uses CCL and SPLASH (a simple scripting language) for querying event streams. A visual authoring tool for creating queries is provided. Queries can also be programmed in CCL. SPLASH provides functions and operators to extend CCL queries. It supports the concepts of analysis of streams and windows. ESP windows can be queried via ODBC and JDBC.

The whole platform is built on a cluster architecture that provides scalability and the ability to add new continuous queries without interruption. It provides also a framework to define reusable modules, functions, and operators, e.g., design time tools (plugins for HANA studio), Sybase ESP Software Developers Kit (SDK) that includes interfaces and tools for building custom adapters. The ESP SDK is available for C/C++, Java and .NET. A visual query as well as CCL programming tool is provided that is based on eclipse. ESP integrates with SAP HANA platform (In-Memory Database technology). SAP Sybase comes with an extensive list of adapter. The main adapters are message buses, database adapters, file adapters, sockets, email (SMTP), web services, financial market data, and FIX (financial information exchange).

3.3.12 Evaluation Summary of CEP Systems

The evaluation found only small differences in functionality of the reviewed systems. The expressiveness of query languages is very similar to each other, almost all systems support the usual adaptors to JMS, JDBC, REST via JSON, SOAP, and different textual file formats like CSV, RTF, or TXT.

The GET project requires integrating the CEP system into the architecture of the GET platform and the GET core platform. Therefore it is advantageous to select a small system. Additionally, the architecture of the GET platform is still subject to change, as the project evolves. Therefore it is essential to choose a system that can be adapted to changing requirements. This requirement suggested to use an open source system and to some degree excluded proprietary systems.

Another consideration was that in the GET project event streams need to be matched against static data, like train schedules, customs opening hours, and process models. But especially the correlation of events to existing process models was not supported by any of the reviewed systems directly. This functionality has to be added during the project. Again this suggested using an open source system.

In regard to event processing languages supported by the systems, EPSPARQL provides slightly more functionalities than Esper's EPL. However, the two systems supporting EPSPARQL, Odysseus and ETALIS appear to be only prototypes, lack an extensive developer community and detailed documentation. Beside the above-mentioned requirements, Esper being one of the most expressive querying languages is a major aspect for choosing Esper for the development of the event processing techniques in GET Service. Additionally, a minor bonus was the availability of Esper expertise for the development. The students who help in the development of the event aggregation service had prior experience with Esper.

All of the above stated requirements are ideally fulfilled by Esper. It has the capability to integrate historic data from JDBC databases in its queries. It is small, extensible, fast, and developed in Java, the language that is also used in the other GET platform components. It also may be extended by semantic annotations for semantic complex event processing (Zhou, Simmhan and Prasanna 2012). Furthermore, it provides excellent documentation and has an active community.

The evaluation of CEP systems is summarized in Table 5.

Table 5 Comparison of reviewed CEP systems

System	Query Language	Expressiveness	Implementation language	Documentation	Technology	License
Esper	EPL, graphical (only EE)	+++	Java .Net	+++	JMS, Http, CSV, JDBC, Sockets	GNU GPL
jBoss Drools	Rete (Rule)	++	Java	++	JMS, files, DB, Sockets	Apache SL 2
WS02	SiddhiQL	++	Java	+	JMS, REST, SOAP, JSON	Apache
TIBCO BE	Rete (Rule), Graphical	++	n.d.	+	JMS, XML, IBM MQ Series	Proprietary
Oracle EP	CQL, ANSI SQL 99, Graphical	++	Java	++	HTTP, DB	Proprietary
MS Stream-Insight	LINQ	++	.Net, C#	++	DB, files, SOAP	Proprietary
Odysseus	StreamSQL, PQL, Streaming-SPARQL	n.d.	Java	+	DB, files, SOAP, JMS, HTTP	Research Prototype
ETALIS	EP-SPARQL	++	Prolog	o	n.d.	GNU LGPL
SAP Sybase ESP	CCL, SPLASH, Graphical	++	C,C++,Java	++	JMS, Sockets, DB,SMTP, HANA	Proprietary
Streambase CEP	StreamSQL	+++	Java, C++, Python	++	JMS, JDBC	Proprietary

3.4 Summary of Event Query Languages and Systems

As described in the previous sections, the suitability of a CEP system for a specific context highly depends on the characteristics of the query language it uses. In the case of GET Service, we need to focus on open source systems and/or standard languages that can be freely used in any system, hence leaving aside many of the approaches included in Sections 3.2 and 3.3. In particular, we decided to use Esper to support event querying in the prototypical implementation of the event aggregation engine, as its SQL-like query language provides all the functionalities required for the basic queries, namely support for specifying disjunction, conjunction, negation, integration with external data sources, different types of aggregations, event instance selection, sequences and event instance consumption. We use the Java implementation of the language, as it is open source and we already have expertise in the language. Furthermore, there is a lot of documentation about Esper and EPL available. Professional support, customization, as well as an enterprise are offered by companies on the market. In this regard professional support and certainty for continuous development of the platform is assured.

4 Design of the Information Aggregation Service

The Information Aggregation Service is a main component of the GET Service platform responsible for collecting events from different sources and processing them in order to offer a unified interface to clients, planners, information providers, and other stakeholders (GET Service, 2012). Thereby, the aggregation service among others supports the use cases of track & trace, vessel arrivals, and capacity visualization (van der Velde, Saraber, Grefen, & Ernst , 2013). The specific functionalities that this service requires in transportation are described in the following sections.

4.1 Import and Export of Event Data

It is important that the interfaces of the GET Service platform adhere to existing messaging standards and interchange formats of all services that are used by all stakeholders involved. For this purpose, the messaging standards of logistics were investigated in deliverable D2.1 (van der Velde, Rook , Saraber , Grefen, & Ernst, 2013). Four common message types where identified: 1) EDIFACT is used by shipping lines, terminal operators, or customs, 2) EDIFACT XML (UN/CEFACT working groups), 3) Business Logistics XML by larger Logistics Service Providers (LSPs), and 4) GUI and Excel uploads and downloads are used by smaller Transportation Service Providers (TSPs). Platforms like Port Community Systems are capable of translating the information exchange for different standards (e.g. EDIFACT towards business logistics XML) and are doing this for large communities (Portbase currently serves over 2800 organisations). Thus, these systems must be considered as sources and may be integrated in the aggregation service. Furthermore, applications might use JSON as exchange format, which has less markup overhead in comparison to XML. To unify the communication in logistics for all stakeholders, the e-Freight project[9] developed a standard framework that also needs to be considered in the GET platform. It is a standard for freight information exchange covering all transport modes and stakeholders. Each of the above mentioned message types can be transported over different channels using different protocols and services, for example through SOAP web services, HTTP protocols, RPC, or FTP file transfers.

Thus, to extract events from all exchanged messages and to publish transportation-related events in the aggregation service of WP6, the information aggregation engine requires four generic interfaces for communication:

[9] http://www.efreightproject.eu/

1. Interface to import messages of events from different sources (e.g., from client devices of LSPs, GET Infra Platform) provided in different formats. Based on the above-mentioned message formats, the engine must be able to call external web services, connect to message queuing services, generate HTTP requests, and download files from FTP. Additionally, it has to offer an interface, to which clients can push events contained in messages.
2. Interface for identifying the event information in these kinds of messages. By implementing adapters the aggregation engine defines where and how to extract events from all the imported messages types. Thus, it must be possible to import events using EDIFACT, XML, Excel, JSON, and the e-Freight format.
3. Interface for submitting event patterns to be notified of the occurrence(s) of events that the stakeholders are interested in. For this purpose, the aggregation service must enable all stakeholders to specify these event patterns in a well-chosen language, preferable in Esper (see Section 3.4). Furthermore, the aggregation service must be extendable to implement the functionality to derive event patterns from transportation plans, logistic process models, and route descriptions as they will be researched in Task 6.4 (GET Service, 2012).
4. Interface to forward events to interested targets. Thus, the aggregation engine in GET Service must itself provide functionalities to publish events and provide them to the stakeholders involved in transportation. Community systems or other platforms might act as intermediate event distributors. Thus, the engine needs to implement a message queuing service to distribute events and also forward notifications containing information on a subset of events. This forwarding may be implemented as HTTP responses but may also be realized through emails to be shown on the mobile client devices. The format for the notifications depends on the client devices but should at least adhere to the message standards mentioned above, including EDIFACT, XML, Excel, JSON, and the e-Freight format.

Each of these interfaces must provide capabilities to access and modify the functionalities in order to adapt to a changing environment. Therefore, the interfaces should provide methods to support the standard operations of create, read, update, and delete (CRUD). For example, partners interacting with the GET Platform should be able to adapt their own aggregation rules to changing plans, or set up new event sources, but also be able to delete rules that are no longer required.

4.2 Normalization of Events

Because events are collected from different sources that can have different formats, events need to be normalized into a common unified format for further event processing. The normalization needs to take place in the aggregation service in order to process events.

The normalization includes the definition of the format of the normalized events, as well as the stored event properties that are available for purposes ranging from information extraction to correlation of events based on values. The different formats and their differing structure mentioned in the previous section imply that the target event format needs to be extensible and general enough to allow for incorporation of structured or unstructured information from all different sources.

The transformation into the unified format can be specified by corresponding adapters. In the context of event processing and the aggregation service of WP6, an adapter refers to a component that formats heterogeneous event data into a suitable input format. For example, an event stream in XML format can be processed by an XML parser and events can be extracted based on conversion rules, which can include mappings for different formats of dates and timestamps to the internal format. The mapping rules should be extensible and reusable, such that the task of connecting new sources to the GET Service Platform can be conveniently performed.

4.3 Integration of Event Processing

Once the events are made available to the aggregation service in a normalized format, the actual event processing has to be performed in form of aggregation and correlation. Thus, the functional requirements for the event processing engine is to support the above mentioned relations between events, i.e. to detect relations based on time, causality, aggregation and correlation. These relations are stored as rules that allow to relate and to aggregate several events.

Section 3 showed that Esper is the most suitable event processing system. Therefore, the aggregation engine must incorporate Esper and register patterns and rules written in Esper's EPL to match them to the incoming events in Esper. Furthermore, more advanced methods are to be incorporated to extract specific rules in the selected Esper event processing language from process models or transport plans (see Sections 4.4 and 4.5).

Furthermore, the aggregation service is expected to capture a large amount of events and needs to be able to process them within a complex environment where many actors subscribe for their respective events. The actual CEP engine that is used in GET Service is therefore required to be *scalable*. Again, for this purpose the implementation should consider the integration of Esper (see Section 3.4).

4.4 Cooperation of Event Processing and Discriminative Classifiers

The ability not only to monitor but also to interpret the context information can be seen as one of the major contributions of the event processing component in GET platform. Indeed, streams of transportation-related events represent a temporal snapshot over the current development of transportation processes. As an example, the consecutive coordinates and altitude levels of an aeroplane trace its movements. The events may rise from different sources (e.g., weather conditions along the route, traffic information in the arrival airport, etc.) and can altogether concur in the creation of a dangerous situation for the regular advancement of the transportation. Therefore, it is of considerable relevance to distinguish the sequences of events that lead to a disruption from those that are safe. Evaluating queries over event streams is a basic approach to this extent. Such queries would weigh the combination of events over time in order to determine whether the current evolvement of facts is likely to end up in a risky situation, or not. However, it would be impractical to predefine all such queries a priori. This is due both to the quantity of possible concurrent causes to check, and to the unfeasibility of foreseeing any possible anomalous sequence of events, let alone grounding it to queries over event streams. To this extent, classifiers from the field of Machine Learning (Mitchell, 1997) can be of significant help. For instance, Support-Vector Machines, SVM (Cortes & Vapnik, 1995) are supervised learning models for linear classifications, i.e., able to identify a hyperplane in the space of features that separates numeric representations of input objects in two different categories. The hyperplane is determined on the basis of a learning process made on labelled historical data. In the context of transportation-related events, e.g., labelled historical data can represent the reported trajectories of aeroplanes, divided into those that were known to have landed in time and in the expected airport, and those, which were known to have been delayed or diverted. Once trained on such data, the classifier (e.g. SVM) can analyse current flights and predict whether they show an anomalous behaviour, or not. The input can be provided by the event processing engine, as long as the transportation process specifies the information to be extracted from events to this extent. The learning systems can be used indeed to correlate available data, in order to detect anomalies based on previous knowledge. The selection of independent and dependent variables for the decision functions is thought to be determined a priori, since they are strongly domain-related. For instance, the SVM can recognise a possible diversion on the basis of features such as gained distance from the departure airport, velocity and altitude of the aeroplane. However, the input sources (e.g., flight monitoring services) as far as the information aggregation and features extraction (e.g., from positional data to distance, velocity and altitude) are meant to be predefined. The upcoming deliverable D6.4 will elaborate on these user-based specifications.

On the basis of the prediction made by the classifier, a new event raising an alert can be generated, in case of anomaly detection. Therefore, it is required for the event stream to be restructured in a way that makes it readable from an external classifier, on one hand. On the other hand, the classification returned as a result has to be treated and transformed into a *virtual event object*. It is worthwhile to recall here from (Baumgrass, Cabanillas, Di Ciccio, Meyer, & Schmiele, 2013) that a virtual event object is an event object, which does not exist per se in the physical world. It represents though a situation that is likely to occur, based on the analysis of happened events.

4.5 Correlation of Events to Processes

An additional function within the aggregation service will be the (semi-)automatic derivation of correlation rules on the basis of process data and transport plans. The algorithms to achieve this functionality will be researched and investigated in the course of the GET Service project by WP6.

The intention is to analyse process models (as they will be provided by WP4 and executed by WP7), route descriptions, or transportation plans as input that is analysed to derive correlation and aggregation rules. For this purpose, the components outside of the aggregation service need to provide these documents in a way they can be parsed and event patterns can be derived. In case of process models in BPMN2.0 format including the definition of process event monitoring points, the approach of (Backmann, Baumgrass, Herzberg, Meyer, & Weske, 2013) may be implemented to identify whether an instance of a process model was executed successfully.

4.6 Notification Mechanism

The aim of the GET Service platform is to offer services to many clients and therefore needs to adhere to common notification paradigms. The publish/subscribe paradigm is very common in event processing (Mühl, Fiege and Pietzuch 2010), and needs to be supported by the information aggregation service. Using this paradigm allows clients and planners to subscribe to certain types of events or aggregated events. For example, a planner might subscribe to all events that are correlated to the respective Transport Execution Plan that the planner has created. Then, if events occur during execution, the planner is notified about their occurrence and can react accordingly.

Beside the publish/subscribe paradigm, regular access to events is required in the GET Service Platform. That is, information providers need the option to add new events directly into the platform via the appropriate interface (push). And additionally, the option to query for recent events from the event history should be made available in that interface (pull).

4.7 Summary

The above-mentioned sections point out that the required functionality exceeds the current capabilities of the aggregation service in the GET Service project. WP6 aims to design appropriate filtering mechanisms at early stages, to reduce the burden on the correlation and prediction activities. Furthermore, the derivation of correlation rules based on processes range from very simplistic approaches (e.g., correlating by container id), to more sophisticated, control flow, location, and time-aware correlation mechanisms, which are candidates to be researched within the deliverables D6.3, D6.4.1, and D6.4.2 (GET Service, 2012). To provide a brief overview of the requirements, a tabular representation is given Table 6.

Table 6 Summary of required capabilities of the aggregation service

Requirement	Description	Scope
R1. Heterogeneous Sources	Connection to different kinds of event sources	D6.3
R2. Heterogeneous Formats	Collect events from different message formats	D6.3
R3. Normalization of Events	Store events of different formats in same normalized format	D6.3
R4. Event Storage	Store normalized events in a central database	D6.3
R5. Event Processing		
R5.1 Event Aggregation	Provide functionality to aggregate events of finer granularity to single events	D6.3
R5.2 Query Subscription	Register queries to be informed of events of interest	D6.3
R5.3 Domain-Specific Query Subscription	Register transportation-related queries to be informed of events of interest	D6.4.1 & 2
R5.4 Domain-Specific Event Correlation	Automatically correlate events to transportation processes	D6.4
R6. Notification Mechanism	Notify subscribed clients of respective events	D6.3
R7. Event Classifiers	Determine criticality of an event for transportation	D6.4.1 & 2

5 Architecture of the Information Aggregation Service

This section presents the architecture of the information aggregation service.

5.1 Aggregation service positioned in the GET Service Platform

The connection of this deliverable with the general GET Service architecture developed in D2.2 (van der Velde, Saraber, Grefen, & Ernst , 2013) is shown in this section. Given the architecture developed in D2.2, the aggregation service of WP6 is part of the Core GET Service platform (see Figure 2). The Core GET Service platform is able to correlate and aggregate the events coming from multiple Infra and Client platforms. It also contains an information warehouse, which stores static (schedules, master data) and dynamic (capacity) data. The aggregated events are published or can be retrieved by client platforms or the extended GET platform for process execution.

In particular, the aggregation service is composed of a subset of components implementing the Core GET Service platform (see Figure 2). It will mainly implement the storage of subscriptions and events (Subscription Store and Event Store) as well as the components that enable the import of events (Event Channel) and the aggregation and correlation of events (Event Aggregator and Event Correlator). In the following sections the aggregation service will be detailed describing how these components will be implemented.

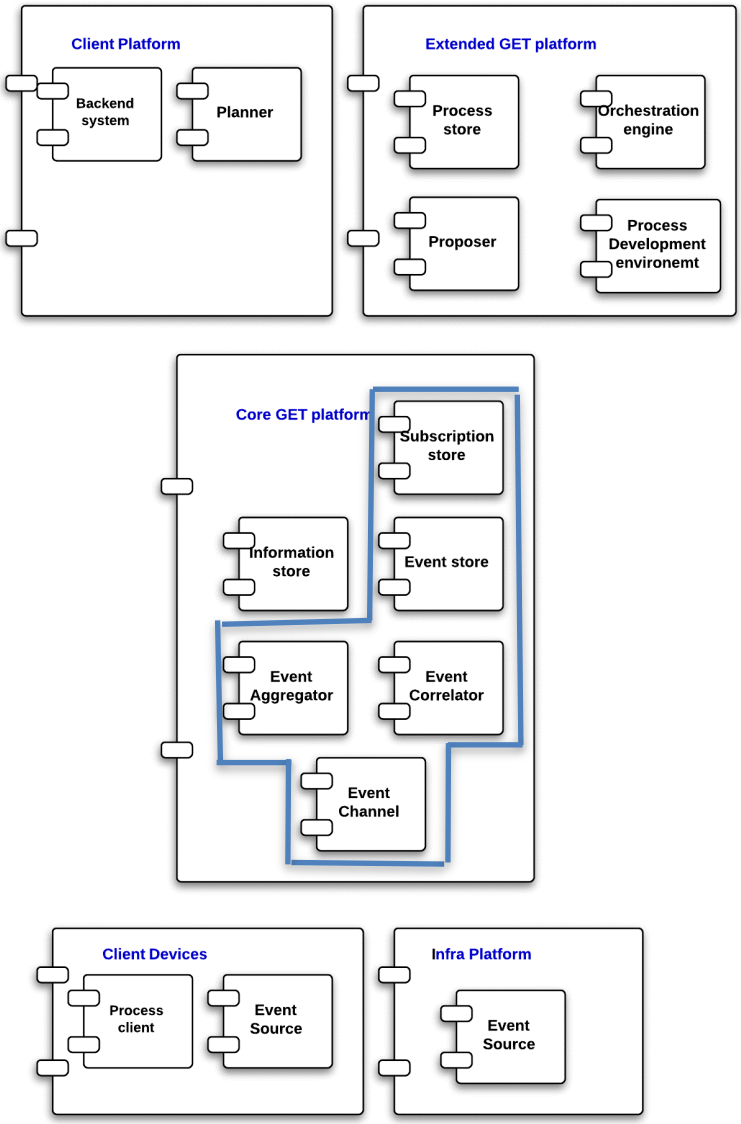

Figure 2 GET Service Platform Components (from D2.2.2)

5.2 Design of External Interfaces

The aggregation service offers four interfaces to be used either by external event sources (e.g. driver, weather stations) or event consumers (e.g. planner, driver). Furthermore, it implements an interface to access the information store to request static information. These interfaces are required to provide the functionalities described in Section 4.

The five interfaces are shown in the component diagram in Figure 3 and summarized as follows:

- The **EventAdministrationInterface** receives the structural description of an event type (see Definition 2) and offers further administrative tasks related to events. The communication through this interface will be implemented in two ways. It can be either initiated by any event source sending the event type description of the events it publishes (push) or it can be configured inside the corresponding event source adapter (see EventSourceAdapter, Figure 5). The implementation will be realized by a web service procedure to which the event source can push the event type description.
- Through the **EventSourceAdapterInterface** the aggregation service is able to receive events (resp. implementing R1 in Table 6). Each event source will be connected through a specific adapter. This adapter then offers an own interface that can be used by the event

source. Each adapter has to internally use the EventImportInterface (see Figure 5) to make received events understandable for the AggegrationService.

- The **EventSubscriptionInterface** is used to register subscriptions in the aggregation service. These subscriptions can be arbitrarily complex, i.e. might be composed of specific event processing queries (see Section 3 for more information on building such queries using different languages). The subscriptions should be pushed to the aggregation service. Therefore, the aggregation service component of WP6 will provide an implementation of a request-response pattern to register subscriptions in the GET Service Platform. The events being imported via the EventImportInterface are forwarded to the event consumers by the aggregation service based on the subscriptions.
- The **EventServicesInterface** combines all services of the aggregation service that are visible to external consumers. For example, a consumer might submit routes to the aggregation service, which can be used in subscriptions later on. However, this interface is supposed to be generic and will be specified more detailed in the scope of D6.4.1 and D6.4.2.
- The **StaticInformationInterface** offers access to information to enrich events. The aggregation service uses it to receive all types of information, e.g. about transportation plans and schedules. The information store will be part of the GET Core platform, however, not part of the aggregation service.

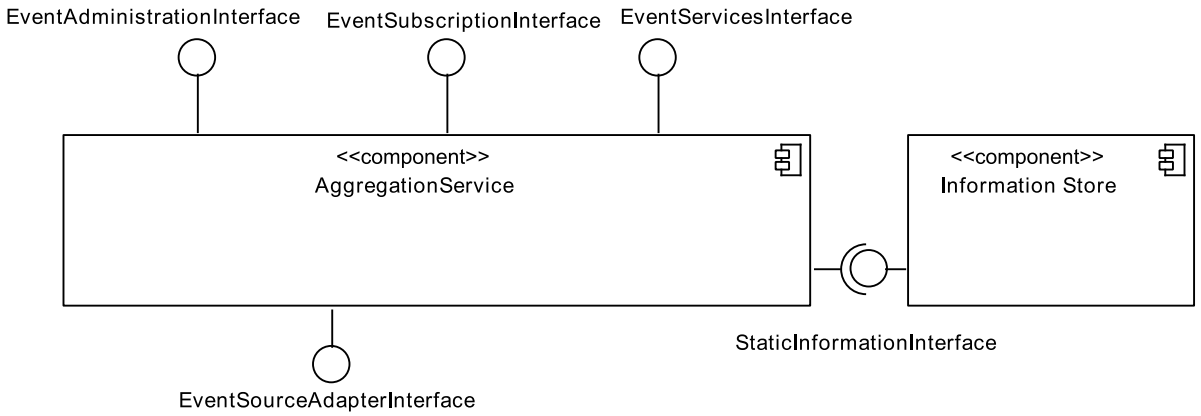

Figure 3 Interfaces of the AggregationService

In accordance with the descriptions of interfaces in D2.2.2, Table 7 summarizes the interfaces with a short description, their inputs and outputs, the interaction pattern realized, and how errors should be handled.

Table 7 Overview of the Interfaces of the Aggregation Service

Interface ID	**EventAdministrationInterface**
Description	Push event type definitions to the platform, necessary in order to import events of this type, administrative tasks on events.
Input	Structural description of an event type (e.g. as XSD), task execution
Output	Confirmation
Interaction Patterns	Synchronous request/response
Error Handling	Synchronous confirmation
Interface ID	**EventSourceAdapterInterface**
Description	Events are pushed by event source, pulled from event sources or received by a subscription to event sources.
Input	Events including a reference to its event type (e.g. XML)
Output	None
Interaction Patterns	Synchronous push or pull, publish/subscribe (always depends on the adapter). Events can be pushed to the platform or the platform can pull events from known event sources on request or with a subscription.
Error Handling	No error handling
Interface ID	**SubscriptionInterface**
Description	Subscribe for events by queries (or other criteria)
Input	event processing query (e.g. String or EPL) or other event criteria
Output	ID of a queue, events
Interaction Patterns	Synchronous request/response, publish/subscribe
Error Handling	Synchronous response or exception, retransmission on publish/subscribe communication
Interface ID	**EventServicesInterface**
Description	Offers services in relation to events, e.g. process model monitoring or route handling.
Input	Process models (e.g. BPMN), transport orders (e.g. XML), or routes (e.g. JSON)
Output	Events
Interaction Patterns	Synchronous request/response
Error Handling	Synchronous response or exception
Interface ID	**StaticInformationInterface**
Description	Request/response interface to access information, e.g. about route information and timetables.
Input	Database queries or function calls to databases
Output	Route, timetable, transportation plan
Interaction Patterns	Synchronous request/response
Error Handling	Synchronous response or exception

5.3 Structure and Functionality of the Information Aggregation Service

In this section the three components EventHandler, EventProcessing, and EventServices, which implement the aggregation service are detailed. Figure 4 shows the three components of the aggregation service from Figure 3, the interfaces they implement and also their connections. In the middle, the **EventProcessing** handles event transformations and querying. Thus, it includes the functionality of event processing and implements the requirements R5.1 and R5.2 and provides the functionality to implement R5.3 and R5.4 shown in Table 6. In Section 3.4, Esper was found to be the most suitable event processing system including its own language. Thus, the EventProcessing

component will be represented by Esper. However, to provide the required event aggregation and correlation functionality for logistics, Esper will be extended by the EventHandler and the EventServices.

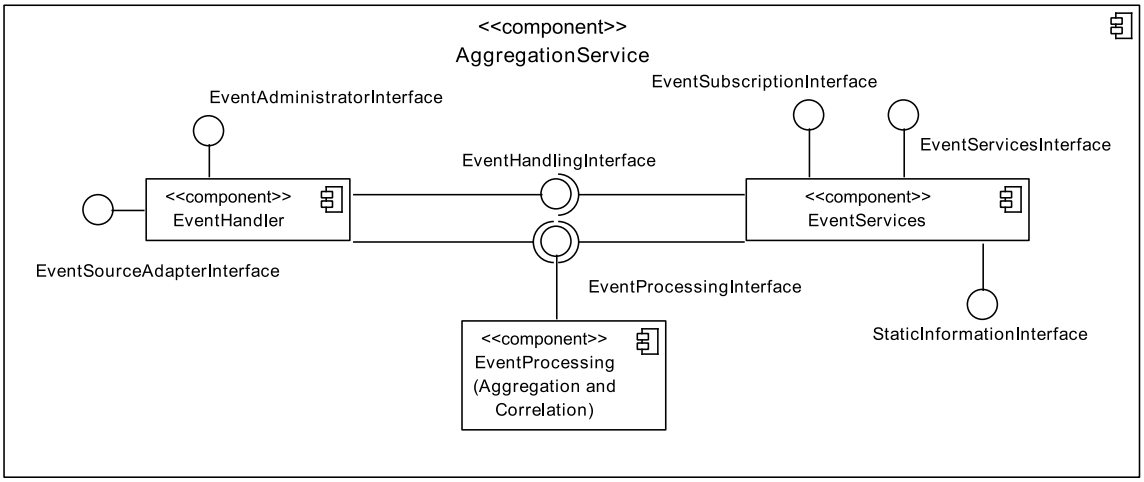

Figure 4 Architecture Overview of the Aggregation Service

The **EventHandler** will be implemented to collect, receive, and handle events from different kinds of systems in different formats. This means, it implements the requirements R1, R2, R3, and R4 in Table 6. For that purpose, it provides the EventAdministratorInterface and the EventSourceAdapterInterface. The internal structure of the EventHandler is represented by five components (see also Figure 5):

- Each kind of event sources requires an **EventSourceAdapter**, which is able to retrieve events from any kind of event source (over the EventSourceAdapterInterface). Event sources differ in the mechanism which they use to provide events, e.g., downloads of event information from a FTP server or offering a web service to request events. Thus, all mechanisms to request events from event sources are considered by implementing a corresponding event source adapter through which requirement R1 shown in Table 6 is realized.
- The **EventReceiver** is responsible for converting the events of an event source into event objects that the aggregation service can process. For example, one EventSourceAdapter receives events in form of an XML document and another adapter in the JSON format (cf. Section 4.1 and R2 in Table 6). Thus, the EventReceiver normalizes events in different formats and converts them into the internal structure for processing; i.e. implementing requirement R3 shown in Table 6.
- Events are stored in the **EventStore**. It corresponds to the EventStore presented in D2.2.2 (see Figure 2) that implements requirement R4 shown in Table 6. Examples for events and event types as they will be stored in the GET Service project are detailed in deliverable D6.1 (Baumgrass, Cabanillas, Di Ciccio, Meyer, & Schmiele, 2013).
- The **EventManager** handles all operations of events. This component is the connection between the Event Receiver, the stores and the EventProcessing component. In particular, it is responsible to save and load events and its types from the stores and also to create event types.

In summary, the EventHandler is the central component of the aggregation service, which implements the *Event Channel* introduced in the core architecture of D2.2.2 (see Figure 2).

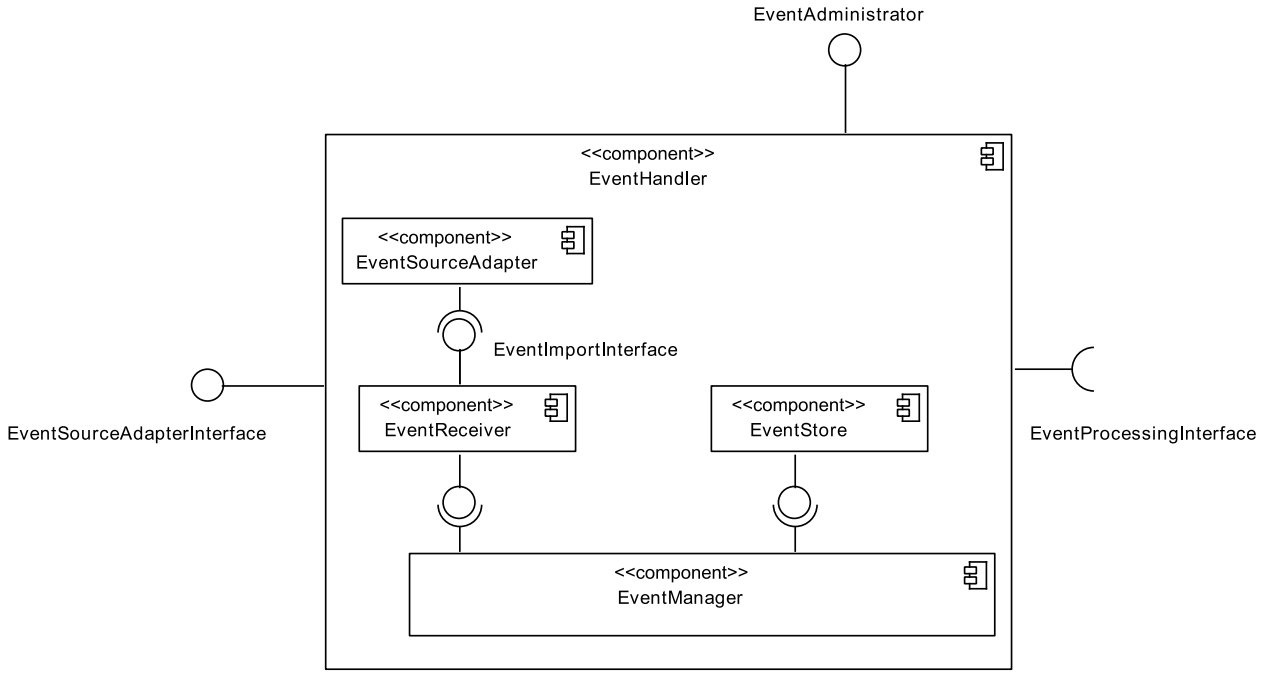

Figure 5 Structure of the EventHandler

The **EventServices** component handles the associations of events to information stored in the event store and handles the communication to event consumers. For this purpose, it includes the EventSubscriptionInterface and the EventServicesInterface to external consumers as well as the EventProcessingInterface to the QueryProcessing component and the StaticInformationInterface to the InformationStore.

At the moment, three main components are required for the implementation (see Figure 6):

- The **SubscriptionManager** handles the publication of events to the event consumers based on subscriptions they provided and which are stored in the subscription store. For this purpose, each subscription must include an address to which the events are pushed.
- All subscriptions are administered in the **SubscriptionStore**. It corresponds to the SubscriptionStore presented in D2.2.2 (see Figure 2).
- The **ServiceUnits** component is a placeholder for all upcoming functionalities that will be developed in the project to enrich events with external knowledge coming from the InformationStore, especially in Task 6.4 (Development of automated event processing techniques). For example, the coordinates given by an event may be used to identify the city in which the event occur which in return might of interest for an event consumer (given by a subscription). A first idea of such enhanced event processing is published in (Metzke, Rogge-Solti, Baumgrass, Mendling, & Weske, 2014). Furthermore, functions for prediction should be developed in this component, to implement the functionality discussed in Section 4.4. In particular, ServiceUnits will be used implement the requirement R7 shown in Table 6.

In summary, the EventServices component holds the functionality to implement the event correlator presented in D2.2.2 (see Figure 2). Its purpose is to correlate events to processes but also to information stored in the information store and is therefore used to extend the platform and realize the requirements R5.3, R5.4, and R7 shown in Table 6. Furthermore, it will be used to allow the subscription to events and therefore implement the requirements R5.2, and R6 shown in Table 6.

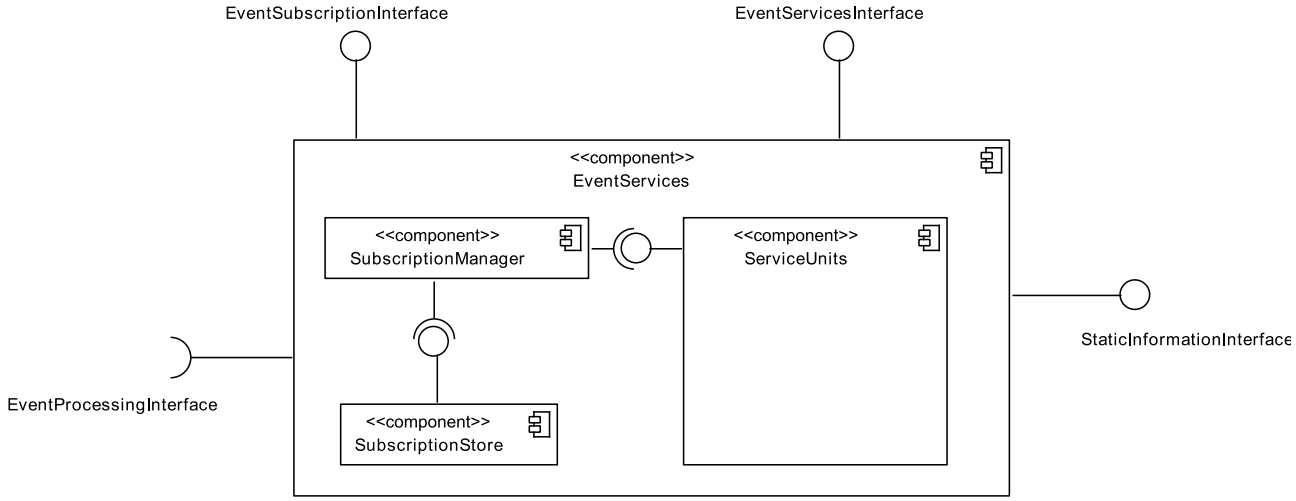

Figure 6 Structure of the EventServices

6 Conclusion

Throughout D6.2, the architecture of the component for the information aggregation service has been described. The design of the information management and processing devised here builds upon the conceptual analysis and definition of transportation-related events provided in preceding deliverable D6.1 (Baumgrass, Cabanillas, Di Ciccio, Meyer, & Schmiele, 2013).

In particular, starting with an analysis of the building blocks that conceptually constitute event stream processing in literature, an insight into event stream query languages and systems has been provided. A comprehensive examination of those features that characterise the languages has been conducted and reported, focusing in particular on the support that the current state of the art could offer to deal with the scenarios described in deliverable D1.1 (Treitl, et al., GET Service Project – Deliverable 1.1: Use Cases, Success Criteria and Usage Scenarios, 2013). Once the different characteristics of query languages have been investigated and compared, their suitability to the needs of GET Service has been assessed. To this end, an evaluation of the currently available event processing systems has been conducted. A discussion on the possible room for improvement in the adoption and adaptation of such systems concludes the analysis of the state of the art.

Thereafter, the requirements that the information aggregation service must fulfil have been detailed. They serve as the basis according to which the architecture of the component is designed. Indeed, this document ends with a thorough analysis of the interfaces offered by the event processing module, along with the description of its internal components and the functionalities offered.

The future realisation of the information aggregation service component will reflect such specifications. Indeed, the upcoming deliverable D6.3 (GET Service, 2012) will describe the implementation, basing on the guidelines explicated in this deliverable and the conceptual model devised in D6.1. For D6.3, the implementation focuses on the requirements detailed in Table 6.

Although all functional requirements are given challenges may be faced during implementation. This is due to the dynamic nature of the development process. These dynamics might occur during the implementation of the single components of the aggregation service and their interaction. More integration effort and dynamics are expected by the integration of the aggregation service in the core GET Service platform. Challenges might also arise from technical requirements (hard- or software) or from necessary event sources that are not publicly available. Furthermore, the complexities of data integration for unifying data, messages, information, and events have to be faced.

7 References

Adi, A., & Etzion, O. (2004). Amit - the Situation Manager. *The VLDB Journal , 13*, 177-203.

Aninic, D., Fodor, P., Rudolph, S., & Stojanovic, N. (2011). EP-SPARQL. *World Wide Web (WWW '11).*

Arasu, A., Babu, S., & Widom, J. (2005). The CQL continuous query language: semantic foundations and query execution. *The VLDB Journal , 15*, 121-142.

Backmann, M., Baumgrass, A., Herzberg, N., Meyer, A., & Weske, M. (2013). Model-driven Event Query Generation for Business Process Monitoring. *Service-Oriented Computing - ICSOC 2013 Workshops.* Berlin Heidelberg: Springer.

Barga, R., & Caituiro-Monge, H. (2006). Event Correlation and Pattern Detection in CEDR. *International Conference on Current Trends in Database Technology*, (pp. 919-930).

Baumgrass, A., Cabanillas, C., Di Ciccio, C., Meyer, A., & Schmiele, J. (2013). GET Service Project – Deliverable 6.1: Taxonomy of transportation-related events .

Braberman, V., Kicillof, N., & Olivero, A. (2005). A Scenario-Matching Approach to the Description and Model Checking of Real-Time Properties. *IEEE Transactions on Software Engineering , 31* (12), 1028-1041.

Brenna, L., Demers, A., Gehrke, J., Hong, M., Ossher, J., Panda, B., et al. (2007). Cayuga. *International Conference on Management of data (SIGMOD).*

Bui, H.-L. (2009). Survey and Comparison of Event Query Languages Using Practical Examples. Munich: Ludwig-Maximilians-Universität München.

Cortes, C., & Vapnik, V. (1995). Support-Vector Networks. *Machine Learning , 20* (3), pp. 273-297.

Cugola, G., & Margara, A. (2012). Complex event processing with T-REX. *Journal of Systems and Software , 85*, 1709-1728.

Diao, Y., Immerman, N., & Gyllstrom, D. (2007). SASE+: An agile language for kleene closure over event streams. *Report* .

Eckert, M. (2008). Complex Event Processing with XChangeEQ. *PhD Thesis* . Munich.

GET Service. (2012). *GET Service Project – Annex I: Description of Work.*

Inc., E. (2008). *Esper Reference Manual*. Retrieved February 26, 2014 from http://esper.codehaus.org/esper/documentation/documentation.html

Inzinger, C., Satzger, B., Hummer, W., & Dustdar, S. (2013). Specification and Deployment of Distributed Monitoring and Adaptation Infrastructures. *ICSOC 2012 Workshops*, *7759*, pp. 167-178.

JBoss Community. (n.d.). *JBoss Rules User Guide*. Retrieved February 27, 2014 from http://www.jboss.org/drools/documentation.html

Kersten, M. (2007). A Query Language for a Data Refinery Cell. *VLDB Endowment (EDA-PS).*

Luckham, D. (2002). *The Power of Events.* Addison-Wesley Longman.

Mühl, G., Fiege, L., & Pietzuch, P. (2010). *Distributed Event-Based Systems.* Berlin Heidelberg: Springer.

Metzke, T., Rogge-Solti, A., Baumgrass, A., Mendling, J., & Weske, M. (2014). Enabling Semantic Complex Event Processing in the Domain of Logistics. *ICSOC 2013 Workshops.* Springer.

Mitchell, T. M. (1997). *Machine Learning.* McGraw-Hill.

Niblett, P., & Etzion, O. (2011). Event Processing in Action. Manning Publications Co.

Raizman, A., Ananthanarayan, A., Kirilov, A., Chandramouli, B., & Mohamed, A. (2010). An extensible test framework for the Microsoft StreamInsight query processor. *Workshop on Testing Database Systems (DBTest)*, (pp. 1-6).

Rozsnyai, S., Obweger, H., & Schiefer, J. (2011). Event Access Expressions: A Business User Language for Analyzing Event Streams. *IEEE International Conference on Advanced Information Networking and Applications*, (pp. 191-199).

Rozsnyai, S., Schiefer, J., & Roth, H. (2009). SARI-SQL: Event Query Language for Event Analysis. *IEEE Conference on Commerce and Enterprise Computing*, (pp. 24-32).

Rozsnyai, S., Slominski, A., & Lakshmanan, G. (2011). Discovering Event Correlation Rules for Semi-Structured Business Processes. *DEBS'11.* ACM.

Schatten, A., & Schiefer, J. (2007). Agile Business Process Management with Sense and Respond. *IEEE International Conference on e-Business Engineering* (pp. 319 - 322). Hong Kong : IEEE.

Seiriö, M., & Berndtsson, M. (2005). Design and Implementation of an ECA Rule Markup Language. *International Conference on Rules and Rule Markup Languages for the Semantic Web*, (pp. 98-112).

Treitl, S., Rogetzer, P., Hrušovský, M., Burkart, C., Bellovoda, B., Jammernegg, W., et al. (2013). *GET Service Project – Deliverable 1.1: Use Cases, Success Criteria and Usage Scenarios.*

Treitl, S., Rogetzer, P., Hrušovský, M., Burkart, C., Bellovoda, B., Jammernegg, W., et al. (2013). *GET Service Project – Deliverable 1.2: Requirements analysis.*

van der Velde, M., Rook , H., Saraber , P., Grefen, P., & Ernst, A. (2013). *GET Service Project – Deliverable D2.1: Report message standards.*

van der Velde, M., Saraber, P., Grefen, P., & Ernst , A.-C. (2013). *GET Service Project – Deliverable D2.2: Architecture Definition.*

Walzer, K., Schill, A., & Löser, A. (2007). Temporal constraints for rule-based event processing. *ACM first Ph.D. workshop in CIKM (PIKM '07).*

Zhou, Q., Simmhan, Y., & Prasanna, V. (2012). *SCEPter: Semantic Complex Event Processing over End-to-end Data Flows.* University of Southern California, Computer Science Department.