UNIVERSIDADE FEDERAL DO ESTADO DO RIO DE JANEIRO
CENTRO DE CIÊNCIAS EXATAS E TECNOLOGIA
PROGRAMA DE PÓS-GRADUAÇÃO EM INFORMÁTICA

AN APPLICATION ARCHITECTURE MODEL FOR EVENT PROCESSING AGENT COMPOSITIONS ON REAL TIME STREAMING ANALYTICS SOLUTIONS

Luís Henrique Neves Villaça

Orientador até 15/08/2018: Leonardo Guerreiro Azevedo

Orientador após 15/08/2018: Sean Wolfgand Matsui Siqueira

RIO DE JANEIRO, RJ - BRASIL
SETEMBRO de 2019

AN APPLICATION ARCHITECTURE MODEL FOR EVENT PROCESSING AGENT COMPOSITIONS ON REAL TIME STREAMING ANALYTICS SOLUTIONS

Luís Henrique Neves Villaça

DISSERTAÇÃO APRESENTADA COMO REQUISITO PARCIAL PARA OBTENÇÃO DO TÍTULO DE MESTRE PELO PROGRAMA DE PÓS-GRADUAÇÃO EM INFORMÁTICA DA UNIVERSIDADE FEDERAL DO ESTADO DO RIO DE JANEIRO (UNIRIO APROVADA PELA COMISSÃO EXAMINADORA ABAIXO ASSINADA.

Aprovada por:

[illegible]

Sean Wolfgand Matsui Siqueira, D.Sc., UNIRIO

[illegible]

Rodrigo Pereira dos Santos, D.Sc., UNIRIO

[illegible]

Geraldo Zimbrão da Silva, D.Sc., UFRJ

RIO DE JANEIRO, RJ - BRASIL

SETEMBRO de 2019

Catalogação informatizada pelo(a) autor(a)

Villaça, Luís Henrique Neves

V172 An Application Architecture Model For Event Processing Agent Compositions On Real Time Streaming Analytics Solutions / Luís Henrique Neves Villaça. -- Rio de Janeiro, 2019.

133 f.

Orientador: Sean Wolfgand Matsui Siqueira (após 15/08/2018); Leonardo Guerreiro Azevedo (até 15/08/2018).

Dissertação (Mestrado) - Universidade Federal do Estado do Rio de Janeiro, Programa de Pós-Graduação em Informática, 2019.

1. Event-Driven Architecture. 2. Real Time Streaming Data Processing. 3. Event Processing Agent Composition. I. Siqueira, Sean Wolfgand Matsui, orient. II. Azevedo,Leonardo Guerreiro, orient. III. Título.

Aos familiares, amigos, conselheiros e educadores.

Agradecimentos

Durante dois anos de trabalho que resultaram nesta tese, diversas pessoas me ajudaram, ensinando, corrigindo os rumos e me apoiando. Não poderia deixar de reconhecê-las.

Agradeço em primeiro lugar à minha família. Aos meus filhos, que sempre se dispuseram em avaliar o que escrevi, inclusive nas diversas submissões de artigos em congressos; à minha esposa, que se desdobrou me apoiando e permitindo que eu me dedicasse a esse estudo; e aos meus pais, que proporcionaram minha formação pessoal e profissional.

Aos meus amigos, que me apoiaram em todos os momentos - e, nas piores situações, me encorajaram a perseverar no Mestrado.

Agradeço imensamente aos dois orientadores que aceitaram minha proposta e me guiaram nessa jornada. Ao professor e amigo de longa data Leonardo Guerreiro Azevedo, que mesmo nas situações mais adversas não deixou de se dedicar de maneira incansável a artigos, minicursos, e a essa dissertação; e ao professor Sean Wolfgand Matsui Siqueira, a quem igualmente admiro, que num momento difícil me acolheu como orientando (e a outros tantos) bravamente encarando esse desafio, somado a tantos outros já comprometidos.

Esse agradecimento se estende aos membros da banca examinadora, Dr. Rodrigo Pereira dos Santos e Dr. Geraldo Zimbrão da Silva, por aceitarem o convite e pela colaboração ao avaliar este trabalho.

Agradeço aos colegas Israel de Oliveira Amorim, Leandro de Oliveira Pastura, Denilson da Rocha Candido, Andre Vieira Scuto, Bruno Bastos Lima, Marcelo Rezende de Fazio, Leticia Vasconcellos Mendes e Marcio Pati Andrada; pela dedicação, colaboração na organização e avaliação do modelo proposto. Todos são para mim uma referência na área de Arquitetura de Software.

Meu reconhecimento aos colegas da Unirio, que me motivaram a seguir lutando pelo meu objetivo: Marcelo Tibau de Vasconcellos Dias, Bruno Mauricio Rodrigues Crotman, Andre de Almeida Farzat, Felipe Ribeiro Santos, Rodrigo da Silva Melo, Fabio Marcos de Abreu Santos, Elton Figueiredo de Souza Soares e Leonardo Mendes Cabral.

Por fim, meus sinceros agradecimentos à Unirio, pelos professores, pela equipe da Secretaria, pelo laboratório, e pelo auxílio no custeio da viagem ao Congresso SBSI; e à Petrobras Transporte S.A., pelo incentivo à minha formação proporcionado pelos gestores Mario Ferreira Miranda Junior, Carlos Roberto da Silva Melo Junior, Jose Ricardo Telles Esses, Paulo José Azevedo Vianna Ferreira, Tarcisio Rangel e Alessandro Leal Eccard.

Villaça, Luís Henrique Neves **An Application Architecture Model for Event Processing Agent Compositions on Real Time Streaming Analytics Solutions**. UNIRIO, 2019. 133 páginas. Dissertação de Mestrado. Departamento de Informática Aplicada, UNIRIO.

RESUMO

Arquiteturas de Processamento de Eventos Complexos (CEP, ou *Complex Event Processing*), mostram alta aplicabilidade em cenários de processamento analítico sobre *streaming* (processamento por fluxos) em tempo real. Embora diretrizes e modelos para essas arquiteturas já tenham sido propostos na academia e na indústria, a composição dos elementos interoperáveis e responsáveis pelo processamento de eventos, conhecidos como EPA (*Event Processing Agent*), permanece um desafio para arquitetos e desenvolvedores de software. Não há um modelo CEP que elucide aspectos relacionados a essa composição.

Este trabalho propõe um novo modelo que trata essa lacuna e contempla requisitos de processamento em alta escala (*Big Data*) por meio de características como construções de processamento baseadas em *streams* e EPA especializados (*e.g.*, em detecção de padrões). Este modelo foi aplicado em um caso real de captura e processamento, via *streaming*, de eventos de utilização oriundos de mais de 200 aplicações. Foram observados indicadores relativos a performance; coesão e acoplamento dos componentes; e assertividade dos resultados, demonstrando potencial para lidar com dados heterogênos que transitam por fluxos de execução sofisticados de forma escalável e eficiente. Especialistas da indústria também avaliaram o modelo qualitativamente, quanto à sua capacidade em atender a requisitos de processamento analítico sobre *streaming* em tempo real.

Palavras-chave: Arquitetura orientada a eventos, Processamento por fluxos em tempo real, Composição de agentes de processamento de eventos.

ABSTRACT

Complex Event Processing (CEP) architectures present high applicability in Real Time Streaming Analytics scenarios. Although guidelines and models for these architectures have been proposed in academia and industry, the composition of its inter-operable elements that are in charge of processing events, known as Event Processing Agent (EPA) remains a recurring challenge for software architects and software developers. There is no CEP model that embraces and clarifies aspects related to this composition.

This work presents a new model that covers this gap, and also addresses large-scale processing (Big Data) requirements through features such as stream-based processing constructions and specialized EPA elements (e.g., for pattern detection). The proposed model has been applied in a real case for capturing, via streaming, utilization events from more than 200 applications in a large company. Indicators regarding performance; degree of cohesion and coupling of components; and assertiveness of processing results were observed, demonstrating potential to handle heterogeneous data through sophisticated execution flows in a scalable and efficient manner. Our proposal has also been qualitatively evaluated by industry experts in terms of its capability to meet real-time streaming analytical processing requirements.

Keywords: Event-Driven Architecture, Real Time Streaming Data Processing, Event Processing Agent Composition.

Contents

List of Figures

List of Tables

List of Nomenclatures

AMQP	Advanced Message Queuing Protocol
API	Application Programming Interface
CAS	Central Authentication Service
CEP	Complex Event Processing
EPA	Event Processing Agent
EPN	Event Processing Network
EPTS	Event Processing Technical Society
HTTPS	HyperText Transfer Protocol Secure (over SSL/TLS)
IP	Internet Protocol
MQ	Message Queue
RDD	Resilient Distributed Dataset
REST	Representational State Transfer
UML	Unified Modeling Language

1. Introduction

This chapter is an overview of this research, and presents the context for the addressed problem, the purpose of the proposed solution, and the scientific research methodology applied in this study.

1.1 Research Context

Complex Event Processing (CEP) solutions provide mechanisms for extraction and generation of valuable information from continuous data feeds, and has benefited from prominent technologies, such as Apache Kafka[1] and Spark[2], to handle large-scale data analytic scenarios on real time, such as stock markets, traffic, surveillance, and patient monitoring (ETZION; NIBLETT; LUCKHAM, 2011; PERERA; SUHOTHAYAN, 2015; ZIMMERLE; GAMA, 2018).

CEP solutions basically capture occurrences within a particular system or domain (events) and include specific logic to filter or transform their information, or to detect patterns as they occur. CEP architectures are inherently complex, especially because of the nature of the operations involved, such as: highly frequent occurrences; large volumes of heterogeneous data; distributed processing; and polyglot data integration - *i.e.*, different types of data models and even data stores might be involved (LEBERKNIGHT, 2008).

EPA (Event Processing Agents) are in the core of CEP architectures. These are entities responsible for the processing of events. CEP business scenarios are processed by

[1]https://kafka.apache.org/
[2]https://spark.apache.org/

interdependent EPA agents. Hence, arrangements of EPA are established so that they can benefit from each other processing capabilities (LUCKHAM, 2002; ETZION; NIBLETT; LUCKHAM, 2011).

EPA compositions, also named EPA architecture, express the power of organizations based on of those agents on CEP scenarios (LUCKHAM, 2002). The idea is to deal with a network of EPAs as just another EPA. The design of EPA compositions should handle requirements, such as efficiency, low coupling and scalability, and it should address demands to facilitate interoperability between EPAs (ETZION; NIBLETT; LUCKHAM, 2011).

This work focuses on Realtime Streaming Analytics scenarios on CEP systems, which accept one or more data streams as input and react to occurrences as they come in, often within few milliseconds, producing one or more data streams as output. A data stream consists of events ordered in time, and each event may bring several attributes related to an occurrence. Solutions other than streaming platforms trigger data aggregations from mechanisms such as rule engines or scheduled batch processes, and do not support those needs (ETZION; NIBLETT; LUCKHAM, 2011; MENDES; BIZARRO; MARQUES, 2013; PERERA; SUHOTHAYAN, 2015), therefore are out of scope of this work.

Our proposal leverages mechanisms from streaming platforms for processing continuous flows of events when composing EPAs. It benefits from features such the elaboration of pipelines of intermediate and terminal operators for stream processing, constructors for parallelism and efficient data structures for storing data while processing,

1.2 Research Question and Motivation

Building cooperating services is a highly complex task, spanning many concepts and technologies that find their origins in diverse disciplines that are woven together in an intricate manner (PAPAZOGLOU, 2009). Industry CEP solutions encompass cooperating services, and demand from system architects and developers the capacity to mitigate challenges of distributed heterogeneous applications, such as fault tolerance, scalability, and performance, to effectively process and manage massive amounts of data produced by distributed data sources. This is required in order to provide solutions that perform real-time analytics for supporting decision-making and timely response(BAPTISTA et al., 2016). On top of those, EPA compositions brings specific requirements, such as aggregating occurrences that relate to each other (process known as windowing) according to composed data structures that represent a processing context (PERERA; SUHOTHAYAN, 2015).

To evaluate studies that could be used as a baseline to structure EPA compositions we conducted a literature review (Section 2.1 and AppendixA) which revealed that no proposal, considering the evaluated date range, organizes the concepts with regard to this subject.

This research investigates the following research question: "How to compose event processing agents to meet demands of Realtime Streaming Analytics solutions?".

The hypothesis is "a model that represents EPA compositions, incorporating stream processing, segregation based on context, historical data processing and incremental training, fulfills requirements for building Realtime Streaming Analytics solutions". As a result, this model can become a guidance for system architects to follow in order to simplify the process of designing such solutions, addressing a current challenge for CEP solution - isolating developers from the complexities of hardcore programming while empowering them with the ability to express CEP scenarios in a simpler and more intuitive manner (DAYARATHNA; PERERA, 2018). This also may improve solution maintainability, which is relevant since use cases for real-time processing do not rely on static implementations (*i.e.*, left untouched for a long time) - software artifacts need to be adapted frequently in order to reflect new requirements and business needs (BAPTISTA et al., 2016).

1.3 Research Proposal and Benefits

This work proposes a model that provides a baseline for compositions of EPAs, assisting on the elaboration of constructs that can be applied on scenarios that cover cases ranging from simple static aggregations of EPAs to dynamic provisioning and decommissioning of those agents. For that, we included elements to represent a complete CEP solution, encompassing all constructs that make it feasible for EPA agents to operate, exchange events and other required information, and provide processed information to interested parties.

It was conceived based on fundamental CEP concepts and Stream processing strategies (Section 2), refined and validated according to discussions on compositions of event processing agents from a variety of industry publications and academic papers. It aims to organize requisites and support the elaboration of the proposed model.

Several advantages are expected from the usage of the model, such as:

- Explicit and formal descriptions of CEP elements aiming at presenting accurate descriptions to mitigate ambiguous interpretations of concepts;
- Reuse through sharing of concepts and functionalities among involved components;
- High level model towards automation, *i.e.*, we offer guidelines for the automation of the processing of business rules. This is independent of underlying implementation technologies (*i.e.*, Spark, Kafka, Flink etc.), which can be chosen for specific implementation cases according to the scenario and available resources.

1.4 Research Methodology

CEP architectures present heterogeneous terminology, data formats and APIs. From a research point of view, the lack of common models, semantics and standards makes the comparison of the existing approaches hard (MENDES; BIZARRO; MARQUES, 2008). This study considered the disperse knowledge on CEP solutions and required constant re-examining of former decisions as we progressed through research stages: planning, defining the model, compartmentalizing it into units of analysis, designing the artifact, implementing a case, collecting metrics, analyzing and documenting the conclusions.

Quantitative metrics were assessed from an experimental study (Section 4.1). The model was applied in a scenario of a global oil transportation company, where a solution collects user requests, as streams of events, related to around 200 systems, and provides a dashboard for monitoring usage of those. We assessed quantitative metrics of our implementation according to guidelines from a performance evaluation framework for CEP systems (MENDES; BIZARRO; MARQUES, 2008, 2013).

Also, a complementary evaluation of our model was performed qualitatively by experienced system architects from industry(Section 4.2). A questionnaire was applied for capturing their feedback in regard to the capability of our model to meet a set of thirteen requirements of real time streaming analytics (PERERA; SUHOTHAYAN, 2015).

1.5 Master Thesis Outline

This thesis follows the application of a research method that contemplates qualitative evaluations and an experimental study in the context of CEP Application Architectures. Figure 1.1 correlates the research stages to the chapter structure of this dissertation, describing where each stage item was approached throughout this study.

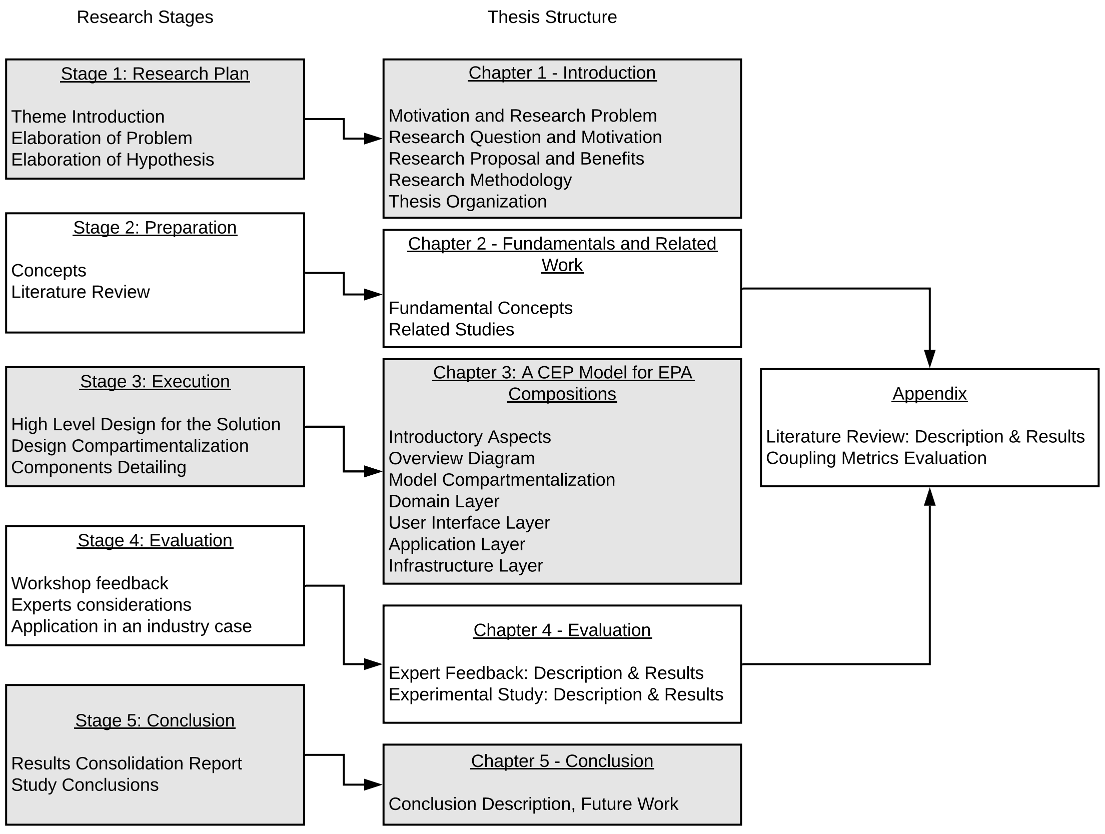

Research Stages
Thesis Structure
Stage 1: Research Plan
Theme Introduction
Elaboration of Problem
Elaboration of Hypothesis
Stage 2: Preparation
Concepts
Literature Review
Stage 3: Execution
High Level Design for the Solution
Design Compartimentalization
Components Detailing
Stage 4: Evaluation
Workshop feedback
Experts considerations
Application in an industry case
Stage 5: Conclusion
Results Consolidation Report
Study Conclusions
Chapter 1 - Introduction
Motivation and Research Problem
Research Question and Motivation
Research Proposal and Benefits
Research Methodology
Thesis Organization
Chapter 2 - Fundamentals and Related Work
Fundamental Concepts
Related Studies
Chapter 3: A CEP Model for EPA Compositions
Introductory Aspects
Overview Diagram
Model Compartmentalization
Domain Layer
User Interface Layer
Application Layer
Infrastructure Layer
Chapter 4 - Evaluation
Expert Feedback: Description & Results
Experimental Study: Description & Results
Chapter 5 - Conclusion
Conclusion Description, Future Work
Appendix
Literature Review: Description & Results
Coupling Metrics Evaluation

Figure 1.1: Research Stages and Thesis Structure

The research planning depicted in Chapter 1 describes the study objectives, research problem, hypothesis for evaluation and methodology used.

Chapter 2 provides the fundamental CEP concepts and definition elements used throughout the work, and present papers identified with relevant contributions pertaining the composition of EPAs.

The solution is described in Chapter 3, being presented first in general terms. Due to its complexity, the overview diagram was compartmentalized, and components within each module were further detailed.

Then, the model was evaluated in Chapter 4 in distinct ways: according to the feedback collected from academia and industry experts, and based on an implementation of a real industry case experiment.

Finally, Chapter 5 presents the conclusions, limitations and future work.

2. Fundamentals and Related Work

This chapter presents the main concepts related to Complex Event Processing architectures and their challenges, relating them to the scope of this study.

2.1 Fundamental Concepts

CEP architectures are inherently built around their most fundamental element - the event, which is an occurrence within a particular domain, indicating something that has happened - or may have happened (*e.g.*, inference such as indication of a financial fraud or a network attack) in that domain. Three actors are the definition elements responsible for managing events (LUCKHAM, 2002):

- Event Producer - introduces into CEP platforms any form of data originated from external systems;
- EPA - performs processing of events introduced either by producer or other EPAs. It can be individually classified as:
  - *Filtering Agents*: filter events based on mechanisms like acceptance expressions or exclusion criteria;
  - *Transformation Agents*: perform operations, such as enrichment (complementing information) and aggregations over multiple occurrences; and,
  - *Pattern-Detection Agents*, whose pattern-detection operations may be perceived under different prisms, such as time (*e.g.*, measuring successive failures in components) or space (*e.g.*, recognition of objects in images);

- Event Consumer - consumes events derived from CEP platforms.

Those agents perform their tasks and interact with each other via mechanisms built according to additional CEP definition elements (LUCKHAM, 2002; ETZION; NIBLETT; LUCKHAM, 2011):

- Event Type - represents event semantics and guide decisions about the distribution of events among CEP components;
- Context - maintains a set of conditions of various dimensions (*e.g.*, temporal) providing a way to group correlated instances of events;
- Global State - refers to data from external sources that complements event information, such as reference data (for enriching events) and to system-wide global values (such as indicators for log level);
- Channel - provides event fetching and delivery capabilities for EPA, Event Consumer and Event Producer components. Figure 2.1 depicts how CEP agents interact with each other via this element.

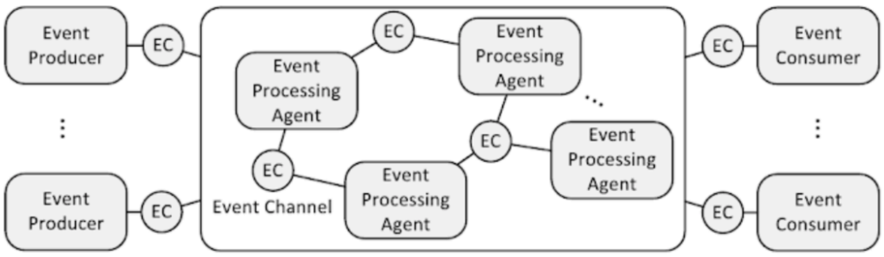


Figure 2.1: Interaction through Event Channel (EC) - adapted from (LUCKHAM, 2002)

Understanding how to elaborate CEP architectures, and how EPAs can be composed into solutions that fit industry needs remains a challenge - especially considering factors such as increasing data volumes, high frequencies, distributed processing, and complexity of composition patterns (LUCKHAM, 2002; PASCHKE; VINCENT, 2009; ETZION; NIBLETT; LUCKHAM, 2011; BINNEWIES; STANTIC, 2012; RAY; LEI; RUNDENSTEINER, 2016; OLLESCH; HESENIUS; GRUHN, 2017).

A case where complexity rises is processing a combination of activities in a segregated context - *e.g.*, detect which bank clients are making more than 3 withdrawals within a 24 hour interval (Figure 2.2). In this case, the events can be processed through context partitions (*i.e.*, event windows) applied on the flow, based on the `client id` and `time` (*e.g.*, 24 hours since a withdrawal transaction). Here, an important aspect is context sensitivity (ETZION; NIBLETT; LUCKHAM, 2011): infinite streams are usual in CEP scenarios, and processing of a continuous flow of events cannot always wait until the last occurrence manifests.

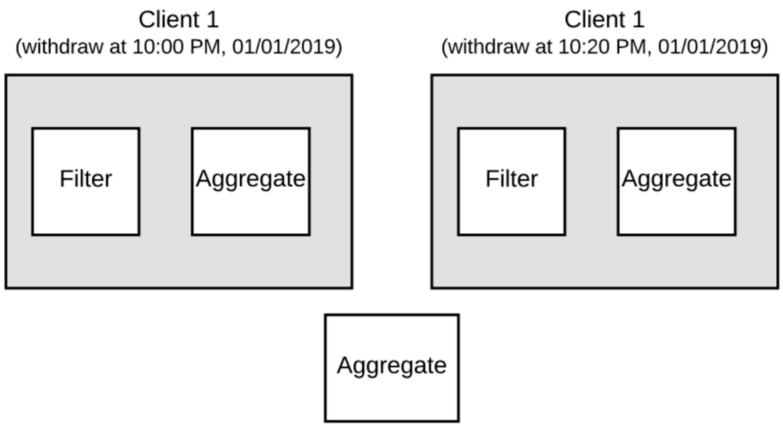

Client 1
(withdraw at 10:00 PM, 01/01/2019)
Filter
Aggregate
Client 1
(withdraw at 10:20 PM, 01/01/2019)
Filter
Aggregate
Aggregate

Figure 2.2: Aggregation of bank transactions

To perform all activities within the segmented context, EPA compositions representing the full operation needs to be mapped into a nested group of task-specific EPAs, all related to context partitions.

We conducted a literature review on ACM, IEEE Xplore, Scopus and Scholar digital libraries to investigate strategies for EPA compositions.

For this review, 2.754 distinct papers were selected out of 6.629 initial matches, scrutinized according to relevance of title and the tags to the theme. By filtering out articles based on abstract and inspecting further relevant tags, the number of matching articles became 394. By selecting relevant papers according to abstract and introduction text, matches went down to 129 After reading and analyzing the chosen papers, only 28 presented relevant information to support this research (CUGOLA; MARGARA, 2012; RENNERS; BRUNS; DUNKEL, 2012; ARTIKIS et al., 2012; LINDGREN; PIETRZAK; MÄKITAAVOLA, 2013; MARGARA; SALVANESCHI, 2013; MENDES; BIZARRO; MARQUES, 2013; STOJANOVIC et al., 2014; BAUER; WOLFF, 2014; NECHIFOR et al., 2014; BAUMGÄRTNER et al., 2015; PERERA; SUHOTHAYAN, 2015; KOLCHINSKY; SHARFMAN; SCHUSTER, 2015; KHARE et al., 2015; CARBONE et al., 2015;

VELASCO; MOHAMAD; ACKERMANN, 2016; BAPTISTA et al., 2016; RISCH; PETIT; ROUSSEAUX, ; RAY; LEI; RUNDENSTEINER, 2016; FALK; GURBANI, 2017; MAYER; MAYER; ABDO, 2017; DOBBELAERE; ESMAILI, 2017; D'SILVA et al., 2017; ICHINOSE et al., 2017; CANIZO et al., 2017; YADRANJIAGHDAM; YASROBI; TABRIZI, 2017; ZIMMERLE; GAMA, 2018; MANJUNATHA; MOHANASUNDARAM, 2018; DAYARATHNA; PERERA, 2018), complemented by 3 studies obtained by snowballing them, which were (MENDES; BIZARRO; MARQUES, 2008; PASCHKE; VINCENT, 2009; TEYMOURIAN; PASCHKE, 2010). Two of these papers (PASCHKE; VINCENT, 2009; TEYMOURIAN; PASCHKE, 2010) also served as a reference for 3 industry articles (BASS, 2006; MOXEY et al., 2010; ORACLE, 2010). So, we end up with a total of 34 publications as a result of this review. This activity revealed that the organization of EPA compositions remains an open question in the literature. However, some works present good descriptions of CEP components (BASS, 2006; PASCHKE; VINCENT, 2009; ORACLE, 2010; MOXEY et al., 2010; ETZION; NIBLETT; LUCKHAM, 2011) and were used in the model proposed in this theses, providing enlightenment for the elaboration and attribution of responsibility of CEP components.

The literature review is described in detail in Appendix A.

2.2 Related Work

This section presents studies that brought relevant contributions to understanding requirements for the composition of EPAs and, to a greater extent, to the elaboration of a model encompassing all CEP components and determining how they interrelate.

2.2.1 Conceptual Models

EPTS[1] elaborated a reference architectural model (PASCHKE; VINCENT, 2009) clarifying CEP common design patterns based on commonalities from three CEP industry models that have been described below.

- IBM Conceptual Model (MOXEY et al., 2010) provides a classification for CEP

[1]Event Processing Technical Society, or EPTS, promotes understanding and advancement in the field of event processing, and develop standards

components based on their main duties: *Emitter*: converts and packages event producer input data to deliver standardized events to the bus; *Event Bus*: receives events from emitters with potentially high frequencies, and delegates its processing to EPAs, aiming to derive a reduced amount of relevant events (to the business); *Event Handler*: provides mechanisms to deliver events from the bus to consumers.

Highlights: Similarities from Event Producer and EPA (event publishing via Bus), and Event Consumer and EPA (event consumption via Bus) were perceived on our study as an opportunity to enforce reusability while modelling those components. But the main contribution from this model is the idea of a conceptual "nested" architecture, where agents could contain within themselves a network of further agents that communicate through a segregated event bus.

- TIBCO BusinessEvents Model (BASS, 2006) provides contextual event responses at real time through the following architecture tasks: *Event Pre-processing*: for normalization, transformation, data cleansing on raw data; *Event Tracking*: for event identification and event pre-selection; *Situation Detection*: for identification based on relationships between events and historical data; *Predictive Analysis*: for impact assessment, *i.e.*, estimation of the impact of complex events on the organization and business processes; *Adaptive Business Process*: for dynamic adjustment of processes based on the overall processing architecture.

  Highlights: Common event pre-selection tasks based on "events of interest" were adopted in the model to provide simple filtering mechanisms for all EPA and Consumer nodes. Also, distributed event-driven architectures provide the underlying communication infrastructure to enable high performance event processing services.

- Oracle Complex Event Processor Model (ORACLE, 2010) provides a platform to process and analyze large scale real-time information via sophisticated correlation patterns based on: *Event sourcing strategy*: where multiple sources of information publish their updates as events, triggering processing mechanisms that consider event state changes; *Event-type schema*: define events from a tuple of attributes. *Pattern matching* based on event type: filter out redundant or irrelevant data and correlate meaningful events to infer critical decisions; *Continuous stream of events*: the base for processing. A query language can assist this task by defining the window of time for evaluation, event correlations and sequencing considerations.

Highlights: This solution incorporated event sourcing strategy for the interaction among EPA nodes within compositions via a segregated channel (structure apart from the channel used by Event Producers, Event Consumers and non-composite EPAs). Also, our proposal was influenced from the fact Oracle CEP solution has evolved to represent an EPN based on decoupled event driven applications, according to microservices architecture[2].

According to those models, an event processing node may be an individual event processing agent or an event processing network (EPN) (PASCHKE; VINCENT, 2009), consisting itself on a collection of event processing agents, producers, consumers, and global state elements connected by a collection of channel nodes (ETZION; NIBLETT; LUCKHAM, 2011). However no further implementation details nor clear guidance to design EPA compositions are provided. Therefore, designing cases where multiple EPAs interact under a complex context is not a trivial task. Also, they have been evaluated by Perera e Suhothayan (2015) according to their capacity to provide required functionalities needed for Realtime Streaming Analytic Solutions. The analysis pointed out a few gaps, such as interacting with historical data sources, and dynamically triggering a detailed analysis based on detected trends. Yet, those models contributed with meaningful design decisions to our model, which we have highlighted above.

Stream-based processing approach indicated on the following CEP models was also considered on the design of our proposal:

- A domain-specific reference model for text data analysis triggered by events related to document processing, considering semantics that correlate text blocks (verbs, nouns etc.) (BAUER; WOLFF, 2014);
- Another domain-specific reference model for controlling sensor and actuator devices, representing characteristics such as sensor attributes, equipment location (RENNERS; BRUNS; DUNKEL, 2012).

[2]https://docs.oracle.com/en/solutions/learn-architect-microservice/index.html#GUID-BDCEFE30-C883-45D5-B2E6-325C241388A5

2.2.2 Domain-Driven Design - Layers and Building Blocks

CEP scenarios in industry are reaching a complexity ceiling where next-generation platform technologies, such as IOT, automation systems, web API and other product-line architectures have become so complex that designers spend a long time mastering specific design patterns, and are often familiar with only a subset of the features they use regularly (SCHMIDT, 2006).

To approach the effort needed to correlate our CEP model components and compartmentalize them into cohesive units of analysis, we adopted principles of Domain-Driven Design (DDD) (EVANS, 2004) - a paradigm that solves problems associated with the composition of software components by: focusing on core domain aspects; and exploring models that emerge out of functional (business) and non-functional (technical) requirements via a common language within bounded contexts. By allowing us to focus separately on distinct requirements, and clarifying separation of duties, DDD strategy effectively met our specific needs.

In our case, a high-level abstract model representing CEP components was successively translated, via a systematic approach, into increasingly more detailed models, so that different modules concentrate on different parts of the design, following DDD strategy (EVANS, 2004). This needed to be accomplished without losing track of the integrated view, so that it did not violate key architectural principles, such as high cohesion and low coupling.

A contribution from DDD principles for compartmentalizing our model is the architecture layer division. In the following, we introduce each layer and describe their expected behaviour:

- Domain - Responsible for representing concepts and information about the business (entities, rules etc.). Technical details of storing it are delegated to the infrastructure;
- User Interface - Provides integration with external actors;
- Application - Defines the jobs the software is supposed to do and leverages domain objects to orchestrate meaningful tasks to the business;

- Infrastructure - Provides technical mechanisms to support higher layers: message sending for application, persistence for domain etc. It may support the interactions between the layers through an architectural framework.

In order to better characterize the components in terms of responsibility, the following design patterns, referenced as DDD building blocks, condenses a core of best practices from object-oriented domain modelling (EVANS, 2004) and were considered by this study:

- Entity: distinguishes objects by their unique identity, rather than attributes (`ID` generation algorithm can be challenging on concurrent processing scenarios);
- Value Object: represents descriptive aspects of the domain, is usually immutable (shared safely) and transient (discarded after an operation);
- Service: an operation offered as an interface, defined in terms of what it can do for a client. For instance, it may provide a subset of the attributes from an Entity, or a calculated value based on those, thus decoupling clients from Entity blocks. It also controls granularity, *e.g.*: medium-grained, stateless Service can be easier to reuse in large systems because they encapsulate significant functionality behind a simple interface, while fine-grained objects can lead to inefficient messaging in a distributed system;
- Event: represents an occurrence at a point in time as a domain object, and is usually immutable. In addition, it typically contains the identity of entities and value objects involved in the occurrence;
- Aggregate: groups related instances assisting them to keep track of its conceptually constituent parts;
- Repository: a collection of objects of a certain type as a conceptual set. It acts like a collection, except with more elaborate querying and persistence capabilities;
- Factory: Shifts the responsibility for creating instances of complex objects to a separate instance, which simplifies the creation process, and hides internal details.

2.2.3 Streaming Constructs

We adopted a stream processing strategy that allows parallel, asynchronous consumption of events (maximizing responsiveness to occurrences). This is a trend for strategies driven by a increasing number and capabilities of devices acting as distributed nodes (LINDGREN; PIETRZAK; MÄKITAAVOLA, 2013).

Benefits from the usage of Stream libraries, such as Java Stream and Flow APIs[3], are higher performance and minimized latency when applied to CEP solutions (BAUMGÄRTNER et al., 2015; ZIMMERLE; GAMA, 2018), obtained via:

- operations on a stream that produce a result without modifying its source - efficient data structures can be used for inference, validation and disposal of data, and storage is minimized;
- tasks that can be implemented in a lazy way, indicating opportunities for optimization, *e.g.*, operations such as `limit(n)` allow computations on infinite streams in finite time (KOLCHINSKY; SHARFMAN; SCHUSTER, 2015);
- pipelines of functional-style operations that can be composed through intermediate operations (which derive new streams) and, terminal operations (which produce a value that represents an aspect of a stream (*e.g.*, average value); and
- natural constructions for parallelism, if there are no sequential constraints for processing over multiple CPU cores.

In fully reactive stream scenarios, the producer may slow down to cope with consumer capabilities (*i.e.* via backpressure, based on reactive streams specifications, under java.util.concurrent.Flow). Khare et al. (2015) confirms the benefit of reactive Stream libraries (Microsoft .NET Reactive Extensions) as it integrates with OMG Data Distribution Service (DDS[4]), a publish/subscribe middleware suitable for industrial IoT applications.

But the reality for many CEP solutions is that original producers are fully detached components (such as sensors) that keep producing events at their own pace, therefore

[3]https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/stream/package-summary.html and https://docs.oracle.com/javase/9/docs/api/java/util/concurrent/Flow.html

[4]https://www.omg.org/omg-dds-portal/

excessive data that cannot be consumed immediately needs to buffered (up to a limit) or even dropped. This is the strategy for event correlation engines such as Esper[5] and Apache Flink[6], which combine event stream processing and CEP capabilities with its own event processing language, based on lambda constructs, to express filtering, aggregation and joins over event streams (STOJANOVIC et al., 2014; NECHIFOR et al., 2014). To avoid vendor-specific implementation details we adopted the following lambda expressions[7]:

- `Predicate<T>, BiPredicate<T,U>` - given T, or T and U arguments, returns a boolean value (useful on filtering operations);
- `Function<T,R>, BiFunction<T,U,R>` - given T, or T and U arguments, returns an instance of R (useful on transformation operations);
- `UnaryOperator<T>, BinaryOperator<T>` - given one or two instances of T, returns another instance of T (useful on reduce operations, to produce a single resultant T instance from a stream of T elements);
- `StreamConsumer<T>` - accepts a single T argument and returns no result (useful for pulling events). This was renamed to distinguish it from EventConsumer.

Notation for related elements is also indicated in the model:

- `Optional<T>` - handle cases where the T instance may be nullable[8];
- `Interface private implementation` method[9], for code re-usability and modularization;
- `Template parameter` notation indicates (in squares) generic[10] parameter type associations for dynamic type casting, as further explained in Section 3.8.

Although Java notation was used, this model can be adapted for any language with stream processing capabilities (C#, C++, Scala, etc.), or any that integrates with streaming

[5]http://www.espertech.com
[6]https://flink.apache.org
[7]https://docs.oracle.com/javase/10/docs/api/java/util/function/package-summary.html
[8]https://docs.oracle.com/javase/10/docs/api/java/util/Optional.html
[9]https://docs.oracle.com/javase/specs/jls/se9/html/jls-9.html
[10]https://docs.oracle.com/javase/tutorial/java/generics/inheritance.html

computation engines (*e.g.*, Spark, Storm, Flink, Kafka Streams). We chose Java due to its relevance in industry - 52% of CEP commercial systems have been implemented using Java (DAYARATHNA; PERERA, 2018).

2.2.4 Realtime Streaming Analytics Patterns

Realtime Streaming Analytics Sytems need to process data as events raise, allowing continuous flow analysis. It accepts one or more data streams, consisting of several time-bounded events as input, and produces one or more data streams as output. Thirteen real-time analytics patterns (PERERA; SUHOTHAYAN, 2015) address the need of a common shared understanding, among practitioners, on requirements for analytics use cases from Complex Event Processing systems:

1. Preprocessing - performed as a projection from one data stream to the other, or through filtering;
2. Alerts and Thresholds - detects a condition and generates alerts based on a simple value or more complex conditions such as rate of increase;
3. Simple Counting and Counting within Segmented Windows - includes aggregate functions (Min, Max, Percentiles etc.) which can be calculated in isolation (*e.g.*, counting failed transactions) or under a time window attached to it (*e.g.*, failure count in the last hour);
4. Joining Event Streams - combines multiple data streams and create a new event stream (using pattern for joining operations);
5. Data Correlation, Missing Events, and Erroneous Data - Match events from different streams, detects missing events in a data stream, and use redundant data to find erroneous events and remove them from further processing;
6. Interacting with Databases - combines real-time data against historical data persisted in a data source;
7. Detecting Temporal Event Sequence Patterns - given a sequence of events, we can write a regular expression to detect a temporal sequence of events arranged on time where each event conforms to a given acceptance criteria;

8. Tracking - tracks objects over space and time and detects given conditions. For instance, certifying all cars adhere to routes and speed limits;
9. Detecting Trends - Detects patterns from time series data and brings them into operator attention;
10. Running the Same Processing Mechanisms in Batch and Realtime Pipelines - combines batch processing tools and on-line processing solutions such as CEP;
11. Detecting and Switching to Detailed Analysis - detects a condition that suggests some anomaly, and further analyze it using historical data. This is used for use cases where we cannot analyze all the data in full detail. Instead, only anomalous cases are viewed in full detail;
12. Using a Model - train a model (often via Machine Learning), and then use it with the Realtime pipeline to make decisions;
13. Online Control - automatizes control - *e.g.*, autopilot and self- driving. Involves problems, like situation awareness and predicting next value.

2.2.5 Industry Processing Requirements

Due to the industry requirements to process large volumes of ingested data in a short period of time, the following aspects have been pointed out on related studies as critical for CEP solutions (CARBONE et al., 2015; D'SILVA et al., 2017; FALK; GURBANI, 2017; MAYER; MAYER; ABDO, 2017).

- Support for batch introduction of events: based on a "store and forward" approach (telecommunications technique to tolerate delay among network nodes) as suggested in an architecture (D'SILVA et al., 2017) that uses Apache Kafka for online and offline consumption and Apache Spark to process streams. Apache Flink (CARBONE et al., 2015) also presents its own strategy for batch introduction via its stream processing API. Our model, not tied to industry solutions, has an offline representation of Event Producer to pull batches of events from a Global State (Section 3.5.1).

- Advanced Filtering: A proposal to move the extraction-transformation-load (ETL) activities further into the event processing pipelines alleviates latency problems whenever constraints prevent us from filtering data at event producers (as it would impact their capabilities) (FALK; GURBANI, 2017). In our study, filtering and projecting data allow analytics platforms to process higher workloads. Subscription criteria containing event types, where each one includes relevant filter attributes, supports the implementation of this aspect in our model.

- Incremental Model Training: our CEP model incorporates Machine Learning capabilities to improve pattern recognition accuracy during EPA processing, based on a study that explores continual learning methodology (MAYER; MAYER; ABDO, 2017) (Section 3.6.1.4). This is built from the idea of adapting prediction models according to evolving data distributions that reflect the external world. It presents challenges in terms of scalability to process ever increasing data volumes via sustainable computing resources (*e.g.*, CPU and memory).

2.3 Summary

This chapter presented innovative design strategies we leveraged for establishing EPA compositions according to organized structures and best practices (such as streaming, advanced filtering and incremental model training). Those are not covered in existing CEP models (LUCKHAM, 2002; BASS, 2006; PASCHKE; VINCENT, 2009; ORACLE, 2010; MOXEY et al., 2010; ETZION; NIBLETT; LUCKHAM, 2011). Therefore, we indicate our model as an enhancement over those, since it leverages existing representations, but also introduces relevant features for realtime streaming analytics solutions.

Table 2.1 organizes the concepts and consolidates relevant knowledge currently scattered over different models and papers.

Table 2.2 consolidates all mapped gaps we aim to address in our proposed model in an overview, consisting of 17 requirements, that encompasses: EPA composition, as depicted in Section 2.1; the patterns from Section 2.2.4 related to realtime streaming analytics; and industry CEP requirements from Section 2.2.5. It also correlates each requirement to capabilities from existing CEP models, according to studies from Paschke e Vincent

(2009), Cugola e Margara (2012), Perera e Suhothayan (2015) and available documentation (BASS, 2006; MOXEY et al., 2010; ORACLE, 2010).

Table 2.1: Reference Studies

Contribution	**References**
Fundamental concepts and data structures	(LUCKHAM, 2002; BASS, 2006; MOXEY et al., 2010; ORACLE, 2010; ETZION; NIBLETT; LUCKHAM, 2011; ARTIKIS et al., 2012)
Segregation of responsibilities	(EVANS, 2004; BASS, 2006; PASCHKE; VINCENT, 2009; MOXEY et al., 2010; ORACLE, 2010; TEYMOURIAN; PASCHKE, 2010; ETZION; NIBLETT; LUCKHAM, 2011)
Handling high volumes of distributed data	(BASS, 2006; ORACLE, 2010; MOXEY et al., 2010; SHARP et al., 2013; PERERA; SUHOTHAYAN, 2015; RAY; LEI; RUNDENSTEINER, 2016; NADAREISHVILI et al., 2016; FALK; GURBANI, 2017; DOBBELAERE; ESMAILI, 2017)
Offline batch processing of events	(CARBONE et al., 2015; D'SILVA et al., 2017; VIDYASANKAR, 2017)
Stream processing on CEP	(MARGARA; SALVANESCHI, 2013; CARBONE et al., 2015; KHARE et al., 2015; BAUMGÄRTNER et al., 2015; ZIMMERLE; GAMA, 2018; DAYARATHNA; PERERA, 2018)
Machine Learning capabilities	(MAYER; MAYER; ABDO, 2017)
Implementation cases and metrics	(CUGOLA; MARGARA, 2012; RISCH; PETIT; ROUSSEAUX, ; VELASCO; MOHAMAD; ACKERMANN, 2016; BAPTISTA et al., 2016; ICHINOSE et al., 2017; CANIZO et al., 2017; YADRANJIAGHDAM; YASROBI; TABRIZI, 2017; MANJUNATHA; MOHANASUNDARAM, 2018)

Table 2.2: Requirements and correlation to existing models

Requirement	**EPTS Reference Architecture**	**IBM Conceptual Model**	**TIBCO Business Events**	**Oracle CEP**
Requirements from Section 2.1				
R1 - A strategy for EPA composition	No	No	No	No
Requirement from Section 2.2.4				
R2.1 - Support for batch introduction of events	No	Yes	No	No
R2.2 - Advanced Filtering	No	No	No	Yes
R2.3 - Incremental Model Training	Yes	Yes	Yes	Yes
Requirement from Section 2.2.5				
R3.1 - Preprocessing	Yes	Yes	Yes	Yes
R3.2 - Alerts and Thresholds	No	Partial*	Yes	Partial*
R3.3 - Simple Counting and Counting within Segmented Windows	Yes	Yes	Yes	Yes
R3.4 - Joining Event Streams	Yes	Yes	Yes	Yes
R3.5 - Data Correlation, Missing Events, and Erroneous Data	Yes	Yes	Yes	Yes
R3.6 - Interacting with Databases	No	No	Yes	No
R3.7 - Detecting Temporal Event Sequence Patterns	Yes	Yes	Yes	Yes
R3.8 - Tracking	Yes	Yes	Yes	Yes
R3.9 - Detecting Trends	Yes	Yes	Yes	Yes
R3.10 - Running the Same Processing Mechanisms in Batch and Realtime Pipelines	No	No	Yes	No
R3.11 - Detecting and Switching to Detailed Analysis	No	No	No	Yes
R3.12 - Using a Model	Yes	Yes	Yes	Yes
R3.13 - Online Control	No	No	No	No

* Applies to models that indicate the possibility to trigger alerts based on simple threshold conditions, but not for more elaborated circumstances such as rate of increase

3. A CEP Model for EPA Compositions

3.1 Introductory Aspects

This section introduces our CEP model proposal. It consolidates features from existing CEP models by identifying and isolating common behaviours of Event Producer, Event Consumer and EPA (Section 3.6.1), as well as by indicating segregation of responsibilities through the usage of Channels, Event Fetchers and Event Emitters (Section 3.7.1), which support interaction among CEP components.

3.2 Overview

Figure 3.1 presents an overview of the model. Definition elements (highlighted in grey) are presented under the context of the DDD layers, depicted within a UML stereotyped package notation to denote their affinity to the purpose of the layer.

According to the expected purpose of domain layer components, described in Section 2.2.2, the following CEP Definition Elements (and dependencies) are represented as members within this layer: Event & Event Type, driving event processing; and Context, correlating processing of event instances over dimensions.

`Event` represent occurrences of real world or simulations. Such instances are classified according to `EventType`, which determines the type of event originally produced (or derived). For instance, event types may correspond to a sensor update, a money deposit, or a fraud indication alert. Event data values captured during occurrences is maintained as `EventAttribute` (e.g., amount of money withdrawn). Those entities were described

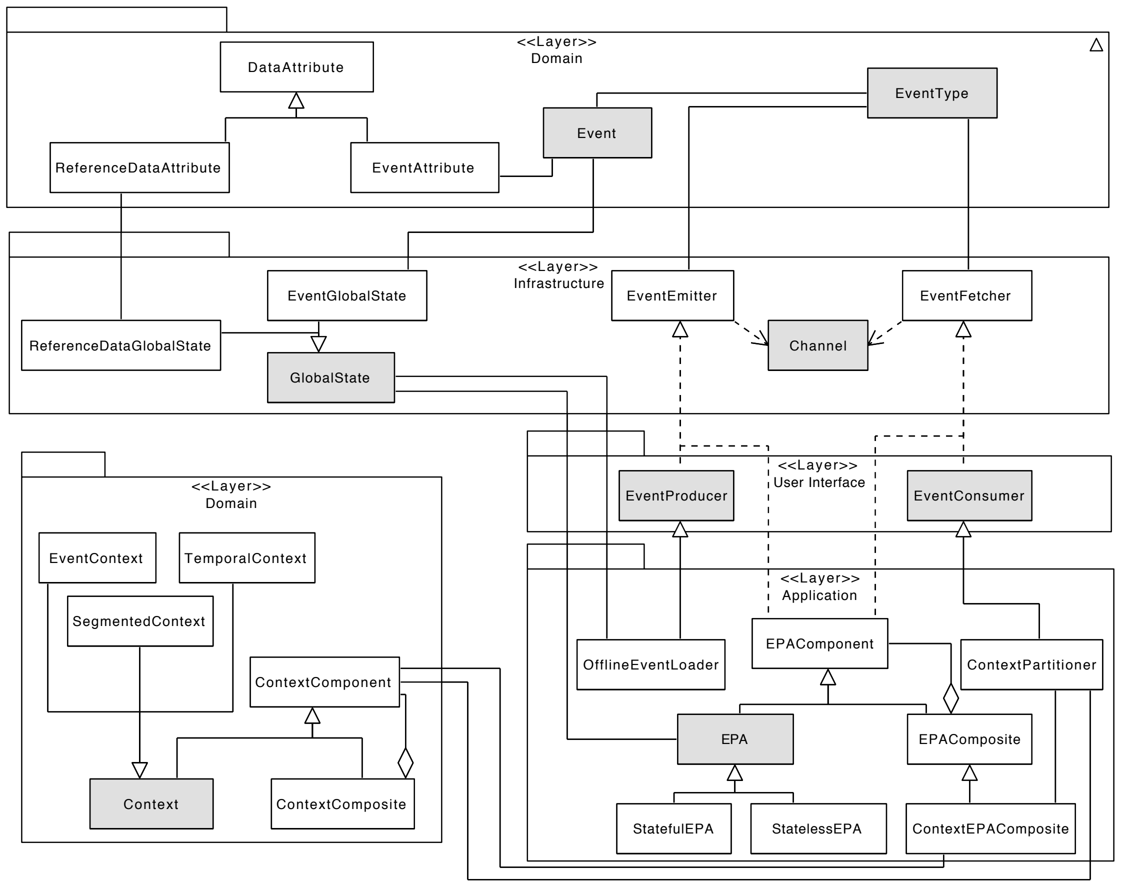


Figure 3.1: CEP Model - High Level Diagram

in Section 3.4.1.

In addition, a domain component named `Context` defines how instances of events can be associated, as described in Section 3.4.2. Events are correlated by EPA nodes that consider partitions of based on event attributes. This provide meaningful aggregation features for event processing based on timeframe, on historical events, or on any attribute (respectively via `TemporalContext`, `EventContext` and `SegmentedContext`). Also, `ContextComponent` and `ContextComposite` were designed for Context compositions: ContextComponent provides the abstraction for partitioning events into processing groups, and is used by `ContextPartitioner` as it dynamically provides instances of `ContextEPAComposite` for such activity (as described in Section 3.6.2).

`EventProducer` and `EventConsumer` represent interaction of systems (usually external to the scope of our model) with CEP solutions, that feed or consume data through specific mechanisms. As such, they fulfill the responsibility assigned to user interface components, as per Section 2.2.2. Event occurrences are originally introduced into the

CEP systems via EventProducer nodes (Section 3.5.1). Their output may be referenced as raw events, as opposed to derived events, created by EPAs. EventConsumer entities, on the other end, consume events from CEP platforms (Section 3.5.2).

`EPAs` are the heart of application layer, since they are orchestrated to process events according to business goals. They share common behaviors with EventConsumer (consumption via channel) and EventProducer (publishing via channel). This aspect is represented in this model via interface multiple inheritances (from `EventFetcher` and `EventEmitter` infrastructure interfaces). `EPAComposite` abstracts the functionality of a composed EPA by delegating its processing to inner components (Section 3.6.1).

EPAs can provide different processing strategies. For instance, they may process events isolated from other events (which happens in `StatelessEPA`), or occurrences can be correlated (by `StatefulEPA`), for which comparisons may need to be considered within partitions of incoming events.

Two Components, positioned under infrastructure layer, allow interaction among elements from higher layers, supporting their requirements via technical mechanisms to achieve: message sending (for services), persistence and information retrieval for shared data; etc.: `Channel`, which brings mechanisms for interacting with the distribution platform; and `GlobalState`, with mechanisms for integration with external data sources.

The event distribution platform allows event retrieval through topic subscription patterns. Access to this functionality and communication to and from this platform, is represented by the Channel abstraction provided in infrastructure layer, described in Section 3.7.1.

GlobalState provides EPA access to more information than what can be obtained from each event (see Section 3.8). `ReferenceDataGlobalState` is able to map information from data stores into `ReferenceDataAttributes` instances, whereas `EventGlobalState` can be used to pull event batches that can be introduced to CEP platforms through scheduling tasks provided by `OfflineEventLoader` (see Section 3.6.3).

3.3 Model Compartments

Partitions of our model (referred to as sub-model) are defined by considering the assignment of components into cohesive units of analysis, according to DDD layers (Section 2.2.2) and the overview model (Section 3.2). As highlighted in the overview, most relevant entities on each sub-model are marked in grey (entities marked in white are detailed in another sub-model, that best meets their responsibilities).

Unified Modeling Language (UML) provides a good amount of flexibility to represent compositions of software components, plus it is well known among IT professionals (THÖNE; DEPKE; ENGELS, 2002). Its stereotypes were used to indicate design patterns presented in Chapter 2, leveraging its extensibility mechanisms to better clarify the meaning of those classes. For instance, applicable patterns related to DDD building blocks (Section 2.2.2) were prefixed with ”ddd_”. Java Stream Lambda Constructs (from Section 2.2.3) were indicated along with components from this model.

3.4 Domain Layer

This layer is responsible for representing concepts and information about the business (*e.g.*, entities and rules). Technical details of storing it are delegated to the infrastructure (EVANS, 2004).

3.4.1 Event & Event Type

`Event` represents occurrences captured from real world instances or inferred from simulated scenarios. Those may happen in the form of updates triggered by monitoring processes, operation exceptions, alerts indicating abnormal behavior etc.

The diagram in Figure 3.2 represents elements related to an Event instance:

- `EventHeader` maintains characteristics of the event instance (such as occurrence time), and can be recognized by an event processor which may not need to understand the remainder of the event structure. It holds an instance of:

  - `EventType`, carrying a set of filter attributes (`filterAttributes`) containing payload attribute names. Subscribers can define filters based on those attributes. This can be helpful when high volumes of data are involved, since we can reduce the content amount to bring just the needed business attributes in the `EventPayload`, which relates to requirements R2.2 and R3.1 from Table 2.2;
  - `EventEmmiter`, a shallow clone of the originator instance (`EventProducer`, for introduced events, or `EPA`, for derived events), helpful for further filtering, relating to requirement R3.1 from Table 2.2;

- `EventPayload` contains values for a set of EventAttribute (which extends the DataAttribute - Section 3.4.3 with a *derived* attribute that indicates if this instance has been originally produced by the source system or derived by CEP);
- `EventOpenContent` allows to store a free-format annotation, *e.g.*, for external systems information. It is an optional attribute.

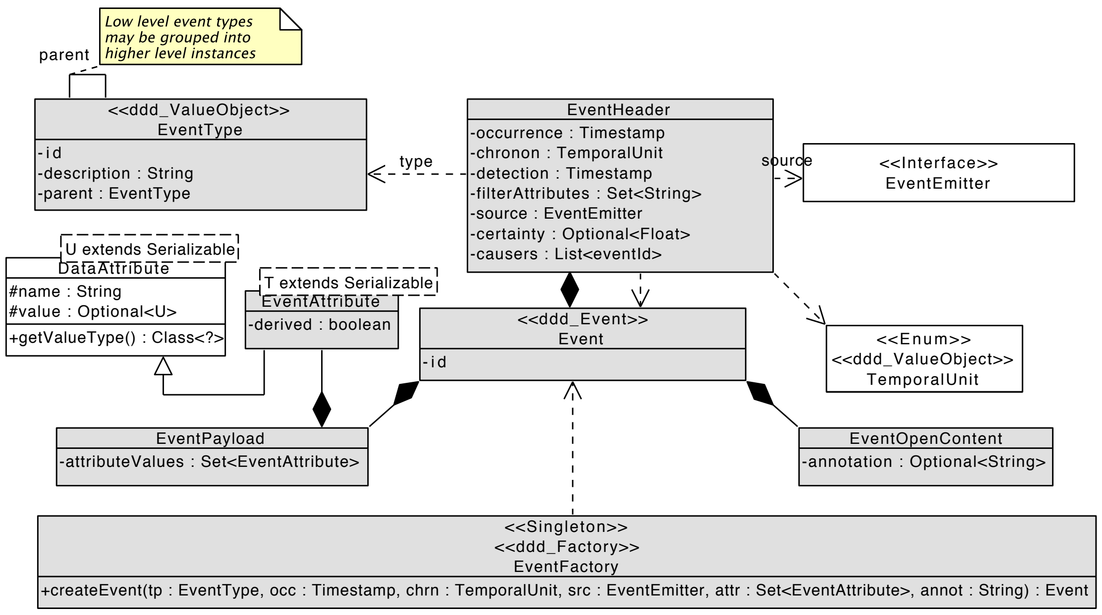

Low level event types may be grouped into higher level instances
parent
<<ddd_ValueObject>> EventType
-id
-description : String
-parent : EventType
type
EventHeader
-occurrence : Timestamp
-chronon : TemporalUnit
-detection : Timestamp
-filterAttributes : Set<String>
-source : EventEmitter
-certainty : Optional<Float>
-causers : List<eventId>
source
<<Interface>> EventEmitter
U extends Serializable
DataAttribute
#name : String
#value : Optional<U>
+getValueType() : Class<?>
T extends Serializable
EventAttribute
-derived : boolean
<<ddd_Event>> Event
-id
<<Enum>> <<ddd_ValueObject>> TemporalUnit
EventPayload
-attributeValues : Set<EventAttribute>
EventOpenContent
-annotation : Optional<String>
<<Singleton>> <<ddd_Factory>> EventFactory
+createEvent(tp : EventType, occ : Timestamp, chrn : TemporalUnit, src : EventEmitter, attr : Set<EventAttribute>, annot : String) : Event

Figure 3.2: Event Diagram

Events can be created by Event emitters via a common implementation artifact named `EventFactory`. They are supposed to provide data for the definition of all event information. The factory instantiates Event and all composed parts (at least a Header and a Payload) based on incoming parameters, and provides dynamic values, such as the time at which the event enter the CEP platform. A Singleton (GAMMA et al., 1995) stereotype suggests a unique instance, which can be accomplished via a decoupled service.

Tables from Section B.1 on Appendix B outline attributes and methods related to model components from this section.

3.4.2 Context

A `Context` represents the conditions that are used to group event instances so that they can be processed in a related way (ETZION; NIBLETT; LUCKHAM, 2011).

Hierarchies of Context (Figure 3.3) allow the possibility of composing EPAs to express scenarios such as raising an alert if someone attempts to make more than three withdraws within a 24-hour period. We can use EPA agents related to windows that start whenever a given customer withdraws money from an ATM machine, and close 24 hours later. Compositions of Context through Window abstractions are depicted in Section 3.6.2.

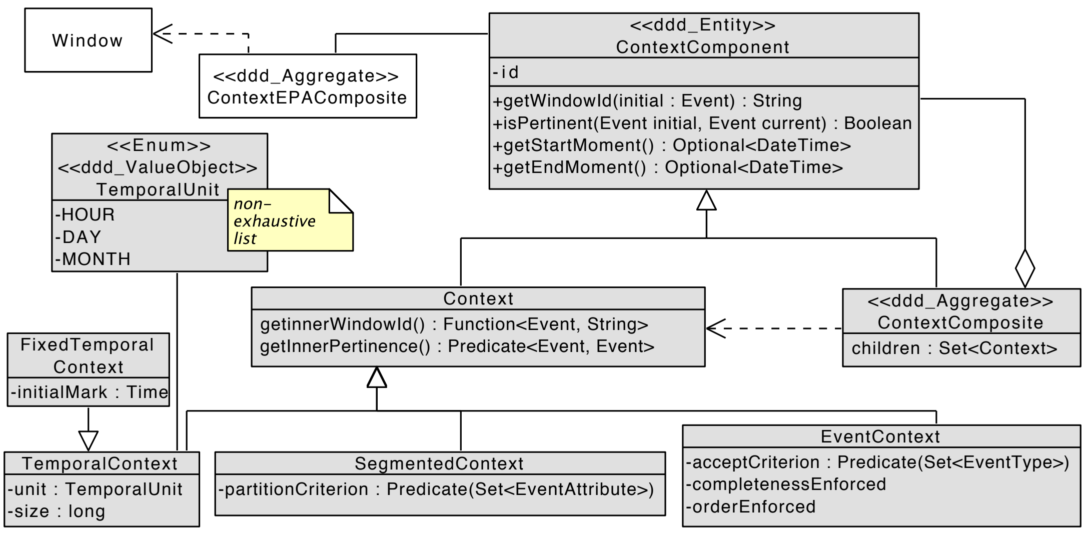

Window
<<ddd_Aggregate>> ContextEPAComposite
<<ddd_Entity>> ContextComponent
-id
+getWindowId(initial : Event) : String
+isPertinent(Event initial, Event current) : Boolean
+getStartMoment() : Optional<DateTime>
+getEndMoment() : Optional<DateTime>
<<Enum>> <<ddd_ValueObject>> TemporalUnit
-HOUR
-DAY
-MONTH
non-exhaustive list
Context
getinnerWindowId() : Function<Event, String>
getInnerPertinence() : Predicate<Event, Event>
<<ddd_Aggregate>> ContextComposite
children : Set<Context>
FixedTemporal Context
-initialMark : Time
TemporalContext
-unit : TemporalUnit
-size : long
SegmentedContext
-partitionCriterion : Predicate(Set<EventAttribute>)
EventContext
-acceptCriterion : Predicate(Set<EventType>)
-completenessEnforced
-orderEnforced

Figure 3.3: Context Diagram

A context may consist on one or more instances of the following types:

- `TemporalContext` - a window based on this partition comprises events that happen during its time frame, and every event starts a new window lasting an amount of computed time based on TemporalContext size and TemporalUnits;
- `FixedTemporalContext` - similar to TemporalContext, but an initial mark con-

stitutes the Window start (e.g. 00:00 hours, instead of the original occurrence time). Hence, successive events within the same time range are assigned to the same Window and will not trigger a new one;

- `SegmentedContext` - creates windows based on attribute values, where each distinct combination of values relates to a different window;

- `EventContext` - creates windows based on a list of event types that represents the occurrence of a scenario of interest to the business.

Composed Context, are used along with ContextEPAComposite (Section 3.6.2) to provision components that form the basis for event grouping, relating to requirement R3.3 from Table 2.2.

Tables from Section B.2 on Appendix B outline attributes and methods related to model components from this section.

3.4.3 Data Attribute

The `DataAttribute` (Figure 3.4) presents the minimal common structure to represent relevant business data (such as the attributes within event payload) or technical information (*e.g.*, maximum timeout value). Its template parameter indicates the usage of dynamic type casting to explicitly restrict U to Serializable types, so that the state of objects can be converted and restored as needed.

Two subtypes are provided: `EventAttribute`, referenced in Section 3.4.1, and `ReferenceDataAttribute` from Section 3.8.

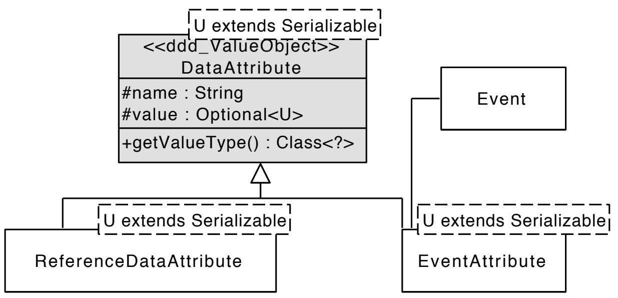


Figure 3.4: Data Attribute Diagram

Tables from Section B.3 on Appendix B outline attributes and methods related to model components from this section.

3.5 User Interface Layer

This layer provides integration with external actors, which might sometimes be another computer system rather than a human user (EVANS, 2004).

3.5.1 Event Producer

The `EventProducer` representation is designed according to its relevant responsibility assignment (LUCKHAM, 2002; ETZION; NIBLETT; LUCKHAM, 2011) : inject event data into CEP Platform through publishing operation, via a Channel instantiated through ChannelBroker, as shown in Figure 3.5.

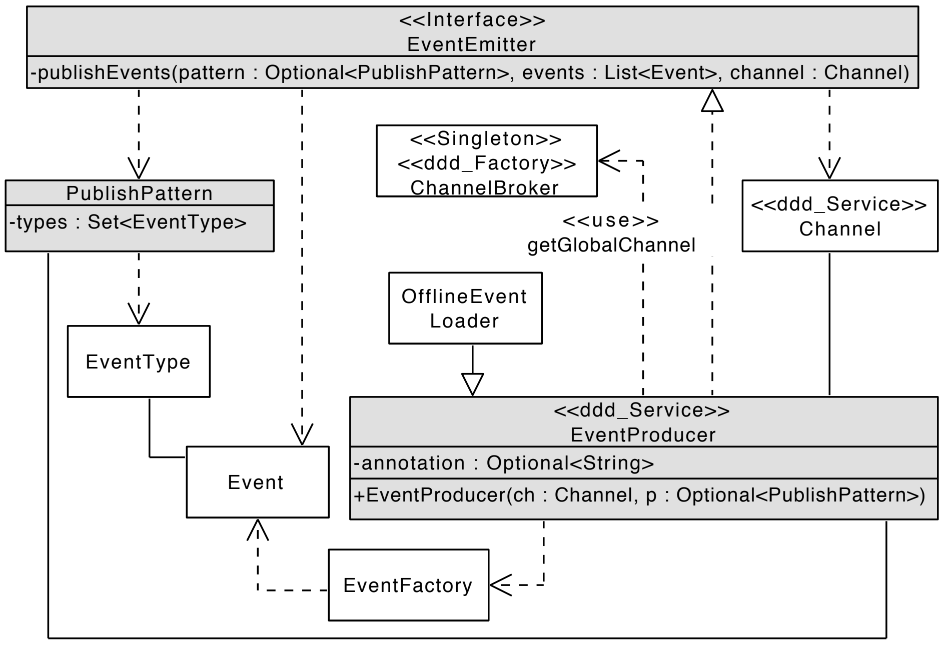


Figure 3.5: Event Producer Diagram

Events are published by an EventProducer according to the `EventEmitter` interface. This interface has a private implementation (see Section 2.2.3) that validates output events based on a Set of EventTypes - indicated on `PublishPattern`, which precedes the introduction of events, relating to requirement R3.1 from Table 2.2 - and delegates publishing of derived events to a Channel instance.

Producer instances relate to external systems, sometimes with an implicit logic to generate the events, whereas an EPA's logic is always explicit (since it is always an internal CEP component). The absence of fetching mechanisms is another difference between them.

An EventProducer has an extension named `OfflineEventLoader` that provides means for offline introduction of events via batch mechanisms, as described in Section 3.6.3.

Tables from Section B.4 on Appendix B outline attributes and methods related to model components from this section.

3.5.2 Event Consumer

The `EventConsumer` node (Figure 3.6) represents services capable of pulling event data from CEP platform. This operation is supported by a Channel, instantiated from ChannelBroker.

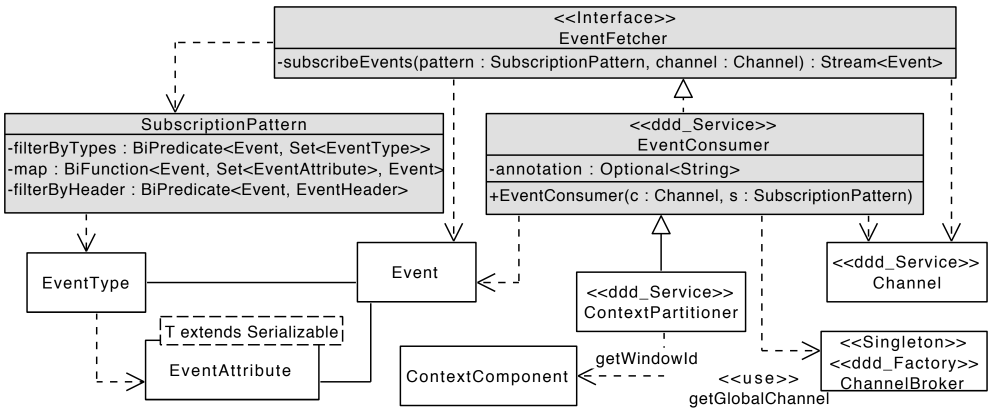

<<Interface>>
EventFetcher
-subscribeEvents(pattern : SubscriptionPattern, channel : Channel) : Stream<Event>
SubscriptionPattern
-filterByTypes : BiPredicate<Event, Set<EventType>>
-map : BiFunction<Event, Set<EventAttribute>, Event>
-filterByHeader : BiPredicate<Event, EventHeader>
<<ddd_Service>>
EventConsumer
-annotation : Optional<String>
+EventConsumer(c : Channel, s : SubscriptionPattern)
EventType
Event
<<ddd_Service>>
ContextPartitioner
<<ddd_Service>>
Channel
T extends Serializable
EventAttribute
ContextComponent
getWindowId
<<use>>
getGlobalChannel
<<Singleton>>
<<ddd_Factory>>
ChannelBroker

Figure 3.6: Event Consumer Diagram

Event consumption is performed through `EventFetcher` interface private implementation (subscribeEvents), by inspecting events according to `SubscriptionPattern`. This pattern enables simple filtering and projection operations for EPA subclasses. SubscriptionPattern guides the Channel to bring data from topics, partitions or segments (deppending on features from distribution platforms). Filtering relates to: EventType (filterByTypes) or EventHeader (filterByHeader) - both based on a BiPredicate acceptance criteria used for Event data inspection. It also provides a BiFunction map implementation for reducing event content size on filtered occurrences, based on relevant EventAttributes.

Using strategies to position Channel to pull subsets of messages closer to distribution platform, this can minimize latency in data traffic (FALK; GURBANI, 2017), which relates to requirements R2.2 and R3.1 from Table 2.2. BiPredicate and BiFunction, among other streaming constructs, were described in Section 2.2.3.

The processing activities for this component are in many cases out of the scope of CEP components, since they relate to external systems, *e.g.*, dashboard and reports from closer-to-business-goals events derived by EPAs. An exception case is an internal implementation of an EventConsumer named `ContextPartitioner`. It allows a continuous stream of events from the GlobalChannel to be partitioned for processing, providing means for dynamic EPA Compositions based on context (as documented in Section 3.6.2).

Tables from Section B.5 on Appendix B outline attributes and methods related to model components from this section.

3.6 Application Layer

This layer defines the jobs the software is supposed to do and leverages domain objects to orchestrate meaningful tasks to the business (EVANS, 2004).

3.6.1 Event Processing Agent

Event Processing Agents (EPA) perform event processing tasks and monitor events to detect certain patterns of information. When patterns match, they trigger actions that output events.

This section presents a generic representation that addresses aspects related to the strategy used for the composition of EPA, and also the interaction of this element with other CEP definition elements (ETZION; NIBLETT; LUCKHAM, 2011; LUCKHAM, 2002). Due to the complexity to represent details from all EPA derivations, according to their distinct features, we chose present a low degree of detail at first, as in Figure 3.7 and detail it further on the following sections.

Similarities identified on EventConsumer and EPA motivated the creation of Event-

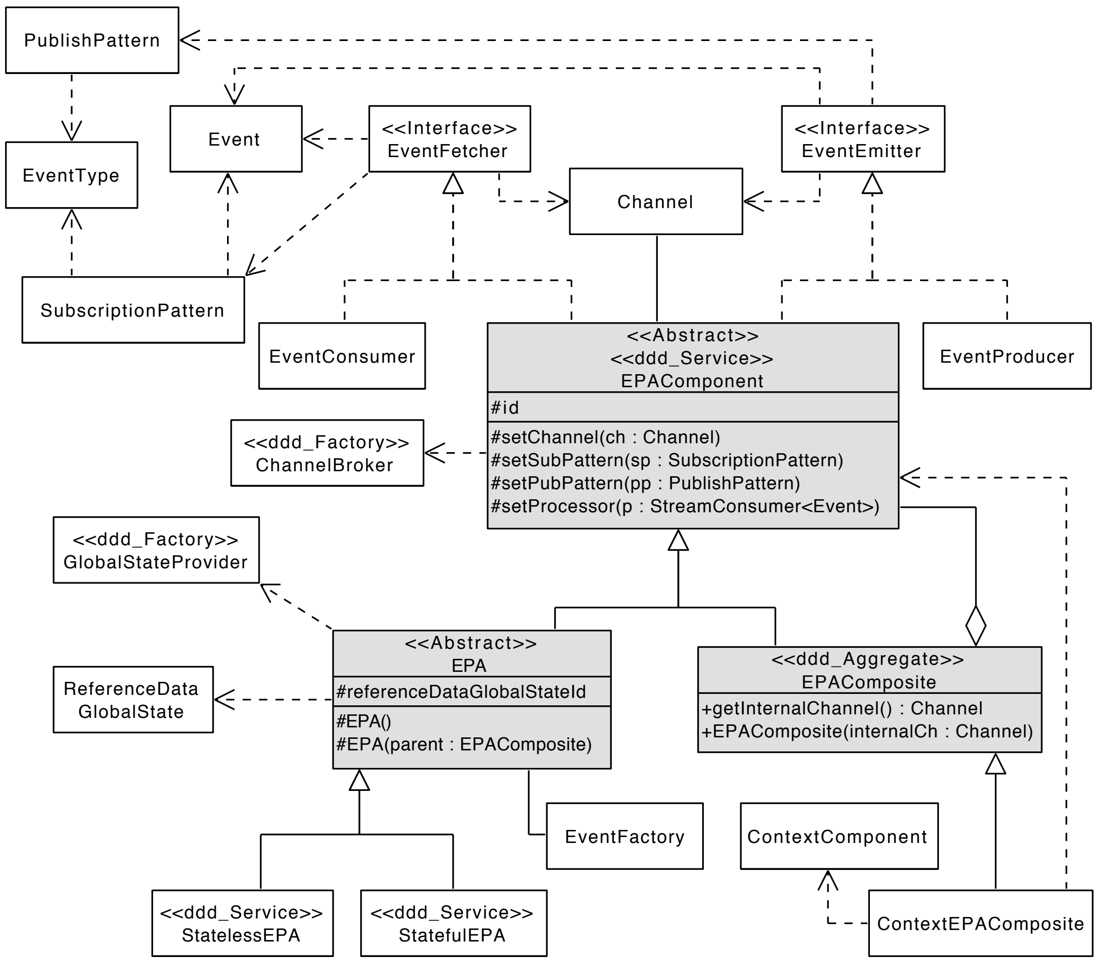

PublishPattern
EventType
Event
<<Interface>> EventFetcher
<<Interface>> EventEmitter
Channel
SubscriptionPattern
EventConsumer
<<Abstract>> <<ddd_Service>> EPAComponent
#id
#setChannel(ch : Channel)
#setSubPattern(sp : SubscriptionPattern)
#setPubPattern(pp : PublishPattern)
#setProcessor(p : StreamConsumer<Event>)
EventProducer
<<ddd_Factory>> ChannelBroker
<<ddd_Factory>> GlobalStateProvider
<<Abstract>> EPA
#referenceDataGlobalStateId
#EPA()
#EPA(parent : EPAComposite)
ReferenceData GlobalState
<<ddd_Aggregate>> EPAComposite
+getInternalChannel() : Channel
+EPAComposite(internalCh : Channel)
EventFactory
ContextComponent
<<ddd_Service>> StatelessEPA
<<ddd_Service>> StatefulEPA
ContextEPAComposite

Figure 3.7: EPA Overview Diagram

Fetcher interface (with common behaviors). Analogously, EventProducer and EPA generate events via EventEmitter interface, in this case using EventFactory.

`EPAComponent` defines the common behavior for EPAs (designed as per composite pattern (GAMMA et al., 1995)) - it fetches events based on SubscriptionPattern (see Section 3.5.2), establishes a communication channel for interactions with CEP actors (producer, consumer and other EPAs), and delivers events according to PublishPattern (see Section 3.5.1).

`EPAComposite` aggregates EPAs, and delegates its processing rules to an organized set of `EPAs`. Operations within EPA composition are isolated from global processes (ETZION; NIBLETT; LUCKHAM, 2011), and here they are performed using a segregated channel (not the global channel), as presented in Figure 3.8. In this picture, we can observe two EPAs (Claim Filter and Enricher) that interact directly with the global channel. We also have two EPAs (Aggregate and Content Filter) that present interactions constrained to operate via an inner channel, and an EPAComposite, which encompasses both. The latter intermediates access, for those inner EPAs, *from* the global channel (delivering subscribed events from global channel - based on its SubscriptionPattern - to its inner channel) and *to* the global channel (delivering events produced by those to the global channel, according to its PublishPattern). Both global and inner channels are provided by a ChannelBroker (see Section 3.7.1). This addresses the gap that relates to requirement R1, from Table 2.2.

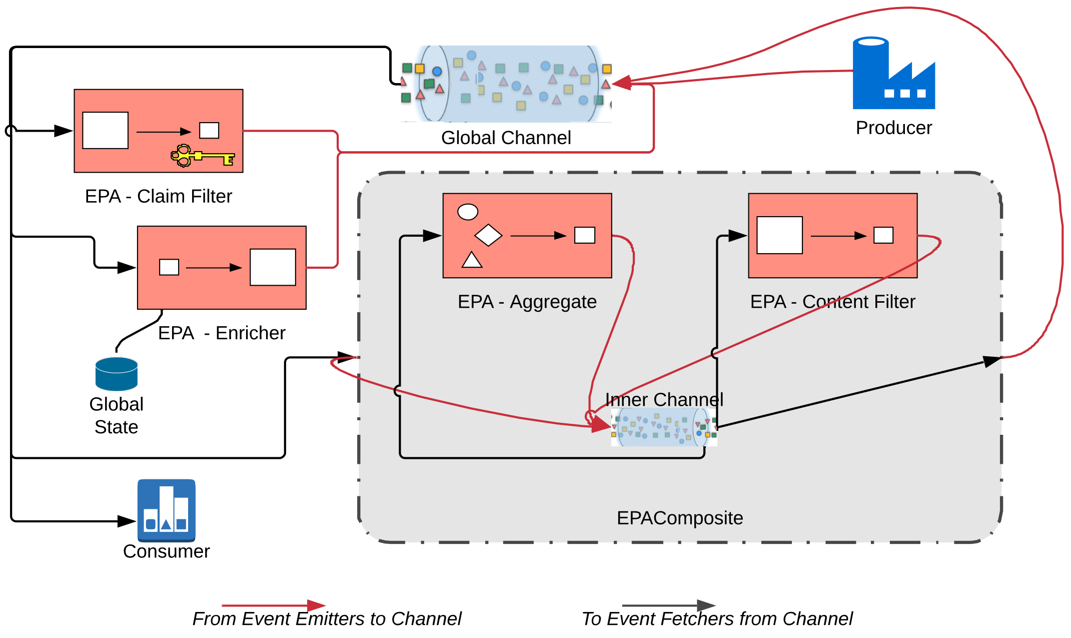


Figure 3.8: EPA Composition

Event streams are usually infinite feeds, but operations may be defined in terms of partitions (known as event windows). `ContextEPAComposite` are EPA compositions dynamically provisioned based on window partitions, and are further detailed in Section 3.6.2.

Also, a GlobalState element is used by EPAs to fetch data outside the scope of events (*e.g.*, business data for enriching events). This is performed by `ReferenceDataGlobalState`, further demonstrated in Section 3.8.

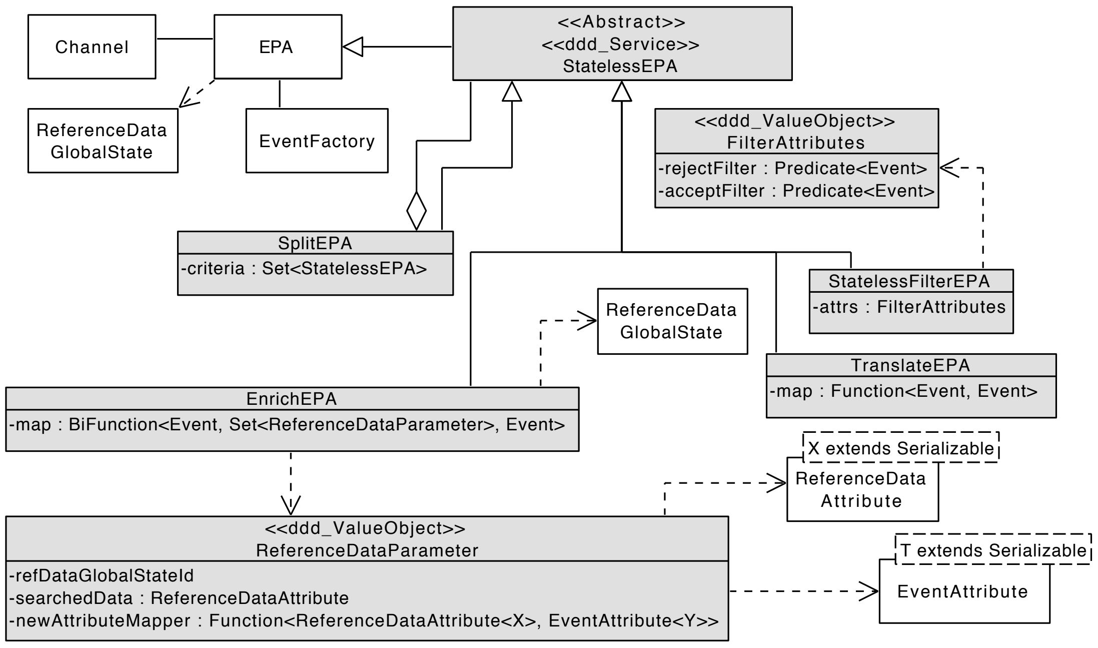
At the bottom of Figure 3.7 we see `StatefulEPA` and `StatelessEPA` extensions, further detailed in Section 3.6.1.1 and Section B.8.

Tables from Section B.6 on Appendix B outline attributes and methods related to model components from this section.

3.6.1.1 Stateless EPA

`StatelessEPA` is an abstract class that encompasses all EPA that processes each input event independently of any other event (*i.e.*, no state is maintained). It also includes a derivation step for the output.

This element does not present specific methods and attributes, but allows aggregation of correlated Stateless EPA to provide split functionality, as described below.



Channel
EPA
<<Abstract>>
<<ddd_Service>>
StatelessEPA
ReferenceData GlobalState
EventFactory
<<ddd_ValueObject>>
FilterAttributes
-rejectFilter : Predicate<Event>
-acceptFilter : Predicate<Event>
SplitEPA
-criteria : Set<StatelessEPA>
StatelessFilterEPA
-attrs : FilterAttributes
ReferenceData GlobalState
TranslateEPA
-map : Function<Event, Event>
EnrichEPA
-map : BiFunction<Event, Set<ReferenceDataParameter>, Event>
X extends Serializable
ReferenceData Attribute
<<ddd_ValueObject>>
ReferenceDataParameter
-refDataGlobalStateId
-searchedData : ReferenceDataAttribute
-newAttributeMapper : Function<ReferenceDataAttribute<X>, EventAttribute<Y>>
T extends Serializable
EventAttribute

Figure 3.9: Stateless EPA Diagram

The diagram of Figure 3.9 presents four Stateless EPA Types that are described above.

- `StatelessFilterEPA`: filters events based on explicit criteria (from `FilterAttribute`) for acceptance or rejection of those elements.
- `TranslateEPA`: receives a single event and generates another event according to the Function implementation for `map` attribute (which relates to requirement R3.2 from Table 2.2).
- `EnrichEPA`: takes a single input event, uses its attributes to query information from a Global State (relating to requirement R3.6 from Table 2.2), and finally creates a derived event (reusing EventFactory) with new attributes according to the BiFunction logic from *map* parameter (relating to requirement R3.11 from Table 2.2 if trig-

gered by PatternDetectEPA, from Section 3.6.1.3). Each new EventAttribute value is calculated based on ReferenceDataParameter, through establishing connectivity to ReferenceDataGlobalState based on `refDataGlobalStateId`; then, fetching a ReferenceDataAttribute according to its template, whose value (of generic type `X`) is used as a parameter to *newAttributeMapper* transformation Function to derive a new EventAttribute (of generic type `Y`). The reason why different letters were chosen is to represent scenarios where `X` and `Y` are different types.

- `SplitEPA`: takes as input a single event and creates a collection of events via multiple `StatelessEPA` instances. Therefore, leveraging StatelessFilterEPA, TranslateEPA and EnrichEPA implementations to derive events. This is a special case of stateless EPA composition.

Tables from Section B.7 on Appendix B outline attributes and methods related to model components from this section.

3.6.1.2 Stateful EPA

An EPA is stateful if the way it processes events is influenced by other event instances that have been processed by it (ETZION; NIBLETT; LUCKHAM, 2011; ARTIKIS et al., 2012). For instance, suppose an event rises every time a quantity of a given product is sold, and we need to find the total sold quantity. To continuously compute a running total, we can use a stateful EPA that emits new events containing the updated total as it receives a sale event - so events emitted by this agent depends on all the events that it has previously received (ETZION; NIBLETT; LUCKHAM, 2011).

The diagram of Figure 3.10 presents the three `StatefulEPA` types:

- `StatefulFilterEPA`: is similar to StatelessFilterEPA, but, in addition, its processing considers previous event instances (for instance, fetching last `N` occurrences, based on `StatefulFilterOperator`). Relates to requirement R3.1 from Table 2.2.

- `AggregateEPA`: produces an output event derived as a function of the incoming stream of events (*e.g.*, a sum operation). This aggregation is computed incrementally, via a `reduceOperator` (a BinaryOperator) along with an identity value

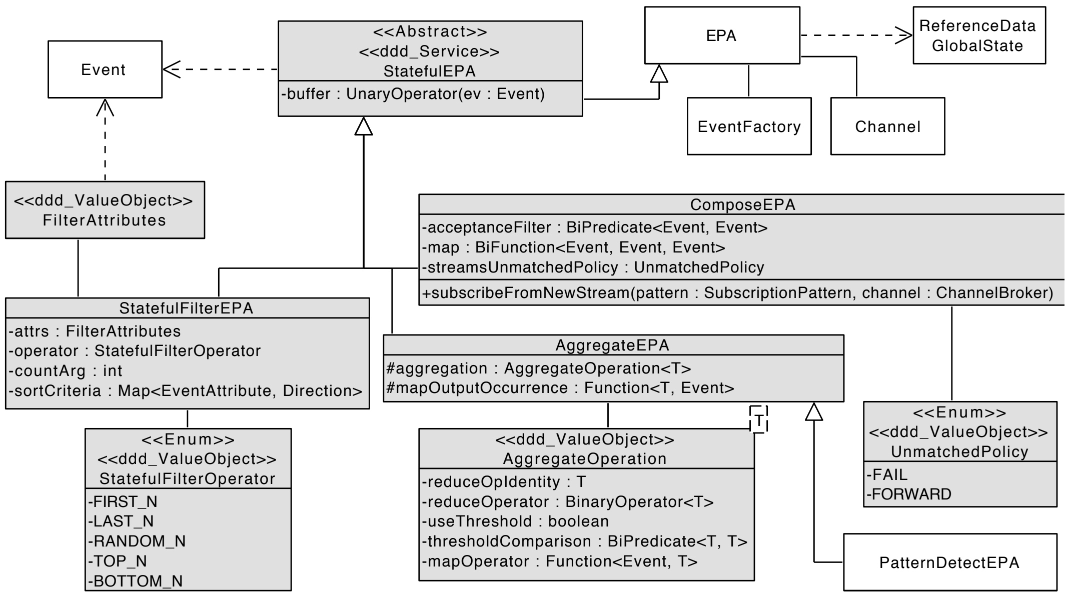

Event
<<Abstract>>
<<ddd_Service>>
StatefulEPA
-buffer : UnaryOperator(ev : Event)
EPA
ReferenceData
GlobalState
EventFactory
Channel
<<ddd_ValueObject>>
FilterAttributes
ComposeEPA
-acceptanceFilter : BiPredicate<Event, Event>
-map : BiFunction<Event, Event, Event>
-streamsUnmatchedPolicy : UnmatchedPolicy
+subscribeFromNewStream(pattern : SubscriptionPattern, channel : ChannelBroker)
StatefulFilterEPA
-attrs : FilterAttributes
-operator : StatefulFilterOperator
-countArg : int
-sortCriteria : Map<EventAttribute, Direction>
AggregateEPA
#aggregation : AggregateOperation<T>
#mapOutputOccurrence : Function<T, Event>
T
<<Enum>>
<<ddd_ValueObject>>
UnmatchedPolicy
-FAIL
-FORWARD
<<Enum>>
<<ddd_ValueObject>>
StatefulFilterOperator
-FIRST_N
-LAST_N
-RANDOM_N
-TOP_N
-BOTTOM_N
<<ddd_ValueObject>>
AggregateOperation
-reduceOpIdentity : T
-reduceOperator : BinaryOperator<T>
-useThreshold : boolean
-thresholdComparison : BiPredicate<T, T>
-mapOperator : Function<Event, T>
PatternDetectEPA

Figure 3.10: Stateful EPA Diagram

(`reduceOpIdentity`) for the accumulation function, relating to requirement R3.3 from Table 2.2. The applicable events can be further reduced through the usage of a threshold attribute and a comparison implementation (via BiPredicate).

- `ComposeEPA`: performs "join" operations of events coming from two input streams. A subscription method is used for matching event instances of the channels according to a condition expression - relating to requirement R3.4 and R3.5 from Table 2.2 (if composed along with a PatternDetectEPA, from Section 3.6.1.3). A `map` BiFunction outputs new derived events from each matched pair of source events.

Tables from Section B.8 on Appendix B outline attributes and methods related to model components from this section.

3.6.1.3 Pattern Detect EPA

Pattern Detect agents explore relationships between events to bring meaningful value to the business. For instance, a health monitor system may infer a standard behavior of monitored values for each patient based on sensors (tracking blood pressure, pulse, respiratory rate, etc.) and provides alerts if a combination of factors (including eventual absence of measurements) indicates abnormality (ETZION; NIBLETT; LUCKHAM, 2011).

These agents are supposed to be processed under ContextEPAComposite constructions

(Section 3.4.2) with a TemporalContext, so that statistics are continuously provided from aggregated groupings of data.

The diagram of Figure 3.11 presents `PatternDetectEPA` types of pattern detection strategies:

- `BasicPatternDetectEPA`: simply detects events matching a set of EventType instances (optionally ordered over time), ordered through time (relating to requirement R3.7 from Table 2.2) or not (`PatternSequence` attribute). A `Pattern-Modal` attribute indicates if at least one instance of all event types is met (ALL), if the set is partially met (SOME) or not met (NONE - relating to requirement R3.5 from Table 2.2);
- `ConditionalPatternDetectEPA`: extends BasicPatternDetectEPA, but requires *condition* to be satisfied by matching events, defined via Predicate;
- `TrendPatternDetectEPA`: detects trends based on how event attribute values change over time. This is performed by correlating pairs of subsequent Event instances via *trendCheck*, a BiPredicate function. Absolute or relative values can be evaluated (e.g., relative distance to a target decreasing at a certain speed). This relates to requirements R2.3, R3.2, R3.8 and R3.9 from Table 2.2. Specific conditions may trigger events that further launch EnrichEPA (from Section 3.6.1.1, relating to requirement R3.11 from Table 2.2) or dispatch a process via GlobalState (from Section 3.8, relating to requirement R3.13 from Table 2.2).

On top of AggregateEPA features (see Section B.8), PatternDetectEPA instances may consume historical events, and are configured with a `MatchingPolicy`, which indicates how to proceed whenever events satisfy the expected pattern. Each policy is based on:

- `EvaluationMode`: determines if this EPA generates output incrementally or at the end of the temporal context;
- `ExcessMode`: characterizes the behavior of this EPA when several instances of the same event type occur: we can either consider the first, or every one, or override with the last occurrence;

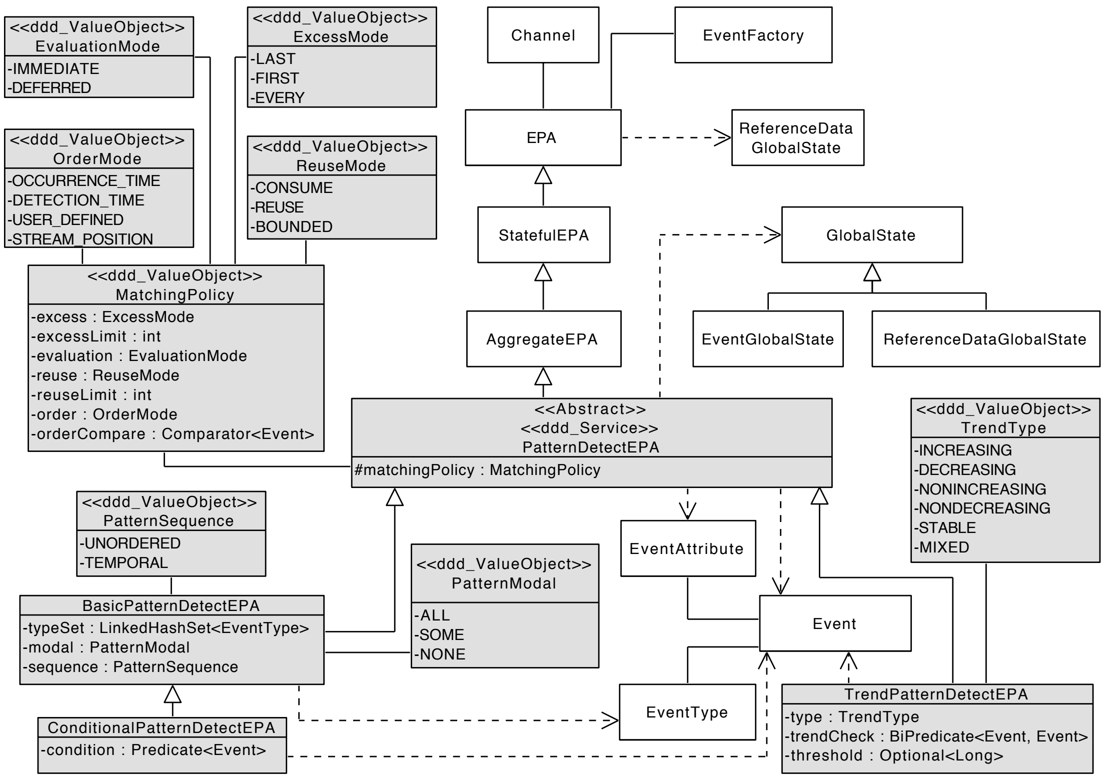

<<ddd_ValueObject>> EvaluationMode
-IMMEDIATE
-DEFERRED
<<ddd_ValueObject>> ExcessMode
-LAST
-FIRST
-EVERY
Channel
EventFactory
EPA
ReferenceData GlobalState
<<ddd_ValueObject>> OrderMode
-OCCURRENCE_TIME
-DETECTION_TIME
-USER_DEFINED
-STREAM_POSITION
<<ddd_ValueObject>> ReuseMode
-CONSUME
-REUSE
-BOUNDED
StatefulEPA
GlobalState
<<ddd_ValueObject>> MatchingPolicy
-excess : ExcessMode
-excessLimit : int
-evaluation : EvaluationMode
-reuse : ReuseMode
-reuseLimit : int
-order : OrderMode
-orderCompare : Comparator<Event>
AggregateEPA
EventGlobalState
ReferenceDataGlobalState
<<Abstract>> <<ddd_Service>> PatternDetectEPA
#matchingPolicy : MatchingPolicy
<<ddd_ValueObject>> TrendType
-INCREASING
-DECREASING
-NONINCREASING
-NONDECREASING
-STABLE
-MIXED
<<ddd_ValueObject>> PatternSequence
-UNORDERED
-TEMPORAL
EventAttribute
<<ddd_ValueObject>> PatternModal
-ALL
-SOME
-NONE
BasicPatternDetectEPA
-typeSet : LinkedHashSet<EventType>
-modal : PatternModal
-sequence : PatternSequence
Event
EventType
ConditionalPatternDetectEPA
-condition : Predicate<Event>
TrendPatternDetectEPA
-type : TrendType
-trendCheck : BiPredicate<Event, Event>
-threshold : Optional<Long>

Figure 3.11: Pattern Detect EPA Diagram

- `ReuseMode`: defines if event instances are to be discarded after consumption or considered in subsequent analyses;
- `OrderMode`: specifies a comparison parameter for event ordering. For temporal order, occurrence or detection time may be used. Another option is the stream position of the event, or a user-defined element (via a Comparator function on Event).

Tables from Section B.9 on Appendix B outline attributes and methods related to model components from this section.

3.6.1.4 Clustering Over Streaming

Continuous processing implies adjusting prediction models while streaming data, according to unsupervised learning algorithms that deal with dynamic data distributions that reflect the external world (MAYER; MAYER; ABDO, 2017). An extension of Pattern Detect EPA, named `ClusteringEPA`, offers our model a functionality for continuously assigning groups of related events, based on their features, into a predefined number of clusters (*k*). This relates to requirement R2.3 and R3.12 from Table 2.2.

Relevant event characteristics for clustering via EPA are represented in Figure 3.12. A template parameter *C* indicates a data type associated to one or multiple attributes - the latter is known as a multidimensional type.

A parameter was included to balance the relative importance of new data versus historical data (via *timeToLive* attribute), allowing faster reaction to changes.

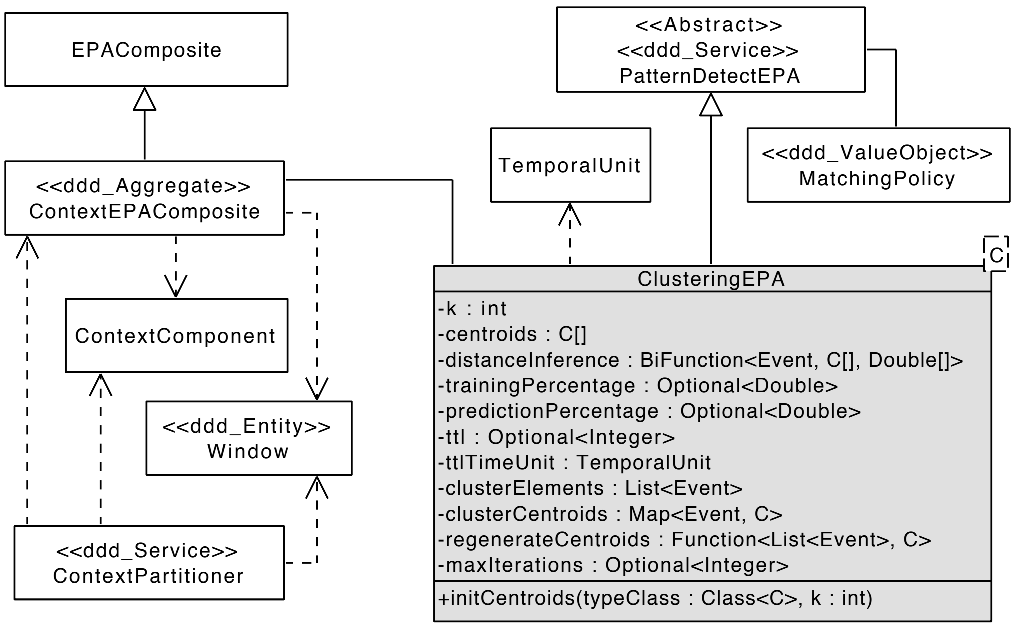

EPAComposite
<<Abstract>>
<<ddd_Service>>
PatternDetectEPA
TemporalUnit
<<ddd_ValueObject>>
MatchingPolicy
<<ddd_Aggregate>>
ContextEPAComposite
ContextComponent
<<ddd_Entity>>
Window
<<ddd_Service>>
ContextPartitioner
C
ClusteringEPA
-k : int
-centroids : C[]
-distanceInference : BiFunction<Event, C[], Double[]>
-trainingPercentage : Optional<Double>
-predictionPercentage : Optional<Double>
-ttl : Optional<Integer>
-ttlTimeUnit : TemporalUnit
-clusterElements : List<Event>
-clusterCentroids : Map<Event, C>
-regenerateCentroids : Function<List<Event>, C>
-maxIterations : Optional<Integer>
+initCentroids(typeClass : Class<C>, k : int)

Figure 3.12: Clustering Diagram

Although this component has been initially based on K-Means (FALK; GURBANI, 2017), the model is not constrained specific clustering implementations - for instance, as we find Manhattan Distance more relevant than Euclidean Distance due to outliers, we might opt for a strategy based on K-Median (WHELAN; HARRELL; WANG, 2015). The core aspect is that any implementation should minimize the distance of the nodes within the cluster, while maximizing the distance between groups, through the following steps, based on Freeman (2015):

1. start with *centroids* (a central tendency value for each cluster) as an output from previous iteration or, initially, randomly assigned;
2. add the new event/events (depends on triggering mechanisms) to *clusterElements*;
3. discard occurrences that should be expired (where the elapsed time since event occurrence > *timeToLive*), meaning it will favor recent occurrences;

4. for each item from *clusterElements*, compute the distance to every centroid - e.g., sum of squared distance between the the item's set of attributes and each centroid´s set of attributes. This is performed via the implementation provided by *distanceInference*;

5. assign every occurrence to the nearest centroid, according to the minimum distance above, in *clusterCentroids* map, updating values as needed;

6. for each cluster, re-compute the centroids from all measurements assigned to it (*regenerateCentroids*), and reassign those values into *centroids*, to be used in step 1 when applicable.

Steps 1-6 are performed until our convergence criterion is satisfied (*i.e.*, when cluster assignments in step 5 remains the same since last interaction), or the number of iterations exceed the value set in *maxIterations*. Once results are inferred, it may trigger events that further launch EnrichEPA (from Section 3.6.1.1, relating to requirement R3.11 from Table 2.2) or dispatch a process, e.g. via GlobalState (from Section 3.8, relating to requirement R3.13 from Table 2.2).

In our model, we can choose to use disjoint or intersecting subsets of events from the stream for training and inference operations (through *trainingPercentage* and *predictionPercentage* attributes). For exclusively predictive cases (*i.e.*, no training data), we can simply run step four (*distanceInference*, using the latest set of centroids).

ClusteringEPA instances are usually nodes within EPA compositions, bound to a context, but can also be standalone nodes, processing infinite streams - for which case *timeToLive* may not be of assistance.

Tables from Section B.10 on Appendix B outline attributes and methods related to model components from this section.

3.6.2 Context Partitioner

The processing of continuous event streams associated to a context is performed according to a segmented event grouping, identified as the `Window` component, depicted in Figure 3.13, which comprises related events.

There are cases where an event arrival demands the creation of a Window instance, and others where it simply becomes part of an already existing Window. An event that generates a Window is its initial event. So, for instance, in TemporalContext, a Window triggered by initial event e_1 in time instance t_1 comprises events e_2 ... e_n occurring between t_1 and t_1+(size x TemporalUnits). Here, event e_2 also starts a new Window which lasts until t_2+(size x TemporalUnits), comprising all events in that interval. In FixedTemporalContext, on the other hand, event e_2 will only be assigned to the window triggered by event e_1, and will not launch another Window.

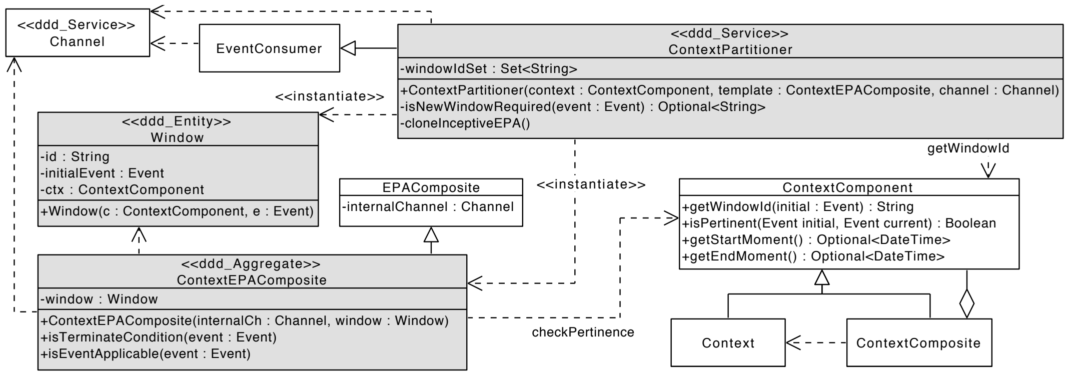
The EPA that handles event processing within context segments is `ContextEPAComposite`. Dynamic provisioning and decommissioning of this agent happens according to the pertinence of events to windows, as per the steps described below. When used along with Context (Section 3.4.2) to provision components that form the basis for event grouping, it relates to requirement R3.3 from Table 2.2.

1. As an event comes in, ContextPartitioner first checks if there is a requirement to create a new Window. It generates a window id based on the Context composition structure (consolidating - via concatenation or other means - all inner id pieces, using *getInnerWindowId* implementation from Context subclasses - see Section 3.4.2).

2. If the generated id is not among its Set of window ids, a new Window is dispatched, and set into a new ContextEPAComposite instance, based on its template (see *constructor* in Table B.53). This EPA runs in a separate thread, with an inner Channel, for as long as the Context is active (and it is no longer active, the EPA terminates).

3. Meanwhile, all other active ContextEPAComposite instances are also pulling subscribed events from their inner Channel, based on the pertinence criteria.

Figure 3.14 demonstrates an example of event windows based on an ContextComponent composed of an Attribute context (based on sensor ids) and a FixedTemporal context (defined with an amount of 2 days). The red boxes denotes single stateful filter EPAs that act within specific contexts, outputting minimal sensor reading value for a sensor on a date range. The picture represents what happens in day 5. The lower right composition is decommissioned, while a new one (triggered by ContextPartitioner), in the upper right



<<ddd_Service>>
Channel
EventConsumer
<<ddd_Service>>
ContextPartitioner
-windowIdSet : Set<String>
+ContextPartitioner(context : ContextComponent, template : ContextEPAComposite, channel : Channel)
-isNewWindowRequired(event : Event) : Optional<String>
-cloneInceptiveEPA()
<<instantiate>>
<<ddd_Entity>>
Window
-id : String
-initialEvent : Event
-ctx : ContextComponent
+Window(c : ContextComponent, e : Event)
EPAComposite
-internalChannel : Channel
<<instantiate>>
getWindowId
ContextComponent
+getWindowId(initial : Event) : String
+isPertinent(Event initial, Event current) : Boolean
+getStartMoment() : Optional<DateTime>
+getEndMoment() : Optional<DateTime>
<<ddd_Aggregate>>
ContextEPAComposite
-window : Window
+ContextEPAComposite(internalCh : Channel, window : Window)
+isTerminateCondition(event : Event)
+isEventApplicable(event : Event)
checkPertinence
Context
ContextComposite

Figure 3.13: Context Partition Model

corner, captures all events from Sensor A from this date on. Those events will also be processed by the upper left composition, which subscribes for Sensor A occurrences, and ignored by the lower left composition, which only fetches Sensor B occurrences.

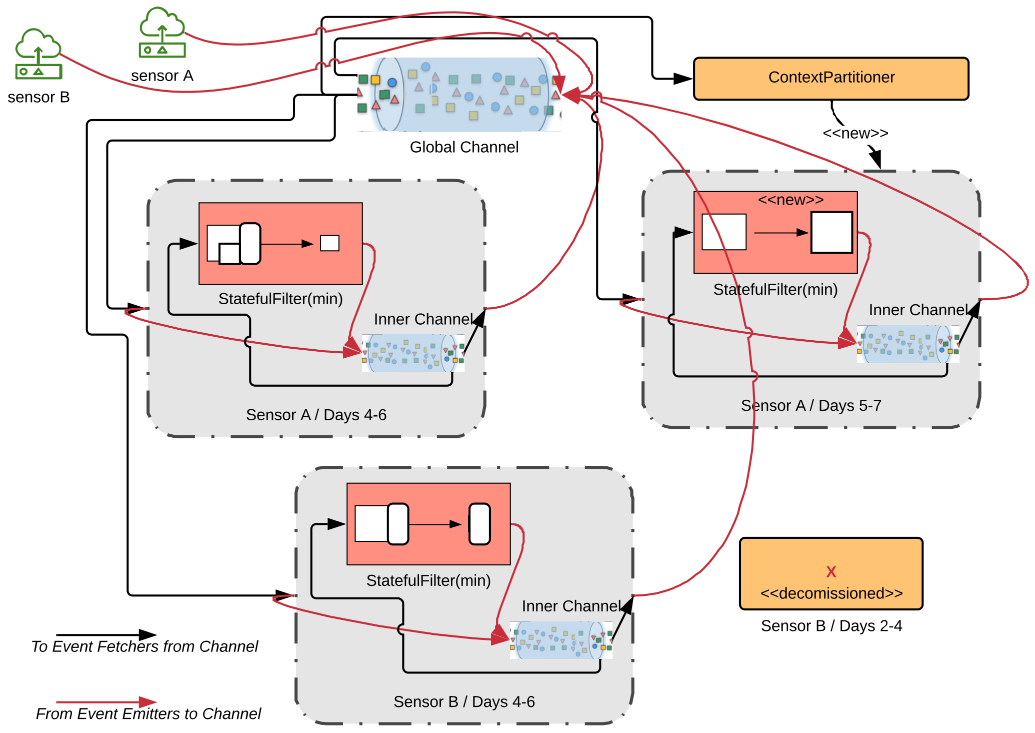


Figure 3.14: EPA Window Partitioning (Day 5)

Tables from Section B.11 on Appendix B outline attributes and methods related to model components from this section.

3.6.3 Offline Event Loader

As illustrated in Figure 3.15, `OfflineEventLoader` is an infrastructure component that leverages Producer functionality for an implementation that introduces batches of events from source systems (usually external repositories) into CEP platforms. OfflineEventLoader contains an instance of EventDataGlobalState (see Section 3.8) to fetch an accumulated list of events from its data source. Those events are further launched into CEP channels via the EventEmmitter implementation. This design reflects the CEP strategy adopted by D'silva et al. (2017), and relates to requirements R2.1 and R3.10 from Table 2.2.

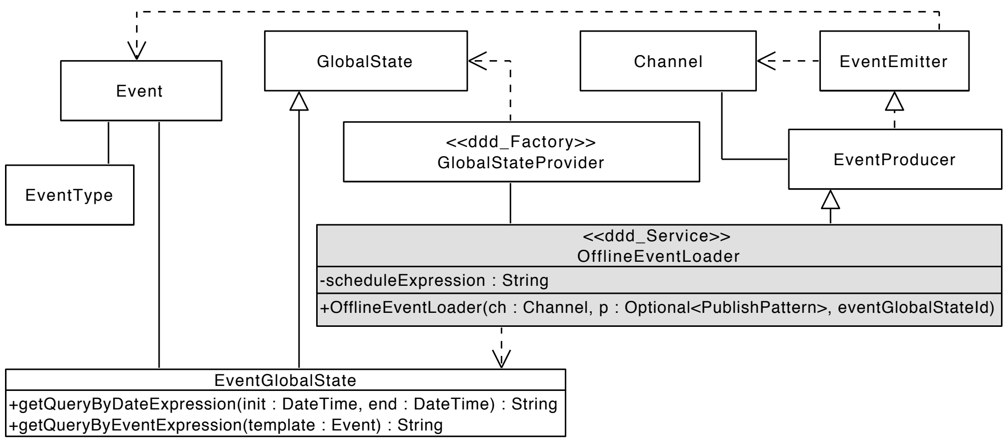

GlobalState
Channel
EventEmitter
Event
<<ddd_Factory>>
GlobalStateProvider
EventProducer
EventType
<<ddd_Service>>
OfflineEventLoader
-scheduleExpression : String
+OfflineEventLoader(ch : Channel, p : Optional<PublishPattern>, eventGlobalStateId)
EventGlobalState
+getQueryByDateExpression(init : DateTime, end : DateTime) : String
+getQueryByEventExpression(template : Event) : String

Figure 3.15: Offline Loader Diagram

An optimization for this component to act closer to real time updates is the adoption of continuous query mechanisms, comprising of sequences of one-time queries adjoined together (VIDYASANKAR, 2017) - as long as it does not impact source systems.

Tables from Section B.12 on Appendix B outline attributes and methods related to model components from this section.

3.7 Infrastructure Layer

This layer provides technical mechanisms to support higher layers, *e.g.*, message sending for application and persistence for domain. It may support the interactions between the layers through an architectural framework (EVANS, 2004).

3.7.1 Channel

Instances of `Channel` (Figure 3.16) capture events from event emitters, placing them on an event distribution platform, and route selected instances of events from there to event fetchers. Channel provides filtering mechanisms, and intermediates all accesses to and from the stream platform. Using such a distribution platform for asynchronously integrated and loosely coupled CEP components favors overall performance - EPA components can be placed on different processes, allowing higher parallelism (ETZION; NIBLETT; LUCKHAM, 2011).

Event distribution is provided by means of two methods:

- *publish*, which introduces events into the stream platform, so that all interested subscribers can pull it;
- *consume*, which fetches events generated from EPA or Producer instances. A SubscriptionPattern (see Section 3.5.2) can assist reducing the amount of data fetched, by establishing filters based on event types and attributes.

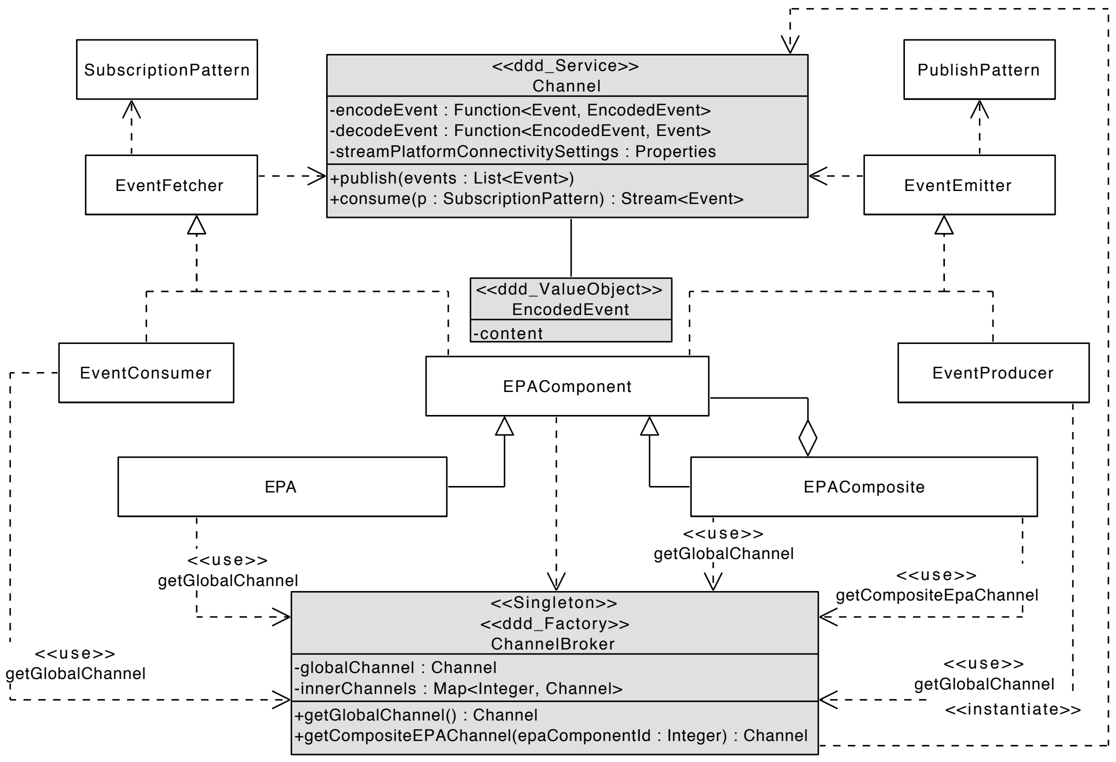


Figure 3.16: Channel Diagram

A unique Global Channel instance provides the infrastructure to transport and route messages between Producer, Consumer, and EPAs that do not relate to a composite EPA (*i.e.*, with no parent). On the other hand, for agents that participate on an EPA composition, a segregated inner channel (one per composition) is provided to handle message exchange between them (as explained on Section 3.6.1), therefore Channel relates to requirements R1 from Table 2.2. A `ChannelBroker` singleton assists EPA by providing both global and segregated Channel instances.

The Channel element is also responsible for optimizing event storage and traffic latency over the transportation platform. Binary encodings are suggested for optimized sizing and performance (MAEDA, 2012). Transformation from Event into `EncodedEvent`, as well as the reverse conversion are provided for sending and retrieving information in a compact form.

Implementations from known industry message brokers may be leveraged, such as event stream partitions and wildcard support on topics (DOBBELAERE; ESMAILI, 2017) for higher performance. Customizations may also be provided for higher workloads, such as topic query projections during subscription (FALK; GURBANI, 2017).

Tables from Section B.13 on Appendix B outline attributes and methods related to model components from this section.

3.7.2 Global State

A `GlobalState` is used by event processing agents whenever they need to access data outside the scope of the event being processed. It refers to information such as reference data, provided by `ReferenceDataGlobalState`, used - among other cases - to enrich events and drive operational processing (*e.g.*, timeout), relating to requirement R3.13 from Table 2.2; and event stores, holding historical events (relating to requirement R2.1 and R3.10 from Table 2.2), for which information is retrieved via `EventGlobalState`.

To provide data information from data stores, `GlobalState` interacts with a `Glob-alStateDataStore`, relating to requirement R3.6 from Table 2.2. Connectivity to the actual data store and interaction calls happens via the data store driver and are out of scope of this study.

Template parameter notations, presented in Figure 3.17, are indicated for `GlobalState`, its subclasses, and related data store (`GlobalStateDataStore`), and demonstrates the usage of dynamic type casting through the following generic types:

- `R`: a generic type related to data store tuple (not necessarily RDBMS), for instance a ResultSet[1] or a Row[2] - the record pointed by a cursor, as we iterate through records fetched from a query.
- `T`: a generic type that relates either to an Event entity or to a mapped `ReferenceDataAttribute`.

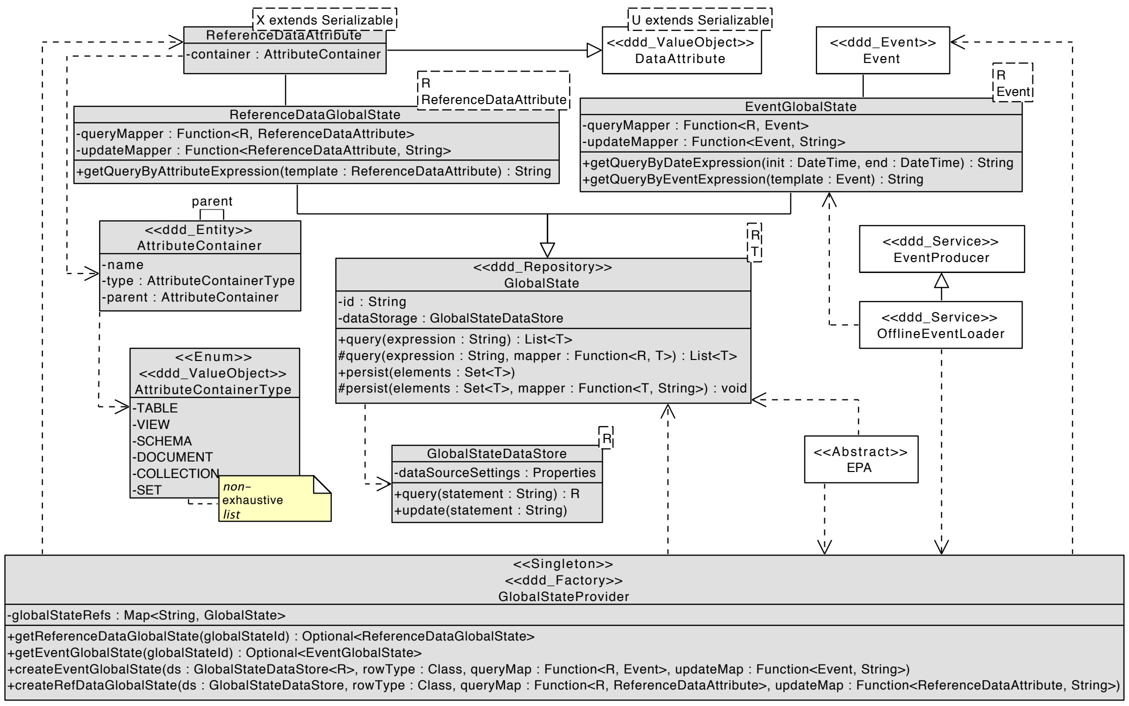

X extends Serializable
ReferenceDataAttribute
-container : AttributeContainer
U extends Serializable
<<ddd_ValueObject>> DataAttribute
<<ddd_Event>> Event
R ReferenceDataAttribute
R Event
ReferenceDataGlobalState
-queryMapper : Function<R, ReferenceDataAttribute>
-updateMapper : Function<ReferenceDataAttribute, String>
+getQueryByAttributeExpression(template : ReferenceDataAttribute) : String
EventGlobalState
-queryMapper : Function<R, Event>
-updateMapper : Function<Event, String>
+getQueryByDateExpression(init : DateTime, end : DateTime) : String
+getQueryByEventExpression(template : Event) : String
parent
<<ddd_Entity>> AttributeContainer
-name
-type : AttributeContainerType
-parent : AttributeContainer
R T
<<ddd_Repository>> GlobalState
-id : String
-dataStorage : GlobalStateDataStore
+query(expression : String) : List<T>
#query(expression : String, mapper : Function<R, T>) : List<T>
+persist(elements : Set<T>)
#persist(elements : Set<T>, mapper : Function<T, String>) : void
<<ddd_Service>> EventProducer
<<ddd_Service>> OfflineEventLoader
<<Enum>> <<ddd_ValueObject>> AttributeContainerType
-TABLE
-VIEW
-SCHEMA
-DOCUMENT
-COLLECTION
-SET
non-exhaustive list
R
GlobalStateDataStore
-dataSourceSettings : Properties
+query(statement : String) : R
+update(statement : String)
<<Abstract>> EPA
<<Singleton>> <<ddd_Factory>> GlobalStateProvider
-globalStateRefs : Map<String, GlobalState>
+getReferenceDataGlobalState(globalStateId) : Optional<ReferenceDataGlobalState>
+getEventGlobalState(globalStateId) : Optional<EventGlobalState>
+createEventGlobalState(ds : GlobalStateDataStore<R>, rowType : Class, queryMap : Function<R, Event>, updateMap : Function<Event, String>)
+createRefDataGlobalState(ds : GlobalStateDataStore, rowType : Class, queryMap : Function<R, ReferenceDataAttribute>, updateMap : Function<ReferenceDataAttribute, String>)

Figure 3.17: GlobalState Model

In order to allow ReferenceDataGlobalState to pull ReferenceDataAttribute using simple queries, AttributeContainer instances can be used to properly locate data structures - *e.g.* a table name, defined under a schema name, where both are represented as AttributeContainer with distinct AttributeContainerType, related via *parent* attribute.

Three methods (*query*, *persist* and *findOne*) support the functionalities for ReferenceDataGlobalState through the following way:

[1] https://docs.oracle.com/javase/7/docs/api/java/sql/ResultSet.html
[2] https://spark.apache.org/docs/2.1.0/api/java/org/apache/spark/sql/Row.html

- EPAs use a GlobalState subtype, supplying either EventAttribute or ReferenceDataAttribute as a template to derive a search expression for the *query* call. If an AttributeContainer, for instance, is populated on a ReferenceDataAttribute template, the query brings all attributes that belong to this container structure.
- EPA can invoke *persist* for Events or Attributes that need to be shared among other EPAs. Mappings from those structures to the persistence command String is provided by *updateMapper* Function implementations, from EventGlobalState and ReferenceDataGlobalState.
- The *findOne* call is used to fetch a unique instance that matches a template parameter (*e.g.* a unique identifier). An Optional response may be empty if no instance (or if more then one) applies.

Tables from Section B.14 on Appendix B outline attributes and methods related to model components from this section.

3.8 Evaluated Requisites

Table 3.1 presents examples of correlations from the requisites from Table 2.2 and the components designed for the proposed model, based on their provided features. Those features were previously discussed over scenarios from Section 3.4.1 to Section . They relate, each one, to one or more requisite descriptions, and their full coverage indicates that our model complies to all mapped requisites.

Table 3.1: Requirements and correlation to proposed model elements

Requirement	**Model Component(s)**
R1 - A strategy for EPA composition	EPA, Channel, Context & subclasses, and ContextPartitioner
R2.1 - Support for batch introduction of events	OfflineEventLoader, GlobalState
R2.2 - Advanced Filtering	EventHeader, SubscriptionPattern
R2.3 - Incremental Model Training	PatternDetectEPA, ClusteringEPA
R3.1 - Preprocessing	EventHeader, SubscriptionPattern, PublishPattern, StatefulFilterEPA
R3.2 - Alerts and Thresholds	TranslateEPA, TrendPatternDetectEPA
R3.3 - Simple Counting and Counting within Segmented Windows	ContextComponent & subclasses, ContextPartitioner, AggregateEPA
R3.4 - Joining Event Streams	ComposeEPA
R3.5 - Data Correlation, Missing Events, and Erroneous Data	ComposeEPA and BasicPatternDetectEPA
R3.6 - Interacting with Databases	GlobalState & subclasses
R3.7 - Detecting Temporal Event Sequence Patterns	BasicPatternDetectEPA
R3.8 - Tracking	TrendPatternDetectEPA
R3.9 - Detecting Trends	TrendPatternDetectEPA
R3.10 - Running the Same Processing Mechanisms in Batch and Realtime Pipelines	OfflineEventLoader, GlobalState & subclasses
R3.11 - Detecting and Switching to Detailed Analysis	TrendPatternDetectEPA, ClusteringEPA, EnrichEPA
R3.12 - Using a Model	ClusteringEPA
R3.13 - Online Control	TrendPatternDetectEPA, ClusteringEPA, GlobalState

4. Evaluation

This chapter presents the evaluation of our model proposal, which consists of the following exploratory studies:

- Industry Real Case Experiment - indicating the feasibility of an implementation based on the model;
- Feedback from Industry Experts - a complementary evaluation related to the capability of our model to meet a set of thirteen requirements of real time streaming analytics

4.1 Industry Real Case Experiment

To investigate benefits of our proposal, we implemented a scenario of a usage monitoring dashboard for a large company, using our application architecture model in a real industry case.

4.1.1 Scenario Requirements

This solution captures user interactions with applications and monitors 208 systems, used by more than 5.000 employees. The company is migrating IT infrastructure to the cloud, and the dashboard main goals were: understand how critical each system was for internal end users; estimate resources workload throughout different periods of the month; and, assist on the definition of the most efficient strategy to provide virtual machine resources.

The solution provided utilization indicators per system, aggregated by day, weeks, or any date range (Figure 4.1), and utilization summaries per user. Also, panels showing measurement indicators per company sectors (Figure 4.2) were required. In this case, an incremental processing strategy provided the classification of systems into 4 clusters, shown in different colours according to their utilization intensity (measured in terms of the number of user sessions collected for each application on a specific period - from red, for the ones with minimum utilization, going to yellow, green and blue - the latter encompassing most used systems).

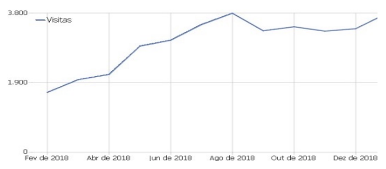


Figure 4.1: Usage Historic Report

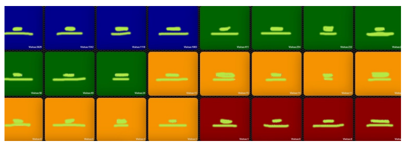


Figure 4.2: Utilization Panel for a Corporate Division

Monitored systems were classified as below, based on mechanisms for capturing their usage-related events:

- Group 1 (G1): Systems with no authentication mechanisms - capture usage events via web tracking;
- Group 2 (G2): Systems with push notification - newer systems use mechanisms to publish events of specific topics into a messaging platform;
- Group 3 (G3): Systems integrated to a Central Authentication Service (CAS) - a batch mechanism captures authentication events from a repository from CAS. This is required due to access time restrictions to non-commercial hours. A cyclic rotation for each G3 system is performed every 10 minutes to retrieve applicable Login, Authentication Error and Authenticated Access events from CAS since last processed timestamp;
- Group 4 (G4): Legacy systems that use its own authentication schema - a scheduled job searches for this information in persisted audit trails (via continuous query or API) and carries it to the broker every few minutes (next to real time). A corporate data source provides maps from ids of system users to CAS corporate ids;

We present our solution scenario using UML Use Case[1] and Activity[2] diagrams. Their description is detailed along the following sections.

4.1.1.1 Event Input and Output

Picture 4.3 demonstrates how utilization occurrences, referenced as raw events, from monitored systems (represented as producers) can be introduced into our CEP platform for further processing, and also how resulting information from CEP processes can be useful to the end user.

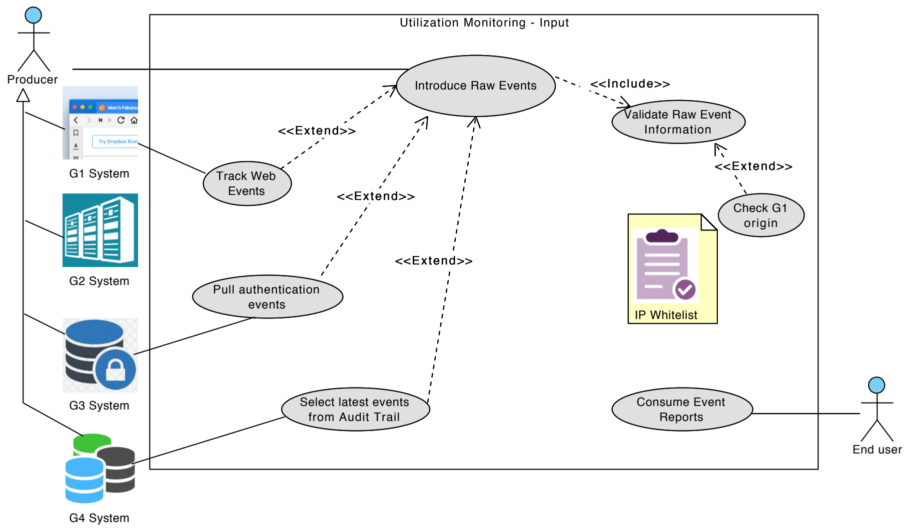


Figure 4.3: Use Case - Event Input and Output

Tables 4.1 to 4.7 provides use case descriptions from this picture.

[1]Use Case Diagram - communicates the high-level functions of the system and the system's scope, including the relationship of actors (who interact with the system) to essential processes, as well as the relationships among different use cases. Source: https://developer.ibm.com/articles/an-introduction-to-uml/

[2]Activity Diagram - Shows the procedural flow of control while processing an activity. Source: https://developer.ibm.com/articles/an-introduction-to-uml/

Table 4.1: Use Case: Introduce Raw Events

Actor	Producer
Precondition	Users visit a monitored system, from groups 1 to 4
Postcondition	Usage-relate events are introduced in CEP platform

Basic Flow

Producer relates to monitored corporate systems. Under the occurrence of events related to those systems utilization, the system pulls information such as system id, timestamp, user or origin of the request. The system also validates some of those fields, for instance to guarantee that mandatory fields do not come as null. Then it formats and populates an event data structures used to properly introduce occurrence data in CEP platform. Systems from group 2 initiate the interaction with event platform following the basic flow.

Alternate Flow 1

When producers are not capable to interact with CEP Platform. In this case special components (such as jobs or listeners) must be implemented to capture utilization information and introduce event data into CEP platform.

Alternate Flow 1a - See *Track Web Events* use case from Table 4.2

Alternate Flow 1b - See *Pull CAS Events* use case from Table 4.3

Alternate Flow 1c - See *Pull Events from Audit Trail* use case from Table 4.4

Table 4.2: Use Case: Track Web Events (extension from Table 4.1)

Precondition	Systems from group 1 are accessed
Postcondition	Event information is collected

Basic Flow

The system provides a web tracking listener that collects the origin IP, system id, and a timestamp related to events. Those are sent by group 1 System according to triggering mechanisms, via an HTTP POST request to this listener. Once this information is received, system proceeds with the basic flow from *Introduce Raw Events* use case (Table 4.1).

Exception Flow

If connectivity fails, the listener is not able to recover missing events. Reconstruction can be performed if web server logs are active, by mimicking http request triggering mechanisms on visited pages.

Table 4.3: Use Case: Pull CAS Events (extension from Table 4.1)

Precondition	Systems from group 3 are accessed, utilization information is kept in CAS repository
Postcondition	Event information is collected

Basic Flow

This requires a system component to connect and retrieve a batch of events from CAS from time to time to pick up all occurrences introduced since the last successful pull operation (it requires keeping the last successful timestamp). Once this information is received, system proceeds with the basic flow from *Introduce Raw Events* use case (Table 4.1).

Exception Flow

If an error aborts this process, missing events and new events (*i.e.*, created after last successful timestamp) are picked up on the next run.

Table 4.4: Use Case: Pull Events from Audit Trail (extension from Table 4.1)

Precondition	Systems from group 4 are accessed, utilization information is kept in their internal repositories
Postcondition	Event information is collected

Basic Flow

Utilization information is mapped from audit trail data structures, within internal tables from group 4 systems. A component needs to continuously query it to pick up all events introduced since the last pull operation (it requires keeping the last successful timestamp). Once this information is received, system proceeds with the basic flow from *Introduce Raw Events* use case (Table 4.1).

Exception Flow

If an error aborts this process, missing events and new events (*i.e.*, created after last successful timestamp) are picked up on the next run.

Table 4.5: Use Case: Validate Raw Event Information (inclusion from Table 4.1)

Precondition	Event information is Introduced
Postcondition	Event introduction process proceeds with valid information
Basic Flow	
Event information is validated, for instance to guarantee that mandatory fields (such as system id and event timestamp) do not come as null.	
Alternate Flow 1 - See *Check G1 Origin* use case from Table 4.6	
Exception Flow	
If any validation rule is rejected, the incoming event is rejected.	

Table 4.6: Use Case: Check G1 origin (extension from Table 4.5)

Precondition	Systems from group 1 deliver events including IP information
Postcondition	Evaluated information is valid
Basic Flow	
Events coming from systems from group 1 do not bring user information, since they do not relate to an authentication process - in this case the IP addresses coming from user HTTP request is used to associate a group of related requests, and only IP adresses that relate to monitored apps (according to a whitelist) are accepted.	
Exception Flow	
A validation rule rejects the introduction process aborts, rejecting the incoming event.	

Table 4.7: Use Case: Consume Event Reports

Actor	End User
Precondition	Raw events are processed and derived events are produced as a result.
Postcondition	Consolidated and detailed reports are provided to end users.
Basic Flow	
Occurrences of interest, from raw and derived events, are consumed and organized in data structures that provide end users their desired information at different detailing levels.	

4.1.1.2 Event Processing Activities

Picture 4.4 demonstrates how raw events, introduced on CEP platforms as displayed on Section 4.1, are further processed by CEP components to derive enriched and aggregated events that show information in a level of detail that is useful for end users. Tables 4.1 to 4.4 provides activity descriptions from this picture.

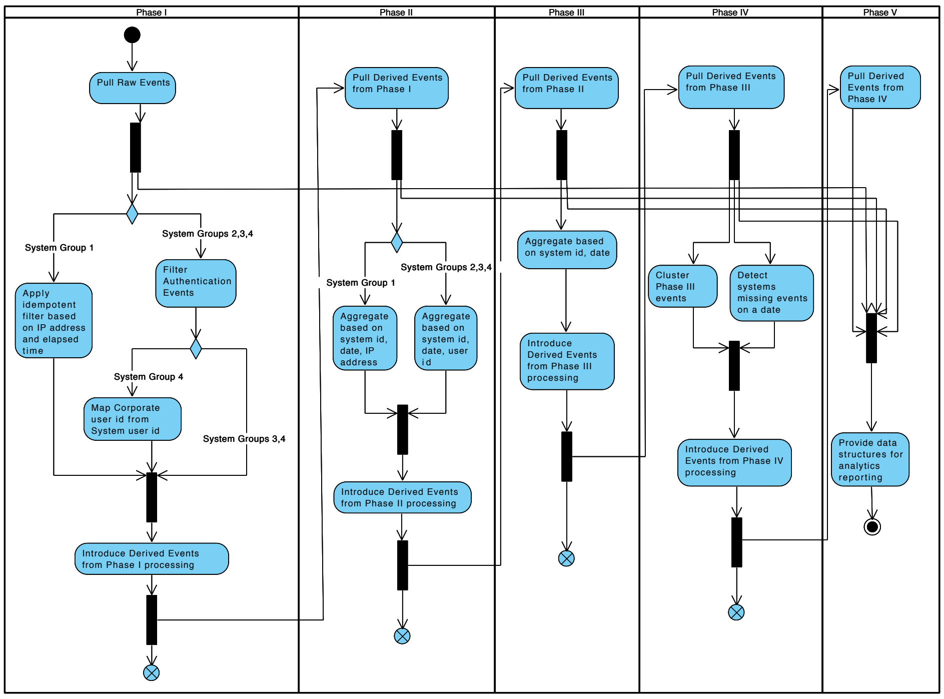

Phase I
Phase II
Phase III
Phase IV
Phase V
Pull Raw Events
System Group 1
System Groups 2,3,4
Apply idempotent filter based on IP address and elapsed time
Filter Authentication Events
System Group 4
Map Corporate user id from System user id
System Groups 3,4
Introduce Derived Events from Phase I processing
Pull Derived Events from Phase I
System Group 1
System Groups 2,3,4
Aggregate based on system id, date, IP address
Aggregate based on system id, date, user id
Introduce Derived Events from Phase II processing
Pull Derived Events from Phase II
Aggregate based on system id, date
Introduce Derived Events from Phase III processing
Pull Derived Events from Phase III
Cluster Phase III events
Detect systems missing events on a date
Introduce Derived Events from Phase IV processing
Pull Derived Events from Phase IV
Provide data structures for analytics reporting

Figure 4.4: Event Processing Activities

Swim lanes from this picture 4.4 reflect flows that represent processing phases that actually occur in parallel. But it clarifies dependencies between then - since output from specific phases (i.e., derived events) serves as input to other phases. It clarifies how event derivation can be performed in phases, where activities on each phase benefit from results made available previously.

On phase I, we pick up any raw event and basically derive events that represent a user session (*i.e.*, a group of interactions with the system over a period of time). This is done by pulling up login occurrences (where authentication applies), or leveraging IP addresses and request timestamps (when user is anonymous). This phase is important since, by discarding other event types, the population of events left for further processing operations substantially reduces (as shown in Section 4.1.6). An activity for systems from group 4 also helps standardising event attributes, enriching those with the corporate user id, mapped based on user id from origin system.

On phase II, starting from events delivered in phase I, event aggregates are provided based on the amount of user sessions (or IP session) for a system, at a specific date.

On phase III, starting from events delivered in phase II, event aggregates are provided according to system utilization summaries (total number of sessions per system) on a date.

On phase IV, starting from events delivered in phase III, the clustering algorithm is dispatched to find four centroids according to utilization summaries - and a routine also detects systems for which no session was observed.

On phase V we indicate that previously processed events provide online reporting capabilities to dashboard users, including the possibility to drill down for detailed information (for instance, retrieve the system with maximum utilization, then pull the user sessions on that system over a period, or select the user / origin IP that triggered the maximum number of hits on the system for a period).

4.1.2 Assessment Goals

To evaluate quantitative metrics of this implementation, we used a performance evaluation framework suitable for CEP (MENDES; BIZARRO; MARQUES, 2008, 2013) as a reference. Metrics were collected on all Channel and EPA components. Producers present a high variability among themselves (with a strong dependency to external systems), so we evaluated event creation once incoming occurrences were ready to be cast into Event instances.

We collected information for the week we had two major transitions of existing (non-monitored) systems to the Dashboard - one with 15 systems, another with 22 systems, indicating high loads of utilization events during 2 nightly time-frames. The following measurements were provided:

- volumes of events processed, under distinct load scenarios - indicators provided in this study distinguish regular load volumes (when utilization events are being monitored online) and heavy loads (when batches of historical data are loaded in parallel to online processing);

- processing time, segregated by processes and components, indicating, indicating which of these may be burdening event processing time;
- throughput indicators (an average indication of the number of events being processed per unit of time), segregating measurements over processes and components, indicating which of these may be burdening total event processing time.

We also evaluated correctness and completeness of results, *i.e.*, observing the percentage of events correctly assigned and processed by EPA nodes, we defined the following terminology, adapting from Fawcett (2006):

- true positive - events properly processed;
- true negative - events properly discarded;
- false positive - events incorrectly processed (either because they should have been rejected, or because they resulted on incorrect outcomes);
- false negative - events incorrectly discarded (because they should have been processed).

And based on that we calculated two metrics, defined as accuracy and precision (FAWCETT, 2006):

- the accuracy metric, obtained by dividing the number of events properly processed (true positives) or properly discarded (true negatives) by the total number of events;
- the precision metric, obtained by dividing the number of events properly processed (true positives) by the total of number of processed events (true positives plus false positives).

4.1.3 Resolution Strategy

The solution architecture components were conceived using the proposed model as a guidance, starting from its meaningful layer compartmentalization, at first clarifying important concepts and components pertaining to each layer, then detailing elements that should be arranged in a solution to provide relevant functionalities in regard to the business goals.

Further sections describe our journey when applying our model for the solution depicted in Figures 4.5 and 4.6. The first picture presents two systems for each group classification, to avoid over-polluting the diagrams.

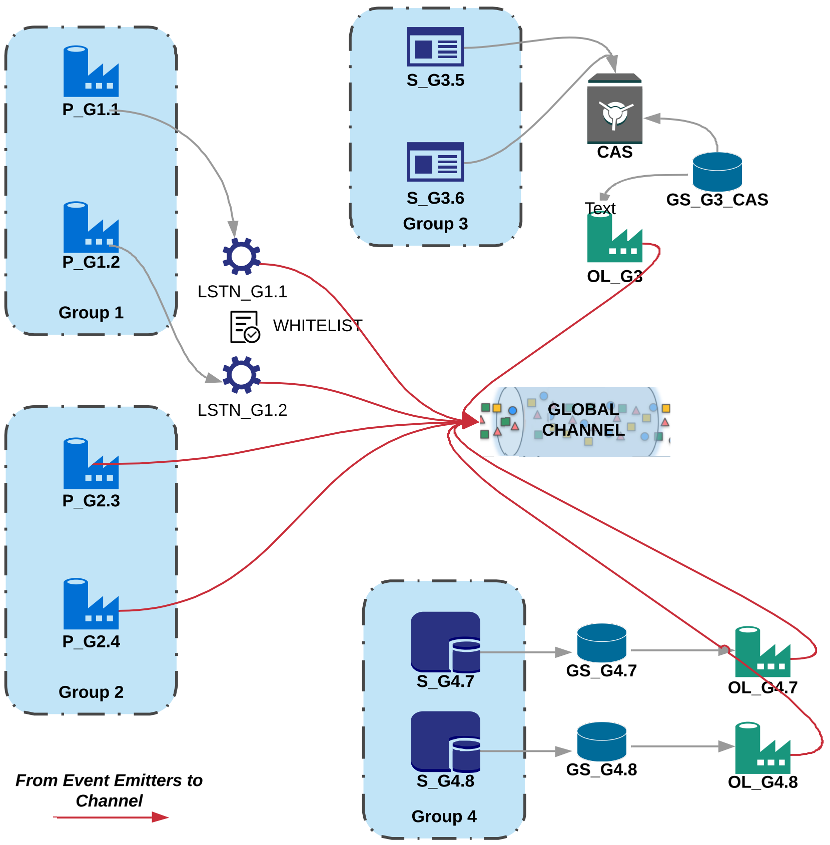
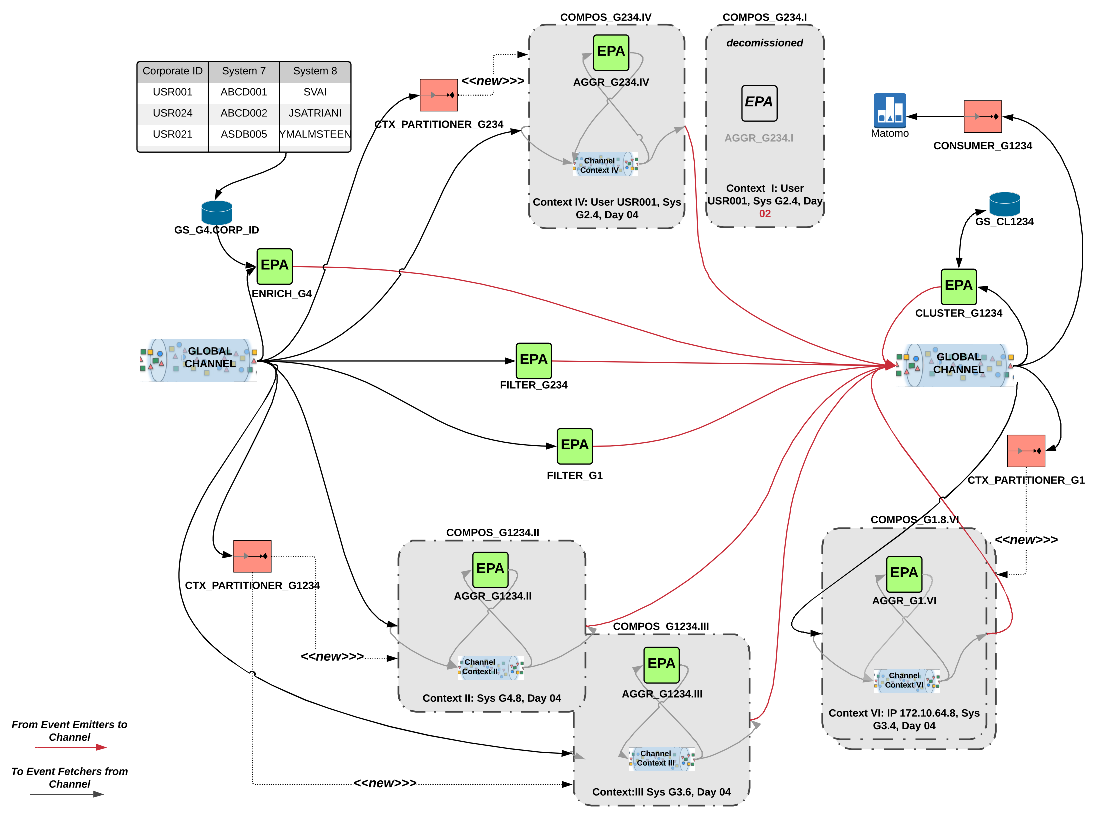


Figure 4.5: Solution from Model - Producers

4.1.3.1 UserInterface Layer

Starting from our monitored systems, which are our external input actors, and based on their classification, we identify two cases that relate to Producer nodes, introducing events into our CEP platform - groups 1 and 2; and two cases that require the usage of a global state for integration with repositories and services - groups 3 and 4.

Systems from group 1 uses web tracking mechanisms that trigger an HTTP POST call every time a system login page is loaded from a browser (each system has a listener, mapped for each producer, named LSTN_G1.1 and LSTN_G1.2). This listener rejects events based on the origin server IP, checking if it does not come from a valid IP, based



Corporate ID	System 7	System 8
USR001	ABCD001	SVAI
USR024	ABCD002	JSATRIANI
USR021	ASDB005	YMALMSTEEN

Figure 4.6: Solution from Model - EPA and Consumers

on a configured whitelist.

Systems from group 2 are able to directly feed our event distribution platform, acting as as CEP producers, named P_G2.3, P_G2.4.

Those listeners and agents are designed according to EventProducer component, and interact with a global channel, described in Section 4.1.3.4. For group 1, all events are considered as regular usage events. For group 2, events whose EventType matches the ones specified in PublishPattern (related to login, logout, authentication error or usage) are forwarded to the channel.

Systems from group 3 and group 4 utilize a strategy to introduce into the CEP platform a batch of accumulated events, thus the usage of OfflineLoader.

For all systems from group 3, a CAS consumer for a single OfflineLoader (OL_G3) periodically fetches occurrences related to login, logout and authentication error and usage events from all systems, and PublishPattern restrict the incoming instances to valid EventType matches.

Systems from group 4, on the other way, present evidences persisted in audit trails over distinct data models and technologies - *i.e.*, a polyglot persistence scenario. OfflineLoader instances for each system (OL_G4.7, OL_G4.8) guarantee login, logout and authentication error and usage events are introduced in a standardized format at this point.

To benefit from CEP processing results, an event consumer (CONSUMER_G1234) interacts with an external application named Matomo Web Analytics[3], feeding this system with raw captured events, related to login, logout, denied authentication and usage) and derived events (summarized utilization information) from all systems, so that Dashboard reports are produced and exposed for the users, near real-time, with drill-down capacity.

4.1.3.2 Domain Layer

Relevant raw event types coming from producers are: *successful_login*, *successful_logout*, *system_usage*, and *unsuccessful_login*.

For systems from group 1, raw events bring an IP address attribute in the Event payload allows an approximation to the user session concept, due to the fact all user requests are anonymous - grouping is based on request source IP address, instead of user ids. For this case, a composed context (C_1) complies: a SegmentedContext, based on origin_ip, a SegmentedContext, based on system_id, and a FixedTemporalContext comprising daily occurrences (starting at 00:00 hours every day).

This already provide useful information for reporting, but the utilization panel (Figure 4.2) and related functionalities requires a higher level of data consolidation. To achieve it, a composed context (C_1234) gathers all daily occurrences based on system_id.

Events include attributes in the EventPayload such as corporate_user_id, group_id, and system_id. Such attributes are used, for instance, in a Composed Context (C_234) that complies: a SegmentedContext, based on corporate_user_id, a SegmentedContext, based on system_id, and a FixedTemporalContext comprising daily occurrences (starting at 00:00 hours every day) of events with *session_user* type. This composition repeatedly yields the sum of utilization occurrences (aggregated by event type, system_id, and corporate_user_id) as events in the following way:

[3]https://matomo.org

Event header: occurrence_time: 00:10:56

Event type: successful_login

Event Payload: group_id=3, system_id=G3.5, corporate_user_id=USR006

Event Payload derived attributes: totalOcurrences=76

Also, five event types were defined to denote consolidations performed by the system for derived events: *session_user*, *session_ipaddr*, *summary_daily_system_session_user*, *summary_daily_system_session_ipaddr*, and *summary_daily_system_session*.

4.1.3.3 Application Layer

Initially, 2 activities were required to derive events that represent a user session for further data consolidation (see the activity diagram from Section 4.1.1.2):

- A StatelessFilterEPA (FILTER_G1) captures events created by LSTN_G1.1 and LSTN_G1.2, filtering occurrences based on the group_id value (matches if it is equal to 1). It also verifies under *rejectFilter* Predicate, and if the same IP sends an event within the last 5 minutes it is rejected (a "session" criteria for idempotent events). Derived events carry the type *session_ipaddr*;
- Another StatelessFilterEPA (FILTER_G234) captures events according to a SubscriptionPattern that implements its *filterByHeader* BiPredicate to pull events that present group_id equals to 2, 3 and 4, and implement its *filterByTypes* BiPredicate to accept only *successful_login* types, deriving events with *session_user* type.

Further, an EnrichEPA (ENRICH_G4) captures *session_user* events, filtering occurrences based on the group_id (matching if value equals 4), and complements events with the standard corporate user id (corporate_user_id attribute), with the assistance of a GlobalState, defined in Section 4.1.3.4 as GS_G4.CORP_ID, that maps this value based on the authenticated credential from origin system. Events for which existing credentials are invalid (if they do not match standard corporate user ids) are discarded (*i.e.*, removed from global channel).

Composite EPAs can now process specific session-related event occurrences, in standardized formats. Two ContextPartitioner components, described below, dispatch composite EPAs for aggregating session-related events - each one including at least one segregated inner channel and an aggregate EPA that sums up events grouped according to composed contexts C_1 and C_234, defined in Section 4.1.3.2:

- CTX_PARTITIONER_G1, related to group 1: subscribes to *session_ipaddr* events and creates ContextEPAComposite nodes (COMPOS_G1) according to C_1 (based on daily occurrence groupings of origin_ip and system_id), deriving, via an aggregate EPA (AGGR_G1), events of type *summary_daily_system_session_ipaddr*.
- The CTX_PARTITIONER_G234 captures *session_user* events and creates creates ContextEPAComposite nodes (COMPOS_G234) according to C_234 (based on daily occurrence groupings of corporate_user_id and system_id), deriving, via an aggregate EPA (AGGR_G234), events of type *summary_daily_system_session_user*.

An indication of the context instance in Figure 4.6, along with its attributes, is provided to show how dynamic compositions relate to it in a hypothetical scenario (including a case where an EPA is decomissioned, which happens once a context instance is no longer valid). It is also shown how inner channels relate to context instances.

Further, utilization indicators are summarized, via CTX_PARTITIONER_G1234, for all systems - it creates COMPOS_G1234 EPA instances from a composed Context (C_1234, from Section 4.1.3.2) based on system_id and daily occurrence. COMPOS_G1234 picks up *summary_daily_system_session_user* and *summary_daily_system_session_ipaddr* event types, and derives, via an aggregate EPA (AGGR_G1234), *summary_daily_system_session* events.

In order provide our Dashboard (Figure 4.2) the capacity to dynamically group systems according to their measured utilization (Section 3.6.1.4), an unsupervised learning strategy, based on our utilization metrics created by COMPOS_G1234 was implemented. CLUSTER_G1234 separates those collected metrics values into four groups that reflect the intensity of their utilization (from no usage to higher indicators). The chosen clustering implementation, based on Spark K-Means[4] is always triggered at an arrival of an *summary_daily_system_session_ipaddr* event. The resulting centroids are persisted in a

[4]https://spark.apache.org/docs/latest/mllib-clustering.htmlstreaming-k-means

Global State attribute that is shared among successive processes, so that values from previous execution are used as the initial assignments for the cluster centroids, at step 1 of the algorithm (except for the first time it ran, where random values were assigned). This way, the prediction model is not rebuilt from scratch every time COMPOS_G1234 outputs an event - instead, it is incrementally updated, leveraging previous effort. Besides colouring system panels, cluster indications are provided in upper right section of the Dashboard (Figure 4.7), showing the three midpoints between the four chosen centroids.

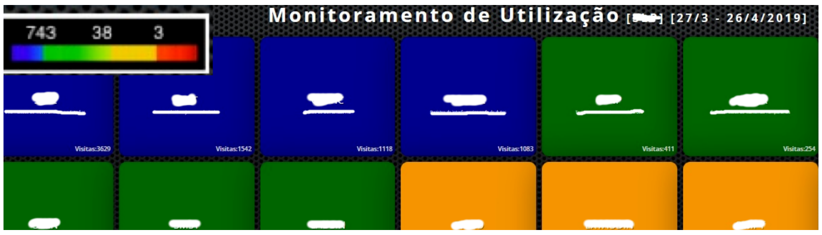


Figure 4.7: Dashboard Color Clusters

4.1.3.4 Infrastructure Layer

At this point, we have already acknowledged relevant information for component aspects to represent business goals, considering the integration among CEP actors via a channel that interacts with a streaming platform, but so far no assumption was made on technologies for this platform. This is a positive aspect of our model - it lets developers discover more about business requirements based on attributes and responsibilities from components in higher layer prior to defining technical components (and then be constrained to its functionalities).

Considering the architecture layout and expected volumes for our systems we opted to use RabbitMQ[5] message distribution platform. The choice was made taking into account also lower costs in terms of server resource processing as we compare it to Kafka[6] (*e.g.*, no need for Zookeper[7] resources for coordination among different nodes), yet bringing low latency and high throughput on message consumption (DOBBELAERE; ESMAILI, 2017). It also handles automatically the triggering of new queues (for inner channel elements).

It is important to notice that different streaming flow volumes and distribution de-

[5]https://www.rabbitmq.com
[6]https://kafka.apache.org
[7]https://zookeeper.apache.org

mands, on other business scenarios, may suggest different platforms.

Note the global channel is represented twice in Figure 4.6 to minimize crossing arrows.

Another relevant infrastructure element indicated in Sections 4.1.3.1 and 4.1.3.2 is the global state.

A GS_G3 GlobalState provides OL_G3 connectivity to CAS data source to fetch Login and Authentication_Error events from G3.

GlobalState instances are also created to pull data from each system in G4 (GS_G4.7, related to OL_G4.7; and GS_G4.8, related to OL_G4.8). For one specific case, data is retrieved via continuous query persistent connections, via Apache Ignite[8], so that updates are perceived closer to transaction occurrences. Another GlobalState (GS_G4.CORP_ID) shared by all G4 offline loaders, supplies user id mapping, from any source system to corporate_user_id;

And GS_CL1234 is used by CLUSTER_G1234 to store the latest inferred set of centroids.

4.1.4 Implementation strategy

We analyzed main principles from Event Data Pump design strategy (VILLAÇA; AZEVEDO; BAIÃO, 2018) for our implementation, heavily used in microservice architectures for close-to-real-time integration among distributed and loosely coupled services, and we notice we can benefit from their alignment with our design requirements. In this strategy, data is sent to orchestrated components that are designed to act as events raise (closer to real-time occurrences), through queuing mechanisms, providing consumer upfront access to updated information, yielding better performance metrics (as opposed to waiting for scheduled jobs to pick up data updates).

Beyond publisher-subscriber mechanisms for queuing and distributing events to parties, this design favors the ability to handle specific processing needs via dynamic provision of resources (for instance scaling up CPU and memory for processing historical data batches, which can be freed upon later) and promote characteristics such as packaging

[8]https://apacheignite.readme.io/docs/continuous-queries

methodologies for deploying autonomous services. Thus, design decisions like orchestrating containerizing EPA compositions and dynamic aggregations, according to context partitioning, were greatly inspired from this architecture strategy.

4.1.5 Multi-threading considerations

The main driver of reactive streams solutions is to govern stream data consumption across asynchronous boundaries between decoupled components – for instance, delegating processing on some of the stream elements to separate threads, while ensuring that the receiving side is not forced to handle a higher amount of data that it can handle (DAVIS, 2018).

This relates to providing mechanisms that allow the queues, which mediate between threads, to be consumed without impacting not only original producers, EPAs and end consumers, but also the infrastructure components that provides storage and processing mechanisms for the queues.

Ensuring certain computations happen on the proper thread (many times outside the main thread, to alleviate processing demands) is a common development challenge when dealing with reactive flows (DAVIS, 2018).

This study proposes a model where the streaming platform and each related agent can be implemented on a segregate process, with an inner logic that can handle the reception of stream events through a single main thread, which may delegate its tasks to multiple threads (if sequential processing considerations do not apply), maximizing performance on individual components. But multiple streaming operations can be established in a processing pipeline while iterating through a possibly infinite number consumed events. To effectively handle the proliferation of threads dispatched by the original thread, strategies such as cached thread pool (DAVIS, 2018) can be applied for reuse and mitigation of the overhead when managing processing.

Thus, by having one separate process per agent (Producer, Consumer or EPA) - which can be viewed, depending on implementation aspects, as container instances, distributed over different hosts - combined with autonomy to dispatch and manage threads that benefit from stream processing features, we are capable to implement solutions that allows better composition and handles the need for scalability.

4.1.6 Results

Evaluation followed the guidelines established in Section 4.1.2.

Solution components were built to handle processing scenarios from regular loads (1000 events/day) to heavy loads (up to 500.000 events per system being introduced at once). The latter happens since every application that starts being monitored carries authentication data from Jan 2012 on, for historical analysis - and this batch load runs over night shifts. Demands for integration with systems that present constraints such as restricted schedules are satisfied with OfflineLoader implementation.

Quantitative metrics were collected on all Channel and EPA components in isolation. Reasons for that was validation of completeness and accuracy of the results, which depends on their interactions during tests.

Two major transitions of non-monitored systems to the solution happened while monitoring - one with 15 systems, another with 24 systems, indicating high loads of utilization events twice.

An offline, post-evaluation strategy was chosen. A separate message queue (MQ) was used as a sink component, keeping all event distribution information. This was further used to establish performance measurements and validate correctness on real data. For that, channel implementations were refactored so that all published and subscribed events, from all channels, were copied to new MQ topics, created for this validation (T_CH_PUB, T_CH_SUB). Also, code was injected in EPA implementations to trigger submission of messages to another MQ topic created for this validation (T_EPA_measurements), where every published message contains: processing start and end date time; and serialized representations of the output event.

After seven consecutive days collecting data, we evaluated each entry from T_EPA to: verify EPA processing time, under regular and heavy load processing scenarios; assert EPA expected outcomes were equivalent to the ones we obtained; confirm that all events that needed to be processed (from T_CH_PUB) were properly assigned (T_CH_SUB) to

EPAs.

Latency measurement modes, presented on Figure 4.8, were defined according to a benchmark study for load generation and performance measurement of CEP systems (MENDES; BIZARRO; MARQUES, 2008), establishing the following indicators, related to processing time:

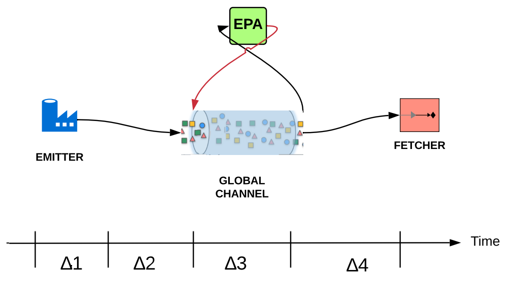


Figure 4.8: Latency measurements - adapted from Mendes, Bizarro e Marques (2013)

- Δ**1** - Event Generation Delay - from the moment events are produced, including the Event instantiation (Event Factory casting Event instance with composed parts), until the channel *publish* method is called;
- Δ**2** - Channel Conversion Delay - from the moment Channel publish is called, including binary transformation of event instances, until a call to the messaging platform is performed with converted event instance. Apache Avro[9] was used for the implementation of binary transformation;
- Δ**3** - CEP EPA Processing Time - from the moment event message is subscribed by the EPA and processed, until either an event is derived by this component (usual StatelessEPA scenarios) or an event is processed and placed on a buffer (usual processing cycle on StatefulEPA scenarios).
- Δ**4** - Fetcher Conversion Delay. Conversion from events from the Messaging Platform to Matomo API. Matomo is feed with specific derived events from GlobalChannel, and dispatches every 10 minutes a re-indexing process which is out of this scope.

For the sole capacity of RabbitMQ to absorb incoming events, store and distribute them, the maximum observed throughput, in terms of the amount of events passing through

[9]https://avro.apache.org

a channel per second, was close to 3000 events per second for publishing operations and 2000 events per second for subscribing operations.

Results from these measurements were indicated in Table 4.8.

Table 4.8: Processing Time Measurements

	Latency (*milliseconds*)
Δ1 - Event Generation Delay	
Average Time Group 1, 2, 3 & 4 Producers	0.1 - 1
Δ2 - Channel Conversion Delay	
Average Time Global / Segregated Channels	0.1
Δ3 - EPA Processing Time	
Average Time StatelessEPA	10 - 20
Average Time StatefulEPA	10 - 30
Average Time ClusteringEPA[i]	3000
Δ4 - Fetcher Conversion Delay	
Average Time Matomo Consumer	0.1 - 0.5

[i]Clustering: initialization step takes around three seconds, in addition to 0.2 seconds per iteration.

Achieved metrics greatly benefit from technical capacities by the chosen event distribution platform: RabbitMQ 3.7.5 (under Erlang 20.3.5), using AMQP[10] as the protocol from publishing messages. We also leveraged a Spark API for the consumption of event streaming from this platform (via Spark Stream, for RabbitMQ[11]). Spark was relevant for this implementation, since it increases the execution speed of streaming processes, speeding up the input/output, by storing the data in memory via RDD (Resilient Distributed Dataset) - its basic abstraction, that represents a partitioned collection of stream elements that can be operated in parallel.

Also, other requirements were met by this solution: advanced filtering via topic routing key patterns (RabbitMQ out-of-the-box feature - as we added group id and system id to the topic name we can further use expressions to filter them); continuous model training (ClusteringEPA) and reusability for both EventEmitter (via AMQP API) and EventFetcher

[10]Advanced Message Queuing Protocol (AMQP) is an open standard (ISO/IEC 19464) for passing business messages between applications and organizations

[11]An API that uses of Spark streaming strategy for consuming RabbitMQ messages - https://github.com/Stratio/spark-rabbitmq

(via Spark Stream).

Results indicate that, on higher load occasions, throughput scaled up to around 50 raw events being processed per second. This was measured by quantifying the number of incoming events from FILTER_G1 and FILTER_G234, at the beginning of Phase I processing, as described in Session 4.1.1.2, according to implementation documented on Session 4.1.3. The amount of events coming as the output from those filter EPAs was observed to between 3 and 20 times smaller than the input volumes, relieving the burden of the next processing stages.

We can observe, from Table 4.8, that the majority of the time is spent actually processing operations, which is reasonable, since we are leveraging technologies that saves time for formatting and distributing messages across CEP components.

We observed no compromise to integrity (all events were processed), nor any impact that caused unavailability of any node, resulting on an overall average processing time of 20 ms, although total round time since event introducing up to their aggregations are delivered to Matomo is highly impacted during batch loads. The first batch with around 40.000 historical events, related to 15 systems, was completed in less then 15 minutes, and the second one, with near 500.000 events for 19 systems, lasted less than three hours. For those cases (offline loading), $\Delta 1$ time can be ignored, since it starts with a batch of produced event instances. Under normal circumstances (where no historical data is processed in parallel) utilization occurrences are processed under one second.

Containers (observed via docker statistics[12]) indicated the need for improvements, especially related to CPU consumption - EPAs are created with a setting to limit its processing to 2 CPU cores, and on a few occasions it consumed all available capacity (especially while starting up each new container instance). Memory consumption was perceived to be between 100 MiB and 800 MiB (840 MB) and those evaluated indicators suggest CPU-bound operations were kept below the threshold.

In terms of correctness and integrity, all events were correctly assigned and processed by EPA nodes. Since there was no improper discarding of events (false negative), nor improper processing of events (false positive), both accuracy and precision metrics yielded

[12]https://docs.docker.com/engine/reference/commandline/stats/

1 (100%) as the result from analysis. Around 550.000 raw events (historical events and 10.000 contemporaneous events) and 350.000 derived events were considered in this analysis, which was simplified with the usage of correlation ids and attributes.

Accuracy for clustering processing was not evaluated here since the implementation delegates the calculation to Spark K-Means[13] solution, sending the last inferred set of centroids (as initial model) and the data set containing aggregated utilization values. As for performance, a Context was defined for CLUSTER_G1234 so that this process is only triggered every half an hour, with evaluation mode set as DEFERRED (to be processed at the end of the context window) due to the delay for Spark RDD set instantiation for system utilization values (the initialization step takes around three seconds, in addition to 0.2 seconds per iteration).

This test was provided in 3 servers configured with Intel Xeon processor (4 cores) and 30 GB RAM, running Docker containers[14] under CentOS Linux, managed via a Docker Swarm[15] orchestrator. Swarm features relevant aspects, such as scalability (guaranteed number of replicas), security (authenticated and encrypted communication between containers) an load balancing (distributing containers between hosts).

Instantiating Composite EPAs (as new Context partition emerges) or shutting them down (as a window terminate condition is reached) was performed in this solution by, respectively, gracefully starting (via a routine that injects context attributes into the VM) or stopping a service container according to a flag indication read by the container orchestrator.

RabbitMQ ran as a single container (with multiple queues) in one of them. A few optimizations on RabbitMQ settings, such as *transient messages*, *thread pool*, and *no acknowledgements* settings were performed, on top of network and OS tunings (for instance ARP caching thresholds). Further optimizations within containers such as data source caching for Global State instances were performed.

[13]https://spark.apache.org/docs/latest/mllib-clustering.htmlk-means

[14]A container is a standard unit of software that packages up code and all its dependencies so the application runs reliably from one computing environment to another. It also leverages the machine's OS system kernel and therefore do not require an OS per application, driving higher server efficiencies. Source: https://www.docker.com

[15]https://docs.docker.com/engine/swarm/

Purging from RabbitMQ erases queue members that have been published more than 10 days ago, but it was disabled during this test. All historical data is migrated and fetch from Matomo database, for historical purposes.

We compare our throughput result, 50 raw events per second, with achievements from academic papers - which use raw events per unit of time for throughput measurements. Examples of throughput indicators in other real-time streaming studies are: 6.000 tweets processed per hour (YADRANJIAGHDAM; YASROBI; TABRIZI, 2017), which is almost 2 events per second; 2500 rows of data for one complex prediction for the state of the wind turbines every 10 minutes (CANIZO et al., 2017), which is around 4 events per second; 500 sensor events per second (MAYER; MAYER; ABDO, 2017); 500 social network events per second (MANJUNATHA; MOHANASUNDARAM, 2018); and 10.000 image events per second (ICHINOSE et al., 2017).

We observe our result falls in the middle of a wide range of throughput indicators, which shows a potential of our model to design solutions that perform reasonably well - although performance was not the main goal of this research.

Finally, we assessed coupling and cohesion aspects for implemented components. Designs with low coupling and high cohesion, according to software engineering experts, lead to products that are more reliable and maintainable (FENTON; BIEMAN, 2014).

This evaluation shows how the model-driven instances from this scenario, depicted in UML object diagrams from Figures 4.9 and 4.10, aim to minimize coupling. To simplify the representation, only the main CEP agents were represented - but on further evaluation we considered all model dependencies.

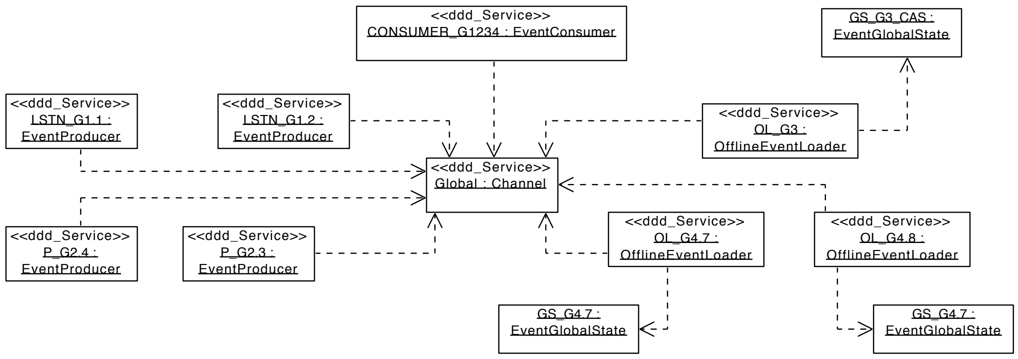

<<ddd_Service>>
CONSUMER_G1234 : EventConsumer
GS_G3_CAS : EventGlobalState
<<ddd_Service>> LSTN_G1.1 : EventProducer
<<ddd_Service>> LSTN_G1.2 : EventProducer
<<ddd_Service>> OL_G3 : OfflineEventLoader
<<ddd_Service>> Global : Channel
<<ddd_Service>> P_G2.4 : EventProducer
<<ddd_Service>> P_G2.3 : EventProducer
<<ddd_Service>> OL_G4.7 : OfflineEventLoader
<<ddd_Service>> OL_G4.8 : OfflineEventLoader
GS_G4.7 : EventGlobalState
GS_G4.7 : EventGlobalState

Figure 4.9: Producers and Consumer

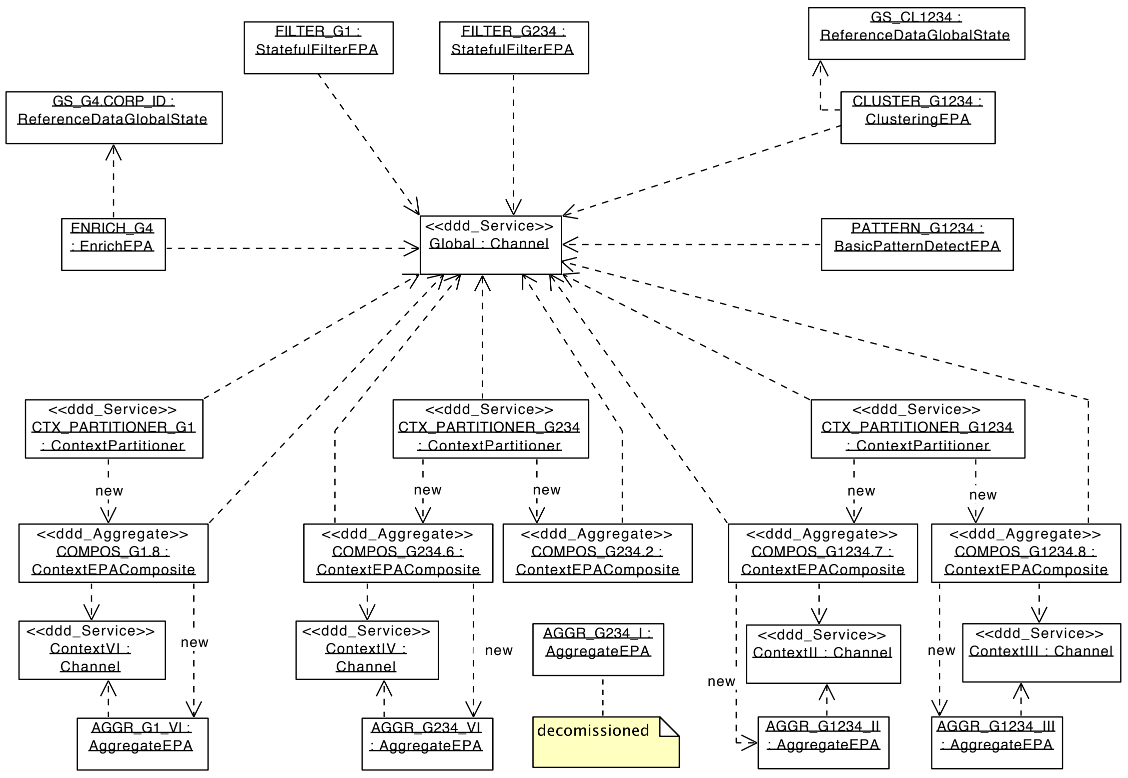

FILTER_G1 : StatefulFilterEPA
FILTER_G234 : StatefulFilterEPA
GS_CL1234 : ReferenceDataGlobalState
GS_G4.CORP_ID : ReferenceDataGlobalState
CLUSTER_G1234 : ClusteringEPA
ENRICH_G4 : EnrichEPA
<<ddd_Service>> Global : Channel
PATTERN_G1234 : BasicPatternDetectEPA
<<ddd_Service>> CTX_PARTITIONER_G1 : ContextPartitioner
<<ddd_Service>> CTX_PARTITIONER_G234 : ContextPartitioner
<<ddd_Service>> CTX_PARTITIONER_G1234 : ContextPartitioner
new
<<ddd_Aggregate>> COMPOS_G1.8 : ContextEPAComposite
<<ddd_Aggregate>> COMPOS_G234.6 : ContextEPAComposite
<<ddd_Aggregate>> COMPOS_G234.2 : ContextEPAComposite
<<ddd_Aggregate>> COMPOS_G1234.7 : ContextEPAComposite
<<ddd_Aggregate>> COMPOS_G1234.8 : ContextEPAComposite
<<ddd_Service>> ContextVI : Channel
<<ddd_Service>> ContextIV : Channel
AGGR_G234_I : AggregateEPA
<<ddd_Service>> ContextII : Channel
<<ddd_Service>> ContextIII : Channel
AGGR_G1_VI : AggregateEPA
AGGR_G234_VI : AggregateEPA
decomissioned
AGGR_G1234_II : AggregateEPA
AGGR_G1234_III : AggregateEPA

Figure 4.10: EPA Processing

Coupling is defined (FENTON; BIEMAN, 2014) as an attribute of pairs of objects. As a metric, it is proportional to the total number of couples that one object has with other objects (assuming that all basic couples are of equal strength).

The first aspect we highlight from the diagrams is how CEP Agents (Producer, EPA and Consumer instances) interact with each other. There is no direct inter-dependencies between them - all communications are established via the channel. Agents present no knowledge of the methods and attributes of other agents. Direct access between agents is avoided, therefore coupling for interactions among agents is minimized, and restricted to the interactions with the channel. Related coupling measurements that corroborate this aspect are described below.

CBO - Coupling Between Objects (CHIDAMBER; KEMERER, 1991; OTIENO; OKEYO; KIMANI, 2015): The number of times methods of one class use the methods or attributes of another class. Multiple accesses to the same object are counted as one access. Only method calls and variable references are counted. When this value is high, it indicates a high degree of class inter-dependency.

NDO - Number of Dependencies Out (OTIENO; OKEYO; KIMANI, 2015): defined as the number of classes that depend on a given class. When this value is high, it indicates reuse for the class.

RECBO - Run-time Export Coupling Between Objects (MITCHELL; POWER, 2004) : It defines the number of class instances accessing the methods of a class at runtime.

Table 4.9 present CBO and NDO measurements based on referenced classes indicated in our CEP model, according to Section 3.3 descriptions. According to Shatnawi (2010), their values (less or equal to 9) indicate low coupling. RECBO was observed according to objects from Figures 4.9 and 4.10. The rational used for its elaboration is demonstrated in Appendix C.

This analysis requires us to analyze a different kind of coupling, since every CEP agent is bound to the event distribution platform through the channel (see observed number of active agents in NDO metric from Table 4.9).

To deal with Channel overhead, technical optimizations were put in place, and isolated measurements on event publishing and subscribing operations show there was no compromise w.r.t interacting with stream platform.

We foresee that the contract established via public methods and attributes is likely to remain the same, despite technical choices of message distribution platforms or stream processing technologies (Spark, Kafka, etc.). Rather than that, specific implementations to interact with other technologies, via their API, is subject to change without affecting its contract - which is less harmful to coupling. Therefore, this indicates stability of dependencies - a situation where application concepts are well defined and the implementations of those concepts are stable(HITZ; MONTAZERI, 1995).

On the other side, CEP agents logic, which varies according to business domain scenarios (a category considered with higher degrees of coupling, according to Hitz e Montazeri (1995)), are only bound to each other via the channel, showing a lower risk of impact as they go through modifications. We can also observe that dynamic EPAs provisioned under

Table 4.9: Coupling-Related Measurements

Class	**Object(s)**	**CBO**	**NDO**	**RECBO**
EventProducer	LSTN_G1.1, LSTN_G1.2, P_G2.3, P_G2.4	5	0 [i]	0 [i]
OfflineEventLoader	*OL_G4.7, OL_G4.8, OL_G3*	7	0 [i]	0~[i]
EventGlobalState	*GS_G3_CAS, GS_G4.7, GS_G4.8*	3	2 [ii]	1 [ii]
EventConsumer	*CONSUMER_G1234*	5	0 [i]	0 [i]
ReferenceDataGlobalState	*GS_G4.CORP_ID, GS_CL1234*	5	2 [ii]	1 [ii]
EnrichEPA	*ENRICH_G4*	9	1 [iii]	0 [iv]
StatefulFilterEPA	*FILTER_G1, FILTER_G234*	7	1 [iii]	0 [iv]
ClusteringEPA	*CLUSTER_G1234*	8	1 [iii]	0 [iv]
ContextEPAComposite	*COMPOS_G1.8, COMPOS_G234.6, COMPOS_G234.2, COMPOS_G1234.7, COMPOS_G1234.8*	9	1 [iii]	0 [iv]
AggregateEPA	*AGGR_G1_VI, AGGR_G234_I, AGGR_G234_VI, AGGR_G1234_II, AGGR_G1234_III*	8	1 [iii]	0 [iv]
Channel	*Global, ContextII, ContextIII, ContextIV, ContextVI*	5	$\propto c$ [v]	$\propto a$ [vi]

[i] No other class from the model indicates dependency for this class.

[ii] Either GlobalStateProvider factory (initially, at creation) or a CEP agent (after creation) references this instance at run-time (both are valid when counting classes.)

[iii] References this class, at creation time: EPAComposite, ContextEPAComposite, or ContextPartitioner.

[iv] After creation, EPAs are not referenced by other model components.

[v] Class static references to this class: this scales according to the number of chosen EPA Specializations (e): our model (Section 3.3) shows currently 20 classes bound to Channel (where 12 of them are EPA specializations).

[vi] The amount of dynamic references to this class is proportional to the number of agents (a): as depicted in Figures 4.9 and 4.10, it scales to 21 active agents, plus one ChannelBroker. In reality, we reached an approximate number of 100 active agents

composed contexts (aggregate compositions, show in grey, in Figure 4.6) follow a similar design used for non-composed EPAs - they both pull occurrences (the latter from an inner channel, instead of global channel), process and output results to their linked channel - *i.e.*, fully independent of other agents.

We also evaluated the degree of cohesion for the Channel class, in order to understand if CEP agents could benefit from further segregation of its responsibilities. We used the Lack of Cohesion in Methods (LCOM) metric as defined by Hitz e Montazeri (1995). It consists of the number of pairs of methods operating on disjoint sets of instance variables, and is therefore an inverse measure for cohesion (the higher are values, higher is the indication a refactoring is needed for a class). Here we reproduce the definition:

Let X denote a class, IX the set of its instance variables of X, and MX the set of its methods.

Consider a simple, undirected graph GX(V, E) with V = MX, and

$$E = \langle m, n \rangle \in \mathbb{V}\times\mathbb{V} | \, (\exists i \in IX : (m \textit{ accesses } i) \wedge (n \textit{ accesses } i)) \\ \vee (m \textit{ calls } n) \vee (n \textit{ calls } m)$$

LCOM(X) is then defined as the number of connected edges of GX

We calculated the metric by creating a list of Channel methods and attributes, and then analyzed their references to each other. Functional Interfaces are treated as methods, whose implementation is provided via lambda expressions.

Figure 4.11 shows the evaluation of LCOM for the Channel class. This picture demonstrates the methods *publish* and *consume* are calling the *encodeEvent* and *decodeEvent*, and since they share an instance of *streamPlatformConnectivitySettings* (to establish connectivity to the stream platform), we can see no disjoint set of edges arising from the graph - therefore LCOM = 1, an indication of high cohesion.

We also observe that the *StreamConsumer* function implementation encompasses EPA specific variations, triggering processing stream pipelines according to each EPA responsibility. This function is evaluated under the occurrence of subscribed events, and as a result it may trigger publishing of derived events, therefore constituting a cohesive flow. Other CEP agents also present a high cohesion degree - EventConsumer, acting upon occurrences fetch from *subscribeEvents* (from EventFecther), and EventProducer, for which event introduction is performed for via *publish* (from EventEmitter).

4.2 Feedback from Industry Experts

In order to evaluate a theoretical conjecture - "a model that represents EPA compositions, incorporating aspects such as stream processing, segregation based on context, historical data processing and incremental training, fulfills requirements for building Re-

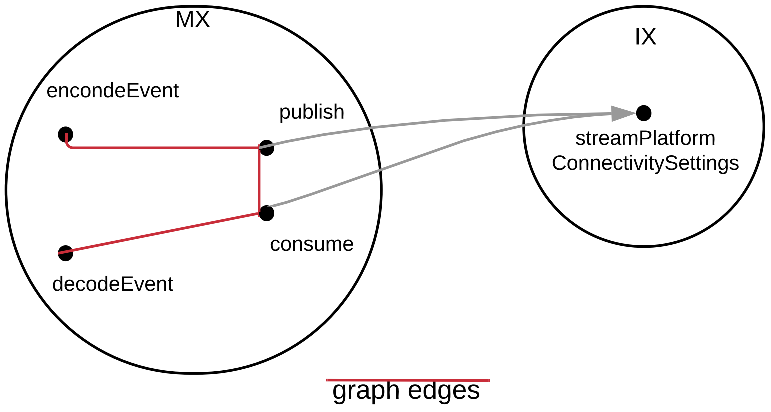


Figure 4.11: LCOM for Channel

altime Streaming Analytics solutions", we elaborated a workshop with experienced system architects.

Evaluators were selected based to their academic and work experience - the requirement was pursuing a graduation degree related to Computer Science and experience in modelling and implementing solutions that handle analysis of high volume of data transactions close to their occurrence, with at least one solution in production.

The skills required for this study restricted the possibilities to find adequate participants for the evaluation, and we were able to gather five participants in our workshop. All chosen professionals work for large size companies, and they are experienced professionals (more then 10 years of experience building systems, at least 5 in the field of system architect). We also tried to compensate the low number of selected participants by choosing them over different companies and industry sectors - two of them work in Finance area, one in Telecommunication and two in Oil & Gas.

4.2.1 Workshop and Evaluation Criteria

The workshop was divided in three parts, described as follows.

On the first part, we introduced the CEP components (from Section 2.1), the DDD layers (Section 2.2.2) and the logic behind the compartmentalization of CEP components according to those layers, presenting our overall diagram (Section 3.2). Then we went through the details for each model partition, from Section B.1 to Section 3.8. We also provided a brief presentation on the applicability of our model on a real industry use case (Section 4.1).

On the second part, we explained the requirements for evaluating our Model - the 13 realtime streaming analytics patterns (Section 2.2.4), along with examples provided in Perera e Suhothayan (2015). At the end of the session, we asked the participants to fill a questionnaire with two sets of questions.

For the first set of questions (presented in Table 4.10, we inquired participants if they understood: the model explanation; the requisites being evaluated (*i.e.*, the analytics patterns); the rules for evaluation; and whether they considered the evaluation reasonable. They were asked to respond with a YES or NO answer.

Table 4.10: First Set of Questions

Q_ACK_1	Was the model explanation clear?
Q_ACK_2	Are the requisites being evaluated clear?
Q_ACK_3	Are the evaluation rules clear?
Q_ACK_4	Do you consider this evaluation reasonable?

For the second set, the questions were elaborated by following GQM strategy (SOLINGEN et al., 2002), based on goals, questions and metrics. For that, we developed a plan beginning by clarifying the specific target, the objects being measured, and the context in which the measures are analyzed (presenting the goal, in Table 4.11). We aimed to evaluate if our proposed model was capable to meet a set of thirteen requirements of real time streaming analytics (PERERA; SUHOTHAYAN, 2015). A questionnaire was then presented with thirteen questions (Table 4.12). Those questions were then correlated to evaluation metrics (Table 4.13).

The participants were asked to evaluate the adequacy of the model for each one of the thirteen patterns, on the second set of questions, by:

- marking, for each pattern, their perception of adequacy on a scale that ranged from

Table 4.11: Main Goal Definition

Main Target	Validate the applicability of the model in a relevant CEP scenario
Purpose	Outcome from experts on the applicability of model for real-world industry streaming analytics solutions, a major application area in CEP
Quality Measurement	Evaluate the model based on feedback from application architects in regards to the model and ways to design it according to each one of the thirteen streaming analytics solution patterns (PERERA; SUHOTHAYAN, 2015)

irrelevant (0) to relevant (3), with no neutral position on this scale, being:

- 0 does not meet the acceptance criteria;

- 1 partially meets the acceptance criteria with severe restrictions;

- 2 meets the acceptance criteria with minor restrictions;

- 3 fully meets the acceptance criteria;

• providing a textual description to justify the answer above, for instance by exemplifying a possible application of the model that meets the requirement.

After collecting the answers, we proceeded to the third part of the workshop. At this stage, we debated the justifications for the answers, aiming to reach a consensus with regards to the capacity of our proposal to meet demands of each streaming analytic pattern.

4.2.2 Results

The five participants (P1 to P5) indicated positive feedbacks for all questions on the first set of questions, being M_{set1_exp}, M_{set1_req}, $M1_{set1\_rul}$ and M_{set1_rea} all equal to 5.

On the second set, we observed the following to be true:

$\forall i, j$, where $1 \geq i \geq 13$ and $0 \geq j \geq 3$, $M_{set2\_ij} \geq 2$

This is because we had no participants who assigned a grade less then 2 (see Table 4.13). The majority of the answers for this set received a grade 3, and examples were provided on each answer, in terms of the application of the model according to scenarios that correlate to the goal assigned for each one of the analytics pattern.

Table 4.12: Second Set of Questions

Q_PATTERN_1	Does the model support projections from one data stream into other data streams?
Q_PATTERN_2	Does the model support alerts generation based on simple and complex conditions, such as rate of increase?
Q_PATTERN_3	Does the model support simple counting and counting within event windows via aggregate functions?
Q_PATTERN_4	Does the model support joining multiple data streams into a new event stream?
Q_PATTERN_5	Does the model support data correlations for detecting missing or erroneous events in a data stream, and acting on their occurrence?
Q_PATTERN_6	Does the model support processes combining real-time data and historical data persisted in a data source?
Q_PATTERN_7	Does the model support the detection of temporal patterns for event sequence?
Q_PATTERN_8	Does the model support tracking objects over space and time dimensions, detecting given conditions?
Q_PATTERN_9	Does the model support the detection of patterns and trends from time series data and bringing information such as outliers into operator attention?
Q_PATTERN_10	Does the model support running processes both in batch mode as well as in realtime pipelines?
Q_PATTERN_11	Does the model support detecting a high number of abnormal occurrences, and further analyze it using historical data?
Q_PATTERN_12	Does the model support usage of a prediction model which, once properly trained, can be used within the realtime pipeline to infer information
Q_PATTERN_13	Does the model support usage of automatized control for problems such as situation awareness?

Table 4.13: Metrics for Questions

	First Set
M_{set1_exp}	Number of people who acknowledged understanding from model explanation
M_{set1_req}	Number of people who acknowledged understanding from requisites from patterns being evaluated
M_{set1_rul}	Number of people who acknowledged understanding rules for the questionnaire evaluation
M_{set1_rea}	Number of people who considered the questionnaire evaluation reasonable
	Second Set
M_{set2_i0}	For Q_i, number of people who chose 0 in the quality scale for the related perception answer
M_{set2_i1}	For Q_i, number of people who chose 1 in the quality scale for the related perception answer
M_{set2_i2}	For Q_i, number of people who chose 2 in the quality scale for the related perception answer
M_{set2_i3}	For Q_i, number of people who chose 3 in the quality scale for the related perception answer

Only 4 cases out of the 65 answers were evaluated with a grade 2 (meaning it still meets the acceptance criteria, with minor restrictions), as per Figure 4.12.

The justified answers were further discussed among all participants, and invalid considerations were not perceived. They relate to the following observations:

- Pattern 4 : Join - On top of combining multiple data streams and create a new event stream as a result, the need to perform joins across event streams based on correlations involving different data formats (audio, video) and attributes (as geolocation) - it was not directly approached, and it may require further extensions. For this matter, we have prototyped an EPA that consumes a picture and dispatches an automatic license plate recognition algorithm to infer an event consisting of set of plate numbers, along with their probability.
- Pattern 9 : Detecting Trends - This is possible via PatternDetectEPA, and an extension is provided to demonstrate the possibility of automatic learning for the prediction of clusters. However further extensions are required as we aim to incorporate functionalities not evaluated in this study, such as supervised machine learning.
- Pattern 13 : Online Control - We may need a practical evaluation for use cases that involve complex online control scenarios (*e.g.* self-driving car) involving problems as deciding on corrective actions, which may require more functionalities than what it has been designed to do.

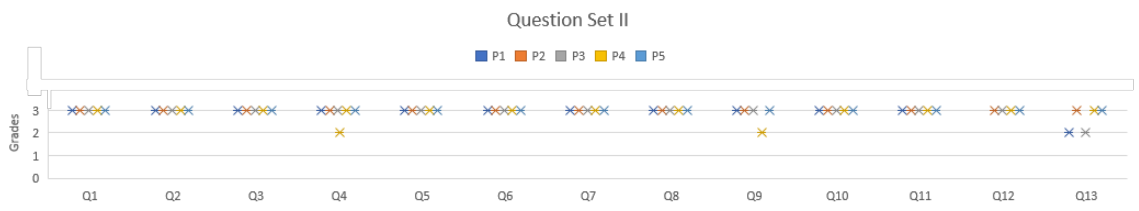


Figure 4.12: Chart - Answers from Second Set of Questions

We can infer potential benefits from the proposed model by observing the convergence in the answers from the five participants. Results indicate the model fully complies to 10 out of 13 patterns, and meets acceptance criteria to remaining ones - whereas, for EPTS Reference Architectural Models (described in Section 2.2.1), only 9 out of 13 patterns apply, according to Perera e Suhothayan (2015), Cugola e Margara (2012).

Patterns covered by this model (missing on EPTS models), relate to the following features:

- Pattern 6: Interacting with Historical Databases - Combining the realtime data processing against the historical data stored in any source is possible by leveraging features from EPA Compositions that involve elements such as EnrichEPA (Section 3.6.1.1) and AggregateEPA (Section B.8), which provide stream processing operations with assistance of Global State (Section 3.8);
- Pattern 10: Same Processing Mechanisms in Batch and Realtime Pipelines - The adoption of OfflineLoader (Section 3.6.3) according to EPA and EventFetcher stream subscribing mechanisms (Section 3.5.2) fulfills the need to process historical data, loaded via batch operations, according the same processing means.
- Pattern 11: Detecting and switching to Detailed Analysis - this pattern is used with the use cases where we cannot analyze all the data in full detail. Anomalous behavior can be triggered as a new event inferred by PatternDetectEPA (Section 3.6.1.3), such as an interlock alert coming from an industrial sensor that requires immediate corrective actions. This event can be consumed within an EPA composition that segregates those occurrences from the Global Channel, where an inner channel that relates to high-priority context data dispatches specific alarm occurrences so that proper actions can be taken.
- Pattern 13: Online Control - This relates to problems that require current situation awareness, based on predicted values. PatternDetectEPA and extensions such as the one provided by ClusteringEPA (Section 3.6.1.4) allow the incremental inference of patterns in real time, assisting on decisions based on models that vary over time.

Also, a benefit introduced in this model can be observed on Pattern 1, where filtering operations can be enhanced through upfront projection mechanisms for consumer components (*e.g.*, via SubscriptionPattern from Sections 3.5.2 and 3.7.1).

5. Conclusion

This chapter presents the conclusion of this study, its contributions, as well as the limitations of the approach and future work.

5.1 Final considerations

Elucidation of how EPA agents can be composed in a CEP solution assists on designing CEP architecture platforms based on inter-operable, reusable components. This is relevant for identifying design strategies that addresses industry CEP requirements and their challenges, due to their demands in terms of distributed data processing and integration with increasing levels of complexity.

5.2 Contributions

The main contribution of this work is the proposed architectural model. This model addressed the main weakness of literature with regard to EPA composition, which is a clear guidance for organizing those elements according to business needs. It aims to minimize its complexity by establishing clear responsibility assignments for components, as well as by highlighting and isolating common functionalities.

Our proposal enhances current CEP representations by:

- Establishing an innovative way to represent EPA compositions via global and segregated inner Channel instances (Section 3.6.1), and to clarify dynamic EPA compositions through arrangements of existing aggregation structures, based on context partitions (Section 3.4.2);

- Representing state of the art event processing strategies in CEP, such as advanced filtering (Section B.8), offline introduction of events (Section 3.5.1), incremental model training (Section B.50) and stream processing (Sections 3.6.1 and 3.7.1);
- Clarifying and reorganizing the hierarchy of EPA types, for instance by providing reuse of StatefulEPAs on SplitEPA (Section B.8), and by handling event processing via stream processing pipelines (Section 3.7.1).

An evaluation of the capacity of the proposed model to meet requirements for realtime analytics patterns indicated benefits from its usage, as we compare it with EPTS architectural models, especially since it addresses solutions for patterns not yet covered by those models, achieved due to introduced features such as stream grouping based on context, processing of streams of events and offline processing of event batches.

Also, an analysis of the results provided based on a use case experiment, demonstrating the metric outcomes from the solution within an industry real use case. The model guides us to implement cohesive components that integrate through a platform (via message streaming) of events, and agents presented a low degree of coupling. Their construction was also driven by container mechanisms, allowing us to provision and decommission services based on required circumstances (such as windows triggering conditions based on context). Other benefits perceived from this solution include: the ability to meet the demand for integration with systems that present constraints such as restricted schedules (via offline loads); advanced message filtering (via platform features); continuous model training (with K-means) and high performance and reusability for emitting and pulling events via streaming operations provided by specialized components. This analysis can serve as a guidance to elaborate other solutions, depending on the architecture application scenarios.

5.3 Limitations and Threats to validity

Despite our goal to simplify the construction of CEP solutions, the model itself presents a degree of complexity, encompassing 6 layers with approximately 60 concepts. This was a trade-off, in order to be able to express different composition scenarios when designing CEP solutions. The experiment on Section provides guidelines and suggestions on how and when to apply the model features.

The case study from Section 4.1 covers a scenario where we explored the elaboration of an architecture. The fact researchers from this study were involved in the implementation poses a possible bias. We mitigate it by collecting feedback from industry experts.

We observed the round time (as a result of total performance) was impacted during two batch loads. It did not compromise our results given the business expectations, but shows an area for improvement, given the demand for computing resources. The initialization time for running (spawning SpringBoot[1]-based containers via runc[2]) presents an opportunity for improvement (it got close to 2 seconds with JVM optimizations). Despite of that, the capability to scale up and down the number of containers according to business demands, which relate to processes and threads sharing our hosts' resources, indicate the technology choice was appropriate for the effectiveness related to our results.

Regarding the choice of Domain-Driven Design principles - another paradigm, named Model-Driven Architecture (MDA) (KLEPPE et al., 2003) was also considered as a methodical approach to define reusable assets. However, it was not used for the model elaboration, since this strategy requires more emphasis on transforming a model into code then clarifying how to correlate the model components based on meaningful contexts. Also, defining the model components requires the analysts to establish model components based on their perception of business domain scenarios, so DDD was naturally applied as guidance. MDA, though, is fully applicable on further stages, when we plan to derive software components from the proposed model.

Although quantitative metrics were observed for only one case study experiment, and comparison against other relevant studies is subjected to interpretation of the nature of

[1]https://spring.io/projects/spring-boot
[2]https://github.com/opencontainers/runc

processing operations, implementation showed potential gains from our model, especially if we consider the feedback from industry experts on top of the measurements. We are also planning to the model on other projects (discussed in the next session) with different settings: one bringing a greater amount of events (from automation sensors), and another one bringing events captured from geographically dispersed locations, as images, from camera sensors.

The feedback from Section 4.2 indicated minor restrictions related to the need for extending some of the model components. In the same section, we indicated the reasons why we were constrained to five participants in our evaluation.

5.4 Future work

The proposed model needs to be refined to encompass other relevant scenarios. It is currently being evaluated in two distinct projects, described below.

One of the projects collects high amounts of data from automation devices and periodically infers predictive maintenance information (based on factors such as vibration and pressure measurements). In this case we have 850 sensors and actuators providing information at least every tenth of a second. Even if we capture, via operational historian[3] aggregated measurements (such such as average and median values) per second, it still results in more then 70 million readings per day - a stronger requirement to process higher amounts of data.

Another case integrates with storage devices fed by several sensor cameras, and infers license plate numbers from vehicles entering or leaving a restricted area, further crossing this data with corporate information. Here, sensors are geographically spread across Brazil, and we are sending the images to a central queue - we might benefit if our computer vision processing could be performed closer to the sensor location, extract the list of plate numbers (with highest probabilities) and send them as text to the central queue, reducing network latency and optimizing overall response time. The distribution of processing over a large geographic extent, on top of architectural aspects discussed in CEP, brings an inter-

[3]A time-series data store that integrates with control systems and captures readings from all sensors - e.g. https://www.ge.com/digital/applications/historian

esting perspective, that has been a field of study in System of systems engineering (SoSE) and was not yet addressed in this study, but may bring valuable contributions - SoSE deals with networks of heterogeneous systems that exhibit geographical distribution, operational and managerial independence, and emergent and evolutionary behaviors that would not be apparent if the systems and their interactions were modeled separately (MAIER, 1998).

Also, we plan to elaborate a process to guide architects and developers on implementations of the model, which can be applied over new scenarios.

Bibliography

ARTIKIS, A. et al. Event processing under uncertainty. In: ACM. *Proceedings of the 6th International Conference on Distributed Event-Based Systems*. [S.l.], 2012. p. 32--43.

BAPTISTA, G. et al. A middleware for data-centric and dynamic distributed complex event processing for iot real-time analytics in the cloud. In: *34th Brazilian Symposium on Computer Networks and Distributed Systems, Salvador, Brazil*. [S.l.: s.n.], 2016.

BASS, T. *Fraud detection and event processing for predictive business*. 2006. <https://www.researchgate.net/publication/306894315_Fraud_Detection_and_Event_Processing_for_Predictive_Business>.

BAUER, A.; WOLFF, C. An event processing approach to text stream analysis: basic principles of event based information filtering. In: ACM. *Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems*. [S.l.], 2014. p. 35--46.

BAUMGÄRTNER, L. et al. Complex event processing for reactive security monitoring in virtualized computer systems. In: ACM. *Proceedings of the 9th ACM International Conference on Distributed Event-Based Systems*. [S.l.], 2015. p. 22--33.

BINNEWIES, S.; STANTIC, B. Oecep: enriching complex event processing with domain knowledge from ontologies. In: ACM. *Proceedings of the Fifth Balkan Conference in Informatics*. [S.l.], 2012. p. 20--25.

CANIZO, M. et al. Real-time predictive maintenance for wind turbines using big data frameworks. *arXiv preprint arXiv:1709.07250*, 2017.

CARBONE, P. et al. Apache flink: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, IEEE Computer Society, v. 36, n. 4, 2015.

CHIDAMBER, S. R.; KEMERER, C. F. Towards a metrics suite for object oriented design. Cambridge, Mass.: Center for Information Systems Research, 1991.

CUGOLA, G.; MARGARA, A. Processing flows of information: From data stream to complex event processing. *ACM Computing Surveys*, ACM, v. 44, n. 3, p. 15, 2012.

DAVIS, A. L. Reactive streams in java: Concurrency with rxjava, reactor, and akka streams. Apress, 2018.

DAYARATHNA, M.; PERERA, S. Recent advancements in event processing. *ACM Computing Surveys (CSUR)*, ACM, v. 51, n. 2, p. 33, 2018.

DOBBELAERE, P.; ESMAILI, K. S. Kafka versus rabbitmq: A comparative study of two industry reference publish/subscribe implementations: Industry paper. In: ACM. *Proceedings of the 11th ACM International Conference on Distributed and Event-based Systems*. [S.l.], 2017. p. 227--238.

D'SILVA, G. M. et al. Real-time processing of iot events with historic data using apache kafka and apache spark with dashing framework. In: IEEE. *Recent Trends in Electronics, Information & Communication Technology, 2017 2nd IEEE International Conference on*. [S.l.], 2017. p. 1804--1809.

ETZION, O.; NIBLETT, P.; LUCKHAM, D. C. *Event processing in action*. [S.l.]: Manning Greenwich, 2011.

EVANS, E. *Domain-driven design: tackling complexity in the heart of software*. [S.l.]: Addison-Wesley Professional, 2004.

FALK, E.; GURBANI, V. Query-able kafka: An agile data analytics pipeline for mobile wireless networks. *Proceedings of the VLDB Endowment*, VLDB Endowment, v. 10, n. 12, p. 1646--1657, 2017.

FAWCETT, T. An introduction to roc analysis. *Pattern recognition letters*, Elsevier, v. 27, n. 8, p. 861--874, 2006.

FENTON, N.; BIEMAN, J. *Software metrics: a rigorous and practical approach*. [S.l.]: CRC press, 2014.

FREEMAN, J. *Introducing streaming k-means in spark 1.2*. 2015. <https://docs.oracle.com/cd/E17904_01/doc.1111/e14476/preface.htm>.

GAMMA, E. et al. *Design patterns: Elements of reusable software components*. [S.l.]: Addison-Wesley Professional, 1995.

HITZ, M.; MONTAZERI, B. *Measuring coupling and cohesion in object-oriented systems*. [S.l.]: Citeseer, 1995.

ICHINOSE, A. et al. A study of a video analysis framework using kafka and spark streaming. In: IEEE. *2017 International Conference on Big Data*. [S.l.], 2017. p. 2396--2401.

KHARE, S. et al. Reactive stream processing for data-centric publish/subscribe. In: ACM. *Proceedings of the 9th ACM International Conference on Distributed Event-Based Systems*. [S.l.], 2015. p. 234--245.

KLEPPE, A. G. et al. *MDA explained: the model driven architecture: practice and promise*. [S.l.]: Addison-Wesley Professional, 2003.

KOLCHINSKY, I.; SHARFMAN, I.; SCHUSTER, A. Lazy evaluation methods for detecting complex events. In: ACM. *Proceedings of the 9th ACM International Conference on Distributed Event-Based Systems*. [S.l.], 2015. p. 34--45.

LEBERKNIGHT, S. Polyglot persistence. *Near infinity[Online], Available: http://www. nearinfinity. com/blogs/scott_leberknight/polyglot_persistence. html*, 2008.

LINDGREN, P.; PIETRZAK, P.; MÄKITAAVOLA, H. Real-time complex event processing using concurrent reactive objects. In: IEEE. *2013 IEEE international conference on industrial technology*. [S.l.], 2013. p. 1994--1999.

LUCKHAM, D. *The power of events*. [S.l.]: Addison-Wesley Reading, 2002. v. 204.

MAEDA, K. Performance evaluation of object serialization libraries in xml, json and binary formats. In: IEEE. *Digital Information and Communication Technology and it's Applications, 2012 Second International Conference on*. [S.l.], 2012. p. 177--182.

MAIER, M. W. Architecting principles for systems-of-systems. *Systems Engineering: The Journal of the International Council on Systems Engineering*, Wiley Online Library, v. 1, n. 4, p. 267--284, 1998.

MANJUNATHA, H.; MOHANASUNDARAM, R. Brnads: Big data real-time node anomaly detection in social networks. In: IEEE. *2018 2nd International Conference on Inventive Systems and Control*. [S.l.], 2018. p. 929--932.

MARGARA, A.; SALVANESCHI, G. Ways to react: Comparing reactive languages and complex event processing. *REM*, p. 14, 2013.

MAYER, C.; MAYER, R.; ABDO, M. Streamlearner: Distributed incremental machine learning on event streams: Grand challenge. In: ACM. *Proceedings of the 11th Int. Conference on Distributed and Event-based Systems*. [S.l.], 2017. p. 298--303.

MENDES, M.; BIZARRO, P.; MARQUES, P. A framework for performance evaluation of complex event processing systems. In: ACM. *Proceedings of the Second International Conference on Distributed Event-Based Systems*. [S.l.], 2008. p. 313--316.

MENDES, M.; BIZARRO, P.; MARQUES, P. Fincos: benchmark tools for event processing systems. In: ACM. *Proceedings of the 4th ACM/SPEC International Conference on Performance Engineering*. [S.l.], 2013. p. 431--432.

MIAN, P. et al. A systematic review process for software engineering. In: *ESELAW'05: 2nd Experimental Software Engineering Latin American Workshop*. [S.l.: s.n.], 2005.

MITCHELL, Á.; POWER, J. F. An empirical investigation into the dimensions of run-time coupling in java programs. In: TRINITY COLLEGE DUBLIN. *Proceedings of the 3rd international symposium on Principles and practice of programming in Java*. [S.l.], 2004. p. 9--14.

MOXEY, C. et al. *A Conceptual Model for Event Processing Systems*. 2010. Http://www.redbooks.ibm.com/abstracts/redp4642.html.

NADAREISHVILI, I. et al. *Microservice Architecture: Aligning Principles, Practices, and Culture*. [S.l.]: O'Reilly Media, Inc., 2016.

NECHIFOR, S. et al. Predictive analytics based on cep for logistic of sensitive goods. In: IEEE. *2014 International Conference on Optimization of Electrical and Electronic Equipment*. [S.l.], 2014. p. 817--822.

OLLESCH, J.; HESENIUS, M.; GRUHN, V. Engineering events in cps-experiences and lessons learned. In: IEEE. *2017 IEEE/ACM 3rd International Workshop on Software Engineering for Smart Cyber-Physical Systems*. [S.l.], 2017. p. 3--9.

ORACLE. *Overview of Oracle Complex Event Processing*. 2010. <https://docs.oracle.com/cd/E21764_01/doc.1111/e14476/overview.htm#CEPGS119>.

OTIENO, C.; OKEYO, G.; KIMANI, S. Coupling measures for object oriented software systems-a state-of-the-art review. *International Journal Of Engineering And Science*, v. 4, p. 01--10, 2015.

PAPAZOGLOU, M. P. Foresight & research priorities for service oriented computing. In: *ENASE*. [S.l.: s.n.], 2009. p. 5--6.

PASCHKE, A.; VINCENT, P. A reference architecture for event processing. In: ACM. *Proceedings of the Third ACM International Conference on Distributed Event-Based Systems*. [S.l.], 2009. p. 25.

PERERA, S.; SUHOTHAYAN, S. Solution patterns for realtime streaming analytics. In: ACM. *Proceedings of the 9th ACM International Conference on Distributed Event-Based Systems*. [S.l.], 2015. p. 247--255.

RAY, M.; LEI, C.; RUNDENSTEINER, E. A. Scalable pattern sharing on event streams. In: ACM. *Proceedings of the 2016 International Conference on Management of Data*. [S.l.], 2016. p. 495--510.

RENNERS, L.; BRUNS, R.; DUNKEL, J. Situation-aware energy control by combining simple sensors and complex event processing. 2012.

RISCH, J.-C.; PETIT, J.; ROUSSEAUX, F. *Ontology-based Supervised Text Classification in a Big Data and Real Time Environment*. [S.l.]: IEEE CoDIT 2106.

SCHMIDT, D. C. Model-driven engineering. *COMPUTER-IEEE COMPUTER SOCIETY-*, Citeseer, v. 39, n. 2, p. 25, 2006.

SHARP, J. et al. Data access for highly-scalable solutions: Using sql, nosql, and polyglot persistence. Microsoft patterns & practices, 2013.

SHATNAWI, R. A quantitative investigation of the acceptable risk levels of object-oriented metrics in open-source systems. *IEEE Transactions on software engineering*, IEEE, v. 36, n. 2, p. 216--225, 2010.

SOLINGEN, R. V. et al. Goal question metric (gqm) approach. *Encyclopedia of software engineering*, Wiley Online Library, 2002.

STOJANOVIC, N. et al. Mobile cep in real-time big data processing: challenges and opportunities. In: ACM. *Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems*. [S.l.], 2014. p. 256--265.

TEYMOURIAN, K.; PASCHKE, A. Enabling knowledge-based complex event processing. In: ACM. *Proceedings of the 2010 EDBT/ICDT*. [S.l.], 2010. p. 37.

THÖNE, S.; DEPKE, R.; ENGELS, G. Process-oriented, flexible composition of web services with uml. In: SPRINGER. *International Conference on Conceptual Modeling*. [S.l.], 2002. p. 390--401.

VELASCO, C. A.; MOHAMAD, Y.; ACKERMANN, P. Architecture of a web of things ehealth framework for the support of users with chronic diseases. In: ACM. *Proceedings of the 7th International Conference on Software Development and Technologies for Enhancing Accessibility and Fighting Info-exclusion*. [S.l.], 2016. p. 47--53.

VIDYASANKAR, K. On continuous queries in stream processing. *Procedia Computer Science*, Elsevier, v. 109, p. 640--647, 2017.

VILLAÇA, L. H.; AZEVEDO, L. G.; BAIÃO, F. Query strategies on polyglot persistence in microservices. In: ACM. *Proceedings of the 33rd Annual ACM Symposium on Applied Computing*. [S.l.], 2018. p. 1725--1732.

WHELAN, C.; HARRELL, G.; WANG, J. Understanding the k-medians problem. In: THE STEERING COMMITTEE OF THE WORLD CONGRESS IN COMPUTER SCIENCE. *Proceedings of the International Conference on Scientific Computing*. [S.l.], 2015. p. 219.

YADRANJIAGHDAM, B.; YASROBI, S.; TABRIZI, N. Developing a real-time data analytics framework for twitter streaming data. In: IEEE. *Big Data (BigData Congress), 2017 IEEE International Congress on*. [S.l.], 2017. p. 329--336.

ZIMMERLE, C.; GAMA, K. A web-based approach using reactive programming for complex event processing in internet of things applications. In: ACM. *Proceedings of the 33rd Annual ACM Symposium on Applied Computing*. [S.l.], 2018. p. 2167--2174.

A. Literature Review

This chapter presents the protocol for this review - a systematic revision, based on a series of standards and templates available (MIAN et al., 2005). Its purpose is to identify academic researchers' considerations related to the design of EPA compositions in Complex Event Processing topics, and correlated discussions on Streaming solutions.

A.1 Planning

A.1.1 Question Focus:

Identify articles that discuss model components related to complex event processing (CEP), allied to Streaming technologies, in order to subsidize the construction of a model that represents the characteristics of the data involved in this process in a manner consistent with the state of the art.

A.1.2 Question Quality and Amplitude:

A.1.2.1 Problem:

Software architectures dealing with complex event processing have a high degree of complexity, especially because of the nature of the operations involved (polyglot data integrations with characteristics of high variability, high frequency of occurrence, large volumes of information), and there is no universal solution that can efficiently resolve all scenarios. Publications suggest a generic CEP processing model, to guide architects in the elaboration of solutions of this nature(LUCKHAM, 2002; ETZION; NIBLETT; LUCK-

HAM, 2011). The model needs a hierarchy of operations to represent possibilities for composing EPA processing tasks to provide the main desired CEP operations (ETZION; NIBLETT; LUCKHAM, 2011). Since the term "Streaming" has been used as a reference to CEP solutions that leverage streaming technologies for event distribution in scalable scenarios, it has also been considered (as an inclusive criteria) in this research.

University research projects dedicated to developing principles of CEP started in the decade of 1990 (ETZION; NIBLETT; LUCKHAM, 2011). But due to high amount of academic studies related to CEP we opted to restrict the search criteria to more recent studies. Also, we identified a broad study related to CEP modelling, covering progress on CEP since the first CEP embryonic attempts in industry (LUCKHAM, 2002) up to 2011, providing guidance for structuring and organizing artifacts that represent CEP components (ETZION; NIBLETT; LUCKHAM, 2011). A deeper level of details was provided from this work to build solutions based on CEP fundamental principles, and it is referenced by more than 800 academic studies (at the time of this research, according to Scholar). Based on its published date we defined a date range for our search criteria (studies published between 2012 and 2018, *i.e.*, following this study).

A.1.2.2 Questions:

What features, attributes, and properties of EPA operations are relevant to determine a model consistent with the state of the art in CEP processing? To answer this question, we propose to evaluate the publications available in the academic world according to their affinity with EPA operations and features. This allows us to infer concepts and taxonomies applicable to the proposed scenario;

A.1.2.3 Search String, Keywords and Synonyms:

One search string was constructed according to keywords and synonymous from different sections of the papers. In abstract: <(”event processing” OR stream OR streaming) AND (architecture OR model OR organisation organization OR arrangement OR composition OR setup OR formation OR design OR distribution)>. Indicated by the author as keywords: <stream OR event>. Complementary, related to the research focus, from any content section: <”real*time” AND (filtering OR transformation OR processing OR trend

OR tendenc* OR "pattern detect*")>, and on publishing date, <between 2012 and 2018>.

A.1.2.4 Outcome Measure:

Number of relevant articles.

A.1.2.5 Population:

Publications related to CEP architecture, dealing with characteristics involved in the operations of nodes that process events.

A.1.2.6 Application:

Software architects, developers, research community.

A.1.2.7 Experimental Design:

No statistical method will be applied.

A.2 Sources Selection:

Define the sources that will be used as the platform where searches for primary analysis will be executed.

A.2.1 Criteria Definition:

Availability to consult the articles on the web; presence of search engines using key words and databases suggested by experts.

A.2.2 Studies Languages:

English.

A.2.3 Sources Identification:

A.2.3.1 Sources Search Methods:

Search through web search engines.

A.2.3.2 Sources List:

ACM[1], IEEExplore[2], Scopus[3], and Scholar[4]

A.2.3.3 Sources Selection after Evaluation:

All publications that have the potential to collaborate with inference about event processing metadata.

A.3 Studies Types Definition:

Studies related to the research topic with the potential to help elucidate metadata related to event processing in CEP surveys will be selected.

A.4 Procedures for Studies Selection:

After filtering the bases, reading the title, the tags, and abstract section, to discard the articles that are not related to the theme. After reading and selecting the remnants, the main aspects observed will serve as a basis for the research.

[1]https://dl.acm.org/
[2]https://ieeexplore.ieee.org/
[3]https://www.scopus.com
[4]https://scholar.google.com

A.5 Collected Result Data:

A.5.1 Filtering Query:

A.5.1.1 Query Indicators:

- Database: ACM
  - Records: 631 matches
  - Query String: recordAbstract:("Event Processing" stream streaming) AND content.ftsec:("real*time") AND content.ftsec:(filtering transformation processing trend tendenc* "pattern detect*") AND recordAbstract:(architecture model organisation organization arrangement composition setup formation design distribution) AND keywords.author.keyword:(stream event)
  - filter: "publicationYear": "gte":2012, "lte":2018
- Database: IEEE Explore
  - Records: 1206 matches
  - Query String: ("Abstract":stream OR "Abstract":streaming OR "Abstract":"Event Proces*") AND ("real-time" OR "real time") AND (filtering OR transformation OR processing OR trend OR tendenc* OR "pattern detect*") AND ("Abstract":architecture OR "Abstract":model OR "Abstract":organisation OR "Abstract":organization OR "Abstract":arrangement OR "Abstract":composition OR "Abstract":setup OR "Abstract":formation OR "Abstract":design OR "Abstract":distribution) AND ("Index Terms":stream OR "Index Terms":event)
  - filter: 2012 - 2018 (range)
- Database: Scopus
  - Records: 1427 matches
  - Query String: ABS ("Event Processing" OR stream*) AND ALL (filtering OR transformation OR processing OR trend OR tendenc* OR "pattern detect*") AND ABS (architecture OR model OR organisation OR organization OR arrangement OR composition OR setup OR formation OR design OR distribution) AND KEY(stream OR event)

  - filter: Limit to Subject Area: Computer Science;
  - filter: Limit to Keywords: Stream, Data Stream, Data Streams, Stream Processing, Complex Event Processing (options with "stream" or "event");
  - filter: Limit to years 2012 up to 2018

- Database: Scholar
  - Records: 2270 matches
  - Query Criteria (less flexible mechanisms are provided): *All words*: composition event stream model; *Exact phrase*: complex event processing; *At minimum one of these words*: architecture model organisation organization arrangement composition setup formation design distribution
  - filter: Limit to years 2012 up to 2018

A.6 Evaluation

The following approach was used to select the most relevant papers:

- Stage 1: Apply a search criteria to filter and export articles from relevant databases;
- Stage 2: Eliminate duplicates;
- Stage 3: Discard article whose tags are not relevant for this study;
- Stage 4: Read title and summary (abstract) of work and establish which ones show any potential for contributing;
- Stage 5: Consider introduction and conclusion of selected articles and keep the ones that relate to the subject searched;
- Stage 6: Snowball on remaining ones where applicable;

By filtering duplicate papers and discarding the one presenting tags that were not relevant (such as signal processing circuit technologies), the number of matches (from all databases) went down to 2754 unique instances. The majority of those articles was focused in practical issues, specially on achieving better performance metrics on big data scenarios, with no contributions on the design aspect for CEP components. Just by filtering out articles based on title and abstract and inspecting further relevant tags, the number of matching articles became 394.

By selecting relevant papers according to abstract and introduction text, matches went down to 129.

After reading and analysing the remaining papers, aiming at identifying relevant discussions that relate to the research questions, only 28 presented relevant information to subside this research (CUGOLA; MARGARA, 2012; RENNERS; BRUNS; DUNKEL, 2012; ARTIKIS et al., 2012; LINDGREN; PIETRZAK; MÄKITAAVOLA, 2013; MARGARA; SALVANESCHI, 2013; MENDES; BIZARRO; MARQUES, 2013; STOJANOVIC et al., 2014; BAUER; WOLFF, 2014; NECHIFOR et al., 2014; BAUMGÄRTNER et al., 2015; PERERA; SUHOTHAYAN, 2015; KOLCHINSKY; SHARFMAN; SCHUSTER, 2015; KHARE et al., 2015; CARBONE et al., 2015; VELASCO; MOHAMAD; ACKERMANN, 2016; BAPTISTA et al., 2016; RISCH; PETIT; ROUSSEAUX, ; RAY; LEI; RUNDENSTEINER, 2016; FALK; GURBANI, 2017; MAYER; MAYER; ABDO, 2017; DOBBELAERE; ESMAILI, 2017; D'SILVA et al., 2017; ICHINOSE et al., 2017; CANIZO et al., 2017; YADRANJIAGHDAM; YASROBI; TABRIZI, 2017; ZIMMERLE; GAMA, 2018; MANJUNATHA; MOHANASUNDARAM, 2018; DAYARATHNA; PERERA, 2018), complemented by 3 studies obtained by snowballing those (PASCHKE; VINCENT, 2009; MENDES; BIZARRO; MARQUES, 2008; TEYMOURIAN; PASCHKE, 2010). Two of these papers ((PASCHKE; VINCENT, 2009; TEYMOURIAN; PASCHKE, 2010)) also served as a reference for 3 industry publications (MOXEY et al., 2010; ORACLE, 2010; BASS, 2006), summarizing a total of 34 studies as a result of this review.

We observed a gap in the literature - there is no model resulting from this research that represents the basic characteristics of EPA composition. There is a wide variety of published articles related to CEP research, but mostly focused on implementation aspects. This study will enable us to bring together all the knowledge dispersed in publications in an artifact that provides guidance on EPA compositions and also represents the state of the art for CEP processing.

B. Model Components - related attributes and methods

B.1 Event & Event Type

Tables B.1 to B.7 outline attributes and methods related to model components from Section 3.4.1.

Table B.1: Event

Attributes	
Name	**Description**
id	a system-generated unique ID for an Event instance, helpful to diagnose issues and trace individual occurrences.

Table B.2: EventPayload

Attributes	
Name	**Description**
attributeValues	a Set of business attributes included in events. To fetch the appropriate object type for the value, casting can be determined from *getValueType* call in EventAttribute instance (inherited from DataAttribute - see Section 3.4.3).

Table B.3: EventOpenContent

Attributes	
Name	**Description**
annotation	an optional free-text explanation of what happened in this particular event. Event producer or EPA can use it to supply an event instance with explanation for documentation purposes.

Table B.4: EventAttribute

Attributes	
Name	**Description**
derived	besides DataAttribute inherited behavior and data structures (see Section 3.4.3), this specialization indicates whether this is a raw event, originated from an EventProducer, or a derived event introduced by an EPA.

Table B.5: EventHeader

Attributes	
Name	**Description**
occurrence	this attribute indicates the occurrence time attribute with a precision given by the event type's temporal granularity (see *chronon*). It may be actually be in a finer granularity, but, for all processing purposes, it should be rounded to the chronon granularity. It records the time at which the event occurred in the external system. This value is provided by the event producer or EPA.
chronon	denotes the temporal granularity via TemporalUnit instance (see Section 3.4.2), the atom of time related to the event occurrence from a particular application's point of view (second, hour, quarter, season, etc.).
detection	The detection time attribute is a timestamp that records the time at which the event became known to the CEP system (in *chronon*'s temporal granularity).
source	a reference to EventEmitter (see Section 3.5.1), containing a shallow clone of the entity that originated this event. It is sometimes useful for an event processor to know where an event instance came from.
certainty	an estimate of the certainty of this particular event, provided by producers, when applicable. The event certainty attribute has a value between 0 and 1.0 - the latter if it is certain that this event occurred as indicated in its payload, and lower based on the risk that the event did not occur as described.
causers	a list of event ids whose accumulated processing generated the current event, denoting a cause-effect relationship.
type	A reference to EventType object, whose class attributes are further described.
filterAttrs	specifies a list of Strings related to BusinessAttribute names. Optimizations introduced in CEP platforms can leverage this list of relevant attributes, thus reducing the payload, minimizing network latency and optimising throughput.

Table B.6: EventType

Attributes	
Name	**Description**
id	uniquely identifies the type of an event element. Used by the event consumer or EPA that receives the event instance to discover its type.
description	a text explanation regarding relevant event characteristics categorized in each event type.
parent	the self association indicates that an event type may be a composition of another event type.

Table B.7: EventFactory

Methods	
Name	**Description**
createEvent	returns an Event instance, provided the following parameters: • [1] EventType; • [2] Timestamp of occurrence; • [3] TemporalUnit granularity value; • [4] EventEmitter instance who provided this event; • [5] a Map<EventAttribute, Object>, related to EventPayload; • [6] an annotation for EventOpenContent (may be empty/null). Other Event attributes are created by the Factory class, such as detection timestamp.

B.2 Context

Tables B.8 to B.15 outline attributes and methods related to model components from Section 3.4.2.

Table B.8: ContextComponent

Attributes	
Name	**Description**
id	a system-generated unique identity value for an instance designed as per the composite pattern (GAMMA et al., 1995), which deals uniformly with a single or a group of Context elements. The id helps for monitoring threads.
Methods	
Name	**Description**
getWindowId	on the arrival of each Event, dispatches a call to its subclasses to consolidate responses into a String that represents a window that holds this occurrence and the following, correlated ones. The arrived event may be the initial occurrence in this grouping (for which a new String is returned) or a subsequent occurrence, assigned to an existing window id string - see how a windows are represented for partitions in Section 3.6.2.
isPertinent	once we have an active window in place we will need to detect event occurrences that relate to it, as they come. For further information see ContextEPAComposite in Section 3.6.2. It is marked as abstract since it can be implemented differently for single and composite subclasses.
getStartMoment	supports a case where we indicate an initial moment to trigger processing of context-based EPAs.
getEndMoment	this attribute supports a case where we indicate a final moment to abort processing of context-based EPAs.

Table B.9: Context

Methods	
Name	**Description**
getInnerWindowId	a Function triggered by *getWindowId* on the arrival of an Event - retrieves a String that represents a window partition on each each composed Context (*e.g.*, a partition based on initial timestamp and an amount of time for a FixedTemporalContext). Its implementation is delegated to subclasses.
getInnerPertinence	the logic behind *isPertinent*, it provides a Predicate (defined for each pertaining Context components) that verifies if an incoming Event matches a window membership criterion. Besides current event, attributes from window initial event, such as creation timestamp, may apply.

Table B.10: ContextComposite

Attributes	
Name	**Description**
children	a Set with all composed Context instances. As the instance overrides *ContextComponent* methods, this helps ContextComposite processing Context compositions, consolidating information from all Set components.

Table B.11: EventContext

Attributes	
Name	**Description**
acceptCriterion	a Predicate based on a Set of EventTypes, used to determine the context acceptance condition for incoming events. If neither order nor completeness are enforced (other attributes), an occurrence matching any of the EventType instances from this set is accepted. If order is enforced, only events in the specified order of the Set are accepted. If completeness is enforced, evaluation for acceptance is postponed and only combinations of Event instances that fully match EventTypes from *acceptCriterion* are accepted.
orderEnforced	this signals the order of EventTypes matters for accepting Event instances.
completenessEnforced	signals the requirement of a full match of event types from the *acceptCriterion* Set to the ones assigned to Event instances. When not enforced, partial matches are accepted.

Table B.12: SegmentedContext

Attributes	
Name	**Description**
partitionCriterion	a Predicate, based on a Set of event attributes, determines the context acceptance condition. Context segmentation that drives EPA processing can be established based on business data classifications (such as profile from bank customers on risk assessment operations over financial transactions).

Table B.13: TemporalUnit

Enumeration	
Values	**Description**
MINUTE, HOUR, DAY, MONTH, ..	Enumeration types of temporal unit.

Table B.14: TemporalContext

Attributes	
Name	**Description**
unit	a TemporalUnit item. Along with *size*, used to establish the time-frame for which the window lasts (*size* x *unit* of time).
size	amount of units of time above.

Table B.15: FixedTemporalContext

Attributes	
Name	**Description**
initialMark	apart from TemporalContext attributes, this sets a specific moment for the start of a window (e.g. 00:00 hours). All successive occurrences within the timeframe are assigned to the same window, therefore provisioning a reduced number of context-based EPA, as we compare it to TemporalContext, where each occurrence dispatches a window and can join other windows.

B.3 Event Data

Table B.16 outlines attributes and methods from Section 3.4.3.

Table B.16: DataAttribute

Attributes	
Name	**Description**
name	identifier for existing attributes. Enables event content filtering on pre-processing stages (see Section 3.5.2.
value	if present, this holds a value of an attribute (thus the usage of Optional). The value can be of any type, as long as it can be serialized (for streaming).
Methods	
Name	**Description**
getValueType	retrieves the class type of the attribute value, enabling consumers to properly cast the value into appropriate data structures.

B.4 Event Producer

Tables B.17 to B.19 outline attributes and methods related to model components from Section 3.5.1.

Table B.17: EventEmitter

Methods	
Name	**Description**
publishEvents	this interface method decouples the activity of publishing events and is reused by event producers and EPA. Parameters are: Optional<PublishPattern>, for validating the events; the derived or published events; and the target channel (see Section 3.7.1).

Table B.18: PublishPattern

Attributes	
Name	**Description**
types	a possibly empty collection of EventType instances, for validation prior to publishing events. If this is not empty, an Event is published if its EventType matches one of the elements from this Set. Required attributes can be enforced by setting them on instances from this collection.

Table B.19: EventProducer

Attributes	
Name	**Description**
annotation	on top of the inherited methods and attributes, EventProducer has a free formatted field (optional) to enable CEP solutions keep track of information that is specific to Event originators.
Methods	
Name	**Description**
constructor	instantiates EventProducer, sets the Channel instance (see Section 3.7.1) and the PublishPattern (Table B.18).

B.5 Event Consumer

Tables B.20 to B.22 outline attributes and methods related to model components from Section 3.5.2.

Table B.20: EventFetcher

Methods	
Name	**Description**
subscribeEvents	an interface method private implementation for subscribing events from channels, reused by EventConsumer and EPA. Parameters are: SubscriptionPattern (Table B.21), for filtering events and attributes; and the source channel (see Section 3.7.1). As a result we obtain a Stream with the relevant occurrences.

Table B.21: SubscriptionPattern

Attributes	
Name	**Description**
filterByTypes	a BiPredicate accepts or rejects each event (first parameter) according to a collection of distinct EventType instances (second parameter) - can be used to validate the subscribed events from EventFetcher stream.
map	a BiFunction allows reducing event content according to a BiFunction implementation that pulls, from original Event occurrences, only the relevant attributes (selected according to a Set of EventAttributes).
filterByHeader	a BiPredicate accepts or rejects each event (first parameter) according to a template EventHeader instance (second parameter) that can be used to validate the subscribed events (for instance restricting event source).

Table B.22: EventConsumer

Attributes	
Name	**Description**
annotation	on top of the inherited methods and attributes, EventConsumer has a free formatted field to enable CEP solutions to keep track of information that is specific to consumers (the destiny software solutions).
Methods	
Name	**Description**
constructor	instantiates EventConsumer, sets the Channel instance (see Section 3.7.1) and the SubscriptionPattern (Table B.21).

B.6 EPA

Tables B.23 to B.25 outline attributes and methods related to model components from section 3.6.1.

Table B.23: EPAComponent

Attributes	
Name	**Description**
id	a system-generated unique ID, applies for single and composite EPAs. This is helpful for EPA operational purposes (*e.g.*, monitoring threads).
Methods	
Name	**Description**
setChannel	sets the Channel instance (see Section 3.7.1).
setPubPattern	sets the PublishPattern (see Section 3.5.1).
setSubPattern	sets the SubscriptionPattern (see Section 3.5.2).
setProcessor	supplies a StreamConsumer implementation, providing event processing logic, triggered as applicable events (based on filtering criteria) are pulled from the channel.

Table B.24: EPAComposite

Methods	
Name	**Description**
constructor	instantiates EPAComposite, sets the internal Channel instance (see Section 3.7.1).
getInternalChannel	retrieves the internal Channel used by the composition members.

B.7 Stateless EPA

Tables B.26 to B.30 outline attributes and methods related to model components from section 3.6.1.1.

Table B.25: EPA

Attributes	
Name	**Description**
referenceData-GlobalStateId	holds a reference to an instance of GlobalState (ReferenceData-GlobalState specialization, see Section 3.6.1), from which it fetches technical parameters, such as log level, timeout, and event buffer (cached for performance purposes).
Methods	
Name	**Description**
default constructor	instantiates EPAs that are not part of compositions.
parameterized constructor	instantiates EPAs that are part of compositions, which communicate via the inner channel indicated by the parent attribute, an EPAComposite.

Table B.26: FilterAttributes

Attributes	
Name	**Description**
rejectFilter	a Predicate implementation for rejecting events, based on attributes bound to Event internal data structures (payload, header, etc. - see Section 3.4.1), possibly validating them based on external data (from GlobalState).
acceptFilter	similar to the Predicate above, but the implementation is provided with the logic to accept events.

Table B.27: StatelessFilterEPA

Attributes	
Name	**Description**
attrs	an instance of FilterAttributes (Table B.26), allows more elaborated filtering operations through the usage of Predicate functions.

Table B.28: TranslateEPA

Attributes	
Name	**Description**
map	a conversion Function implementation, allows simple Event transformations based on existing attributes.

Table B.29: ReferenceDataParameter

Attributes	
Name	**Description**
refDataGlobal-StateId	identifies the GlobalState used to pull information from an external data source.
searchedData	the identifier for the data attribute required to pull the external information - defined as a ReferenceDataAttribute (see Sections 3.4.3 and 3.8).
newAttribute-Mapper	a Function implementation to transform the attribute retrieved from external source into the new event attribute (EventDataAttribute, from Sections 3.4.3 and 3.8).

Table B.30: EnrichEPA

Attributes	
Name	**Description**
map	a BiFunction implementation drives complex Event transformations based on both existing attributes and external Data (fetched via GlobalState instances). New Event attributes may be incorporated into existing Event data structure, while existing data may or not be preserved, according to this logic.

B.8 Stateful EPA

Tables B.31 to B.37 outline attributes and methods related to model components from section .

Table B.31: StatefulEPA

Attributes	
Name	**Description**
buffer	a UnaryOperator that buffers each Event while traversing the stream of Events. This operation is pipelined in front of the main StreamConsumer operation, returning the same Event after buffering so that it remains an argument to StreamConsumer.

Table B.32: StatefulFilterOperator

Enumeration	
Values	**Description**
FIRST_N, LAST_N, RANDOM_N, TOP_N, BOTTOM_N	Enumeration types that drive sampling processes.

Table B.33: StatefulFilterEPA

Attributes	
Name	**Description**
attrs	an instance of FilterAttributes (Table B.26).
operator	An instance of StatefulFilterOperator (Table B.32) - determines a sampling strategy. Depending on the strategy, the EPA is required to be processed according to a Context. Except for FIRST_N case, filtering process is only triggered once all events, under a context partition, are received (*i.e.*, a window terminate condition is met - see Section 3.6.2 for details).
countArg	sampling size - "N", in StatefulFilterOperator types.

Table B.34: AggregateOperation

Attributes	
Name	**Description**
reduceOpIdentity	an associative operation identity value. Acts as a starting value for *reduceOperator* processing (*e.g.* 0 for sums, 1 for product, etc.)
reduceOperator	a BinaryOperator to infer an aggregated value of type T. Computed incrementally while traversing a stream of events, pulling a T value for each (through *mapOperator*), and considering the previously calculated value (partial result) of type T.
useThreshold	indicates if a threshold should be used (*e.g.*, if the calculated sum reaches an unsafe boundary for the business).
threshold-Comparison	a BiPredicate confirms the reduced inferred value reaches a threshold (*e.g.*, calculated value ≥ `threshold`).
mapOperator	a Function returns a value of type T from an Event instance, based on one attribute or a combination of its attributes, and possibly considering data external to the event instance.

Table B.35: AggregateEPA

Attributes	
Name	**Description**
aggregation	supplies the AggregateOperation (Table B.34) instance needed to infer an object of type T, according to a reduction operation. The generic T parameter varies from a simple structure (*e.g.*, a sum value) to a complex one (*e.g.* centroid nodes, see Section 3.6.1.4).
mapOutput-Occurrence	a Function implementation that derives a Event based on the output (T) from reduction operation.

Table B.36: UnmatchedPolicy

Enumeration	
Name	**Description**
FAIL, FORWARD	Indicates the strategy to be adopted when an event coming from one stream does not match any event from the other stream - a scenario applicable for ComposeEPA (Table B.37) processing. FAIL means an abnormal terminate condition is reached, whereas FORWARD means this occurrence is buffered and processing continues.

Table B.37: ComposeEPA

Attributes	
Name	**Description**
acceptanceFilter	specifies a matching condition, provided two events, one from the usual stream (via an inner or global channel) and the other from the stream specified via *subscribeFromNewStream*. This BiPredicate returns true if they match, and it is verified every time an event arrives, coming from either one of the streams. The matching condition is checked between this event and each event from the other stream (stored in a buffer). This is usually processed under finite partitions, since a polinomial order of growth is expected when traversing buffers for occurrences.
map	BiFunction logic to produce a new Event from two matching Events (each from a different stream). Triggered every time acceptanceFilter returns true.
streamsUnmatchedPolicy	UnmatchedPolicy reference to be verified whenever a new event from one stream does not match any of the accumulated events from the other stream.
Methods	
Name	**Description**
subscribeFromNewStream	indicates the Channel and SubscriptionPattern to fetch events from a separate stream, whose occurrences may present relevance when compared to the ones coming from EPA usual Channel. For instance, an EPA may subscribe to "departure" events and check their crossed correlation with "arrival" events.

B.9 Pattern Detect EPA

Tables B.38 to B.49 outline attributes and methods related to model components from section 3.6.1.3.

Table B.38: EvaluationMode

Enumeration

Values	**Description**
IMMEDIATE, DEFERRED	Indicates that the evaluation should be performed at occurrence time, or postponed to a Context terminate condition (see Section 3.6.2).

Table B.39: OrderMode

Enumeration

Name	**Description**
STREAM_POS, USER_DEFINED, DETECT_TIME, OCCURR_TIME	Indicates the criterion to be considered for the order of events. Among the options we have: the stream position (from streaming platform); a user-defined attribute (being the logic provided via *orderCompare*); the actual event occurrence time; and the time an event becomes known to our CEP solution.

Table B.40: ExcessMode

Enumeration

Name	**Description**
LAST, FIRST, EVERY	Indicates the criterion to be considered if the amount of events exceed the established *excessLimit* (see Table B.42). We may disregard the limit - EVERY; or consider only the first or last occurrences. This usage implies EPA is processed according to a TemporalContext, and EvaluationMode is DEFERRED.

Table B.41: ReuseMode

Enumeration	
Name	**Description**
CONSUME, REUSE, BOUNDED	Types that act as an expiration condition. We may consider an occurrence only one time, then remove from any future buffering mechanism - CONSUME; or consider *N* number of times, meaning buffering is maintained until a *reuseLimit* (see Table B.42) is reached - BOUNDED; or even REUSE, where no expiration is in place.

Table B.42: MatchingPolicy

Attributes	
Name	**Description**
excess	an ExcessMode (Table B.40) instance.
excessLimit	maximum amount of Event instances, to be considered along with *excess* attribute.
evaluation	an EvaluationMode (Table B.38) instance.
reuse	a ReuseMode (Table B.41) instance.
reuseLimit	maximum number of times an event is supposed to be considered, to be considered along with the *reuse* attribute.
order	an OrderMode (Table B.39) instance.
orderCompare	the lambda expression to be considered along with the *order* attribute (as long as it is established as USER_DEFINED).

Table B.43: PatternDetectEPA

Attributes	
Name	**Description**
matchingPolicy	an instance of MatchingPolicy (Table B.42), set in order to clarify the semantics of pattern matching operations. Drives EPA agents to act on the occurrence of multiple events.

Table B.44: PatternSequence

Enumeration	
Name	**Description**
UNORDERED, TEMPORAL	Indicates if searched EventType items should be considered in the same order established in *typeSet* (Table B.46).

Table B.45: PatternModal

Enumeration	
Name	**Description**
ALL, SOME, NONE	Indicates the processing logic in regard to the *typeSet* (Table B.46) attribute: ALL, if only a full match of EventType items satisfies the criteria; SOME when the criteria is satisfied by at least one match; or NONE, where the criteria is accepted if none of the subscribed events apply to any element from this set.

Table B.46: BasicPatternDetectEPA

Attributes	
Name	**Description**
typeSet	a LinkedHashSet of EventType items is defined in this attribute. This EPA is always processed according to a context, with its EvaluationMode marked as DEFERRED. At the closure of the context segment, the subscribed event instances are evaluated according to *sequence* and *modal* attributes.
modal	an instance of PatternModal (Table B.45). Considering EventType from all incoming events within a segmented TemporalContext, and their relationship to items from *typeSet*, this attribute indicates if a matching should be satisfied by all *typeSet* participants, or by some (or even none) of its members.
sequence	an instance of PatternSequence (Table B.44), relates to the order EventType items (from *typeSet*) should be considered.

Table B.47: ConditionalPatternDetectEPA

Attributes	
Name	**Description**
condition	this Predicate provides an acceptance criterion based on Event data structure. It is evaluated for each occurrence and possibly reduces the population of matched occurrences.

Table B.48: TrendType

Enumeration	
Name	**Description**
INCREASING, DECREASING, STABLE,MIXED, NONINCREASING, NONDECREASING	indicate the desired tendency, according to the relationship between sequential events (provided in *trendCheck*, from Table B.49) and how this is expected to vary through time.

Table B.49: TrendPatternDetectEPA

Attributes

Name	**Description**
type	an instance of TrendType, establishes the pursued tendency.
trendCheck	a BiPredicate function correlates pairs of sequential Event instances via *trendCheck*, and also maintains a set historical measurement values.
threshold	an optional element can be provided so that this processing reacts (deriving new events) only to changes quantified with a value greater than this attribute (the value comes from *trendCheck* logic). Larger thresholds can also be used to pick outlier events.

B.10 Clustering EPA

Table B.50 outlines attributes and methods related to the main model component from section 3.6.1.4.

Table B.50: ClusteringEPA

Attributes	
Name	**Description**
k	contains number of centroids.
centroids	an array with *k* entries holding each centroid, an element of generic type C.
distanceInference	BiFunction that derives values that represent distances between the occurrence (first parameter) and each Centroid, of type C, from *centroids* (second parameter). This implementation outputs an array with those distances with Double precision, where each array position in the return type correlates to the centroid position from the second parameter.
regenerate-Centroids	Function that derives a new centroid (according to median, means, medoid or any algorithm that best represents the goal) from a list of the Events currently assigned to one cluster.
trainingPer-centage	applies within a Context - this optional attribute indicates the percentage of data used for training, picking it from initial events within a context segment.
predictionPer-centage	applies within a Context - this optional attribute indicates the percentage of data used for prediction, picking from final events within a context segment.
timeToLive	time to live - if provided, the amount of time an Event is considered relevant since its occurrence. This attributed is used in conjunction with *ttlTimeUnit*.
ttlTimeUnit	the temporal unit associated with *timeToLive* amount.
clusterElements	List of relevant Event instances considered for clustering
clusterCentroids	Map that relates each relevant Event instance from *clusterElements* to the centroid which presents minimal distance, as seen in *distanceInference*.
maxIterations	an optional attribute to limit the run-time of the clustering algorithm by establishing the maximum number of iterations.
Methods	
Name	**Description**
initCentroids	initializes *centroids*, can be performed via reflection Array *newInstance* call. This requires a strong typing, thus the Class parameter. The number of clusters (array size) is the second parameter.

B.11 Context Partitioner

Tables B.51 to B.53 outline attributes and methods related to model components from Section 3.6.2.

Table B.51: Window

Attributes	
Name	**Description**
id	a system-generated unique ID for an individual Window instance, which is helpful for ContextPartitioner (Table B.53) to identify instances that have already been provisioned.
initialEvent	the first of a series of events that belong to a window. This occurrence has a particular relevance, since the window id may have been originally inferred based on its attributes (such as occurrence timestamp and customer profile).
ctx	the ContextComponent instance (see Section 3.4.2) associated with the Window. This instance assists on determining the pertinence of further event occurrences to this Window, via a call to its *isPertinent* method.
Methods	
Name	**Description**
constructor	instantiates a Window according to an initial Event and an existing ContextComponent, provided by ContextPartitioner.

Table B.52: ContextEPAComposite

Attributes	
Name	**Description**
window	holds a Window instance, provided by ContextPartitioner (Table B.53), that relates to this instance.

Methods	
Name	**Description**
constructor	sets the Channel instance (see Section 3.7.1) and the Window instance, provided by ContextPartitioner (Table B.53).
isTerminateCondition	pulls the ContextComponent associated to the *window* instance and calls *getEndMoment* to verify if it should cease activities.
isEventApplicable	dispatched at the arrival of subscribed Event instances, this method pulls both ContextComponent and initialEvent associated to the *window* instance and calls *isPertinent* to verify if the incoming Event applies to this Window segment.

Table B.53: ContextPartitioner

Attributes	
Name	**Description**
windowIdSet	holds a collection of ids from all distinct Window instances provisioned by ContextPartitioner. As this instance pulls events, each new occurrence is evaluated according to *isNewWindowRequired* - which indicates whether a new Window (Table B.51) should be provisioned (therefore updating this collection) or if this event applies only to an existing Window.
Methods	
Name	**Description**
constructor	instantiates ContextPartitioner, sets the ContextComponent instance (see Section 3.4.2), the global Channel (see Section 3.7.1) and the ContextEPAComposite instance (Table B.53), which serves as a template for the creation of EPAs from *cloneInceptiveEPA*.
isNewWindowRequired	each new event is evaluated according to this method - which indicates whether a new Window should be provisioned, or if this Event applies to an existing Window. It does do by using the subscribed Event as *initial* parameter to *getWindowId*, from ContextComponent, which retrieves a Window Id that may already exist in *windowIdSet*. If that is the case, the event will be handled by the existing ContextEPAComposite associated to the existing window. Otherwise, it provisions a new Window and its related ContextEPAComposite instance, via *cloneInceptiveEPA*.
cloneInceptiveEPA	dispatched every time a new Window instance is required. It reuses *template* parameter from the constructor to provision a new ContextEPAComposite, which starts subscribing and handling the processing of all events that belong to this new Window - *i.e.*, an agent is provisioned to handle occurrences from an event stream segment, according to its Context semantics, for as long as the Context is valid.

B.12 Offline Event Loader

Table B.54 outlines attributes and methods related to Section 3.6.3.

Table B.54: OfflineEventLoader

Methods	
Name	**Description**
scheduleExpression	an expression (*e.g.* with standard cron syntax) used to schedule batch loading tasks to run periodically at a specific date and time.
constructor	in addition to EventProducer constructor parameters (see Section 3.5.1), this requires an identity value for EventGlobalState, which allows it to pull batches of events from a repository (see Section 3.8). Further in the process, it can publish events based on Channel and PublishPattern parameters (see Section 3.7.1).

B.13 Channel

Tables B.55 to B.57 outline attributes and methods related to model components from Section 3.7.1.

Table B.55: EncodedEvent

Attributes	
Name	**Description**
encodedContent	holds a serialized Event, resulting from *encodeEvent* Function call, from a Channel.

Table B.56: Channel

Attributes	
Name	**Description**
encodeEvent	a Function transforms Event instances into a proper format for optimized sizing and performance (EncodedEvent, from Table B.55);
decodeEvent	a Function rebuilds Event instances out of EncodedEvent;
streamPlatform-ConnectivitySettings	connectivity attributes related to the chosen streaming platform.

Methods	
Name	**Description**
publish	establishes a connection with stream platform and invokes its available mechanisms for publishing events as messages. Through the usage of *encodeEventFunction*, transforms Event into EncodedEvent instances prior to pushing them into the platform. A batch of EncodedEvent elements (instead of a single instance) can be pushed for better latency.
consume	establishes a connection with stream platform and invokes its available mechanisms for subscribing events according to criteria (for instance, via message topics). At this moment, advanced filtering may be dispatched on stream platforms (if available) while fetching events - *i.e.*, based on the incoming *SubscriptionPattern* parameter (See Table B.21) we may restrict pulling to events (and event attributes) that match the specified acceptance criteria. Through the usage of *decodeEventFunction*, this method transforms EncodedEvent into Event instances as they are consumed from the platform - which is invoked as part of stream pipeline processing. If stream platforms do not provide advanced filtering mechanisms, this component can filter data after decoding is performed. In any case, as a result, the channel clients will be provided a Stream of Event instances, which may be an infinite Stream, populated as applicable events arise. This implies in establishing connectivity for continually fetching subscribed messages, until a termination criteria is met.

Table B.57: ChannelBroker

Attributes	
Name	**Description**
globalChannel	the global channel instance.
innerChannels	a map containing all inner channels, associated to an EPA composition. Its key is the composite EPA id parameter (coming from *getCompositeEPAChannel* call).
Methods	
Name	**Description**
getGlobalChannel	retrieves the instance for *globalChannel* (creates if it does not exist yet).
getComposite-EPAChannel	checks *innerChannels* map for the existence of a channel associated to the EPA id (method parameter) and returns it if so. Otherwise, creates a new Channel instance, segregated from the Global Channel (an inner channel, for all EPAs within the indicated composite EPA), and add it to the map.

B.14 Global State

Tables B.59 to B.65 outline attributes and methods related to model components from Section 3.8.

Table B.58: AttributeContainerType

Enumeration	
Values	**Description**
DOCUMENT, COLLECTION, VIEW, SCHEMA, TABLE, SET, ..	Enumeration types of container data structures.

Table B.59: GlobalStateDataStore

Attributes	
Name	**Description**
dataSourceSettings	properties containing settings such as connectivity and caching for a data store. Based on that, an internal query provider instance is created, whose implementation is left out of scope of this model.
Methods	
Name	**Description**
query	pulls information (of generic type *R*) from the data store by interacting with to the internal query provider instance, based on *dataSourceSettings*.
update	inputs information into the data store by interacting with the internal query provider instance, based on *dataSourceSettings*.

Table B.60: AttributeContainer

Attributes	
Name	**Description**
name	identifies the data structure element (VIEW, TABLE, DOCUMENT, etc) by its name.
type	an instance of AttributeContainerType (Table B.58).
parent	indicates a parenthood relationship to another instance of AttributeContainer (*e.g.* a DOCUMENT element relates to a parent element of type COLLECTION, as a TABLE relates to a SCHEMA).

Table B.61: ReferenceDataAttribute

Attributes	
Name	**Description**
container	in addition to DataAttributes inherited data structure and behavior (see Section 3.4.3), this attribute provides the container entity, within its data source, that encompasses this element - an instance of AttributeContainer (Table B.61).

Table B.62: GlobalStateProvider

Attributes	
Name	**Description**
globalStateRefs	a map that maintains a reference for previously created ReferenceDataGlobalState (Table B.64) and EventGlobalState (Table B.65) instances.
Methods	
Name	**Description**
getReferenceDataGlobalState	pulls the ReferenceDataGlobalState instance (if applicable) from *globalStateRefs*, based on provided id, validates connectivity to this data source and retrieves an Optional element wrapping it. Clients should use it check connectivity from time to time.
getEventGlobalState	pulls the EventGlobalState instance (if applicable) from *globalStateRefs*, based on provided id, validates connectivity to this data source and retrieves an Optional element wrapping it. Clients should use it check connectivity from time to time.
createEventGlobalState	instantiates EventGlobalState, provided the following parameters: • [1] GlobalStateDataStore instance (Table B.59) with connectivity settings; • [2] The representation for the database row or tuple; • [3] queryMap, the Function implementation to pull an Event from a database row; • [4] updateMap, the Function implementation to pull an update statement String from an Event. Adds the new instance to *globalStateRefs* prior to returning it.
createRefDataGlobalState	Returns an ReferenceDataGlobalState instance, provided the following parameters: • [1] GlobalStateDataStore instance with connectivity settings; • [2] The representation for the database row or tuple; • [3] queryMap, the Function implementation to pull a ReferenceDataAttribute from a database row; • [4] updateMap, the Function implementation to pull an update statement String from a ReferenceDataAttribute. Adds the new instance to *globalStateRefs* prior to returning it.

Table B.63: GlobalState

Attributes	
Name	**Description**
id	a system-generated unique ID for an individual GlobalState instance, which is helpful for providing EPA proper state elements.
dataStorage	GlobalStateDataStore instance that intermediates data store accesses.
Methods	
Name	**Description**
query [+]	pulls information from the data store according to an expression String. This invokes the protected *query* implementation, which maps the query data structure outcome into the proper model object.
query [#]	used by GlobalState subclasses, that pulls information from the data store (via dataStorage *query*) and maps each resulting record into an Event or ReferenceDataAttribute instance, based on the *queryMapper* Function provided by subclasses.
persist [+]	inputs information into the data store according to an expression String. This invokes the protected *persist* implementation, which maps the proper model structure into an update statement.
persist [#]	used by GlobalState subclasses, that inputs information into the data store (via dataStorage *update*) and maps each input Event or ReferenceDataAttribute instance into a statement, based on *updateMapper* Function provided by subclasses.

[+] Denotes a *public* method.

[#] Denotes a *protected* method.

Table B.64: ReferenceDataGlobalState

Attributes	
Name	**Description**
queryMapper	used in conjunction with query operations, maps each record retrieved from the database into a ReferenceDataAttribute (Table B.61).
updateMapper	used in conjunction with persistence operations, maps each ReferenceDataAttribute (to be persisted) into an update statement.
Methods	
Name	**Description**
getQueryByAttributeExpression	pulls a query statement based on the ReferenceDataAttribute provided as a parameter, serving as a template. For instance, the ReferenceDataAttribute *name* and *container* can be provided (so that a query would bring a single instance), or just the *container* (a query would bring all attributes within the container).

Table B.65: EventGlobalState

Attributes	
Name	**Description**
queryMapper	to be used in conjunction with query operations, maps each record retrieved from the database into an Event (described in Section 3.4.1).
updateMapper	to be used in conjunction with persistence operations, maps each Event to be persisted into an update statement.
Methods	
Name	**Description**
getQueryByDateExpression	pulls a query statement based on a date range (according to an initial and final date - by convention we consider those as inclusive dates).
getQueryByEventExpression	pulls a query statement based on the Event provided as a parameter, serving as a template. For instance, the Event *id* can be provided, so that a query would bring a single instance; or just the EventHeader *source* attribute, and a query would bring all Events produced by the specified EventEmmitter.

C. Coupling Metrics Evaluation

Here we provide the rational used for the elaboration of Table 4.9.

As previously indicated, for each class we identify dependencies from the CEP model in Section 3.3, according to the definitions of CBO, NDO and RECBO, from Section 4.1.2. For RECBO, refer to instances depicted in Figures 4.9 and 4.10. This analysis did not include: dependencies to Java API objects (such as List), Enumeration instances and template parameters (which relate to API objects defined dynamically). References indicated for the metrics obtained on Table C.1 are described on Table C.2.

Table C.1: Coupling-Related Measurements

Class	**Object(s)**	**CBO**	**NDO**	**RECBO**
EventProducer	*LSTN_G1.1, LSTN_G1.2, P_G2.3, P_G2.4*	5 [i]	0 [ii]	0 [ii]
OfflineEventLoader	*OL_G4.7, OL_G4.8, OL_G3*	7[iii]	0 [ii]	0~[ii]
EventGlobalState	*GS_G3_CAS, GS_G4.7, GS_G4.8*	3[iv]	2 [v]	1 [v]
EventConsumer	*CONSUMER_G1234*	5 [i]	0 [ii]	0 [ii]
ReferenceDataGlobalState	*GS_G4.CORP_ID, GS_CL1234*	5[vi]	2 [v]	1 [v]
EnrichEPA	*ENRICH_G4*	9[vii]	1 [viii]	0 [ix]
StatefulFilterEPA	*FILTER_G1, FILTER_G234*	7[x]	1 [viii]	0 [ix]
ClusteringEPA	*CLUSTER_G1234*	8[xi]	1 [viii]	0 [ix]
ContextEPAComposite	*COMPOS_G1.8, COMPOS_G234.6, COMPOS_G234.2, COMPOS_G1234.7, COMPOS_G1234.8*	9 [xii]	1 [xiii]	0 [ix]
AggregateEPA	*AGGR_G1_VI, AGGR_G234_I, AGGR_G234_VI, AGGR_G1234_II, AGGR_G1234_III*	8 [xi]	1 [xiv]	0 [ix]
Channel	*Global, ContextII, ContextIII, ContextIV, ContextVI*	5[xv]	$\propto c$[xvi]	$\propto \alpha$[xvii]

Table C.2: Coupling-Related Measurements (Notes)

[i] Dependencies: Channel, ChannelBroker, EventEmitter, EventFactory, Event and Event Type, and PublishPattern (for producers) and SubscrptionPattern (for consumers).

[ii] No other class from the model indicates dependency for this class.

[iii] Dependencies: Channel, ChannelBroker, PublishPattern, EventEmitter, EventType, EventGlobalState, GlobalStateProvider.

[iv] Dependencies: Event, GlobalStateDataStore, EventAttribute.

[v] Either GlobalStateProvider factory or an offline event loader references this instance at run-time (both are valid when counting classes.)

[vi] Dependencies: Event, GlobalStateDataStore, ReferenceDataAttribute, AttributeContainer, AttributeContainerType.

[vii] Dependencies: Event, EventFactory, ReferenceDataAttribute, EventAttribute, ReferenceDataGlobalState, ReferenceDataParameter, GlobalStateProvider, PublishPattern, SubscriptionPattern.

[viii] References this class, at creation time: EPAComposite.

[ix] After instantiation, those elements are not referenced by other model components (they are only monitored via container orchestration platform).

[x] Dependencies: Event, EventFactory, ReferenceDataGlobalState, GlobalStateProvider, FilterAttributes, PublishPattern, SubscriptionPattern.

[xi] Dependencies: Event, EventFactory, ReferenceDataGlobalState, GlobalStateProvider, MatchingPolicy, PublishPattern, SubscriptionPattern, AggregateOperation.

[xii] Dependencies: Event, EventFactory, ReferenceDataGlobalState, GlobalStateProvider, MatchingPolicy, PublishPattern, SubscriptionPattern, Window, ContextComponent.

[xiii] References this class, at creation time: ContextPartitioner.

[xiv] References this class, at creation time: ContextEPAComposite.

[xv] Dependencies: Event, EncodedEvent, SubscriptionPattern, and inspect EventType and EventAttribute (for advanced filtering)

[xvi] Class static references to this class: this scales according to the number of chosen EPA Specializations (e): our model (Section 3.3) shows currently 20 classes bound to Channel, where 12 of them are EPA specializations: ChannelBroker, EventProducer, OfflineEventLoader, EventConsumer, ClusteringEPA, EnrichEPA, StatelessFilterEPA, StatefulFilterEPA, ContextPartitioner, EventEmitter, EPAComposite, EventFetcher, ContextEPAComposite, SplitEPA, TranslateEPA, AggregateEPA, ComposeEPA, BasicPatternDetectEPA, ConditionalDetectEPA, TrendPatternDetectEPA.

[xvii] The amount of dynamic references to this class is proportional to the number of agents (a): as depicted in Figures 4.9 and 4.10, it scales to the 21 active agents, plus one ChannelBroker. In reality, we reached an approximate number of 100 active agents