Mātārere: A Layered Framework for Specifying Event-Based Systems

Agnès Voisard † and Holger Ziekow*

TR-10-002

February 2010

Abstract

Event-based systems (EBS) have received increasing attention in the past decade from various communities. Central to these systems is the notion of event, which is often considered as ”a happening of interest”. An EBS encompasses a large range of functionalities on various technological levels (e.g., language, execution, or communication). Many approaches exist to handle events and the solutions are not always either defined on a clear level or well understood. This is mainly due to the difference between the communities that tackle the issues, which also leads to some confusion in the terminology. The notion of events itself is often not clearly defined and varies from one approach to the other.

This paper presents Mātārere, a layer decomposition of a generic EBS into levels of abstraction. The goal of the solution proposed here is threefold: (1) provide unifying terminology, (2) understand better the various existing approaches which take place on different levels, and (3) offer some support to build an EBS using this uniform framework. A strength of our approach lies in the consideration of recursion - in the sense that all layers may repeat themselves on different system levels - allowing a better modeling of complex event-based systems.

†Fraunhofer Institute for Software and Systems Engineering (ISST) and Freie Universität, Steinplatz 2, 10623 Berlin, Germany
* Institute of Information Systems, Humboldt-Universität zu Berlin, Spandauer Straße 1, 10178 Berlin, Germany

This work was partially funded by the Deutscher Akademischer Austausch Diesnst (DAAD) through a postdoctoral fellowship.

Table of Contents

1 Introduction

Event-based Systems (EBS) now serve as a support to a wide spectrum of applications, which range from supply chain management to luggage tracking (see [38] for a survey of applications in that domain). Within these applications, each event can have a different nature and be for instance an RFID read, a new sensor value, or an increasing stock value. Basically, the kernel of such a system reacts to incoming events by triggering an action, often by sending a notification to the user but not necessarily: For instance, in a sensor-actuator system an actuator might retract a blind if it is snowing, or in a manufacturing application a machine might start automatically if it is fed with new production material.

A key paradigm behind EBS is *Complex Event Processing* (CEP) [42], originally developed to solve problems involving a large number of real-time events. Generally, these systems now provide the basis for event capturing, communication, query definition, and different types of processing, among others. The challenge is to orchestrate technologies for such different tasks when building an event-based system. However, today no holistic standard definitions exist for event-based systems and their building blocks. Moreover, many existing approaches concentrate on only some parts of event-based systems. While some systems focus on particular functionalities such as user interaction, others focus on event capturing such as sensor reading or http requests.

In the past few years, the field has received increasing interest from researchers with various background - and not only active databases - as illustrated by the growing number of forums (Cf. for instance the 2007 Dagstuhl workshop on the issue [21], the web-based forum on complex event processing [2], the ACM International Conference on Distributed Event-Based Systems series [1]), and the creation of an interest group named the "Event Processing technical Society" [26], together with its associated symposium). A number of books has also been published on the issue (see for instance [42, 48, 15]).

Many EBS exist that encompass all the basic functionalities that such a system should offer. It would be impossible to cite all of them. Moreover, some systems emphasize particular issues (e.g., languages or performance). In this paper, we make in particular a reference to the following approaches and systems: The HiPac project [23], SAMOS [31], SNOOP [17], READY [35], Siena [13], Cayuga [24], NiagaraCQ [22], TelegraphCQ [19], STREAM [47], TinyDB [46], Fjord [45], and SASE [37].

Looking at the literature, one can see that the field has been of interest to many communities and that some problems have been tackled from different view points. This results in many terminologies that sometimes induce some confusion. Besides, given the amount of existing EBS and their richness in terms of potential functionalities, it is our belief that a systematic description of these systems would help to explain their focus and solutions. This is the subject of this paper. We aim at defining a common framework that:

- provides precise yet simple terminology,
- helps to understand functionalities of existing systems and their possible similarity/redundancy, and

– serves as a reference to build event-based systems.

The layered-approach that we follow here is a natural approach to capture the overall picture of an event-based system. In general, decomposition into levels of abstraction is a key to an efficient architecture. The approach presented here has been successfully adopted in areas such as networking and communication as part of the Open Systems Interconnection Initiative (OSI) [72] and more recently in particular software or large system design, e.g., as we did in [65]. In addition, it allows the consideration of recursion at all levels. The principle is to decompose a generic EBS into levels of abstraction, from a high level - the application level - down to low levels such as sensor interfaces. In our work we present concepts for five layers, namely, from top to bottom: (1) an application layer that covers applications on top of event-based systems, (2) a language layer to formulate high level event queries, (3) an execution layer for running event queries, (4) a communication layer for exchanging events between system components, and (5) a capturing layer for capturing event data from sources such as sensors. At each level we consider the data that is handled together with the operations on them and the key actions that are taken. We thereby refine event-driven architectures to the level of concrete operations and technologies for implementation.

A roughly similar approach was already proposed in [43]. The authors suggest to move away from technical details using abstraction hierarchies together with multilevel viewing. This approach is rather process oriented and does not have the genericity offered by Mātārere. [27] describes the kernel of simple event-based systems in a rather theoretical manner and uses it as a basis to build a generic EBS. More recent, the work of [53] could be seen as a step in the direction of our approach; however, it does not consider distribution and it concentrates on the plain layers (i.e., without recursion) of event-based systems. Moreover, our work is more system oriented as it concentrates on the heart of an EBS. In addition, our framework allows us to present a clear terminology in that domain, which, to the best of our knowledge, has not been proposed so far.

This paper is organized as follows. Section 2 illustrates our approach through a simple example and gives background information, including a precise terminology. Section 3 presents the general layered framework Mātārere. Section 4 discusses the different layers. Section 5 shows how to orchestrate the whole system by mapping layers. Finally, Section 6 presents our conclusions. An appendix summarizes the different layers and their associated characteristics.

2 Background

This section gives the necessary background to understand our approach in decomposing recursively an overall event-based system into levels of abstraction. We start (Section 2.1) with an example that illustrates our general discourse. We then give basic terminology in the EBS domain (Section 2.2) and explain the main mechanisms of event detection and alerting (Section 2.3).

2.1 Reference Example

In this section we illustrate the scope of the addressed systems and aim at providing some intuition about relevant technologies along a reference example. Note that this example is only a placeholder for other applications that use the paradigms of EBS and should not be understood as a limit for the scope of Mātārere.

We take manufacturing as application domain. Manufacturers use technologies of EBS to monitor and control operations on the shop floor. In modern factories, machine sensors, RFID readers, and worker terminals at production stations provide a rich set of event data about the production. EBS prepare these input events for use in higher level application, e.g., alerting systems. For our example we choose an application that notifies the shift supervisor if a machine temperature leaves a specified range. The system sends a text message to the supervisor's cell phone to notify about such problems, allowing timely reactions.

Two types of input are required for generating the notification: (1) a knowledge base to create human-readable text messages and (2) real time information about current temperatures. The knowledge base is specific to the alerting application. However, the real time information about temperatures can be acquired with generic technologies of EBS. The alerting application therefore registers standing complex event queries that define relevant temperature changes. For example, the application can query if temperature at any machine leaves a certain range. Such a query may be phrased using high level language constructs of the EBS. A CQL query for our example may look as follows, assuming an upper bound of 40 and a lower bound of 15:

```
Select machineNo
From SHOPFLOORMACHINES [Range 5 Minutes]
Group By machineNo
Having avg(temperature) > 40 OR avg(temperature) < 15
```

The EBS is logically located between the alerting application and the event sources on the shop floor (see Figure 1). It translates the given query for quality problems into executable operators and allocates processing resources, e.g., on a control server in the plant. The EBS also links in the events from the relevant data sources, in this case machine sensors that detect waste. Subsequently, the system collects all waste-related events in real time and checks them against the query condition. If the condition matches, the alerting system receives a corresponding message and generates a notification for the shift supervisor. All communication is *push based* (in this case, the information is sent to the subscriber, i.e., the supervisor) and all processing steps are conducted instantly. This enables early detection of quality problems and timely reaction[1].

[1] Note that the action is taken outside the EBS. The EBS simply notifies and thereby triggers the actuator

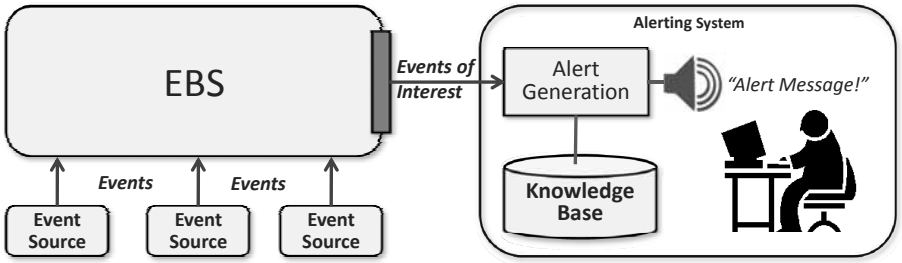

Fig. 1. Alerting application

2.2 Terminology

This section aims at clarifying the terminology and sets the vocabulary used in this paper. Many concepts exist and even the term "event" is understood differently in different communities, although the event glossary of the EPTS community [44] is a notable effort to homogenize the concepts to be considered (note, however, that the different types of events are not detailed in that glossary). The semantics of operators to be applied to events also varies from one approach to the other. In the following, we aim at providing simple - yet powerful - definitions.

Informal presentation of events The definition of events and event processing differ throughout the literature. Before providing definitions let us highlight some aspects of events and event processing that contribute to an intuitive understanding of relevant concepts.

An event is usually understood as "a happening of interest" [42] or "a significant change in the state of the universe" [20]. An RFID read, an increased temperature value on a thermometer, a constant temperature in an area A for one hour, fraud in using an ATM machine, order cancelation, or a storm coming into an area are examples of events. These various, quite different examples illustrate in particular the following points:

1. **Application-dependence**. Event specifications (in the sense that one can specify events of interest) are application dependent. Even with the same phenomena occurring, what can be "of interest" may differ from one application to the other. For instance, when measuring the temperature, the relevant events may be when the temperature increases or when it stays constant for one hour. In general, an event is of interest if it is required when processing a registered event query.
2. **Different abstraction levels**. Event-based systems may consider events at many levels of abstraction, from the application level (e.g., an environmental application considering storms) down to the sensor level (e.g., a new temperature value on a particular thermometer).

3. **Two event capturing modes**. Events can occur in two distinct ways: (1) an event can occur as a report on some values read periodically (e.g., an RFID read) and (2) events can occur based on changes, typically the state transition of an entity of interest at a certain point in time (e.g., an increased temperature value). The first category of events encompasses *status* events while the second one considers *changed-based* events.
4. **Context-dependence of event relevance**. The relevance of an event w.r.t. a query can depend on its context, such as a history of preceding events. For instance, in some applications, one needs to count the events (e.g., the number of entered pincodes in an ATM machine to detect fraud), in others, one does not (e.g., when an order is canceled; canceling it many times has the same effect).
5. **Event complexity**. Some events are derived from many events, e.g., a storm about to occur in an area A. To detect a storm, one needs to gather different values in particular areas (e.g., temperature, pressure, humidity), run sophisticated meteorological models, and apply the rules that lead to a storm.

Regarding the last point, such events are usually denoted *complex* (sometimes called composite) events, to be distinguished from primitive events. Primitive events are issued from sources (e.g., a thermometer or a RSS feed). They contribute to the detection of complex events. Whether an event is primitive or not is system dependent. Typically, events that enter the EBS are considered as primitive in the given scope. Note that sometimes "complex" and "composite" are used interchangeably. However, there is a trend to define a complex object as an elaborate object such as a storm composed of object and additional information (e.g., simulation methods) and a composite object as "only" composed of other objects. On the other hand, the term "complex" is often used by the database community because of the analogy with complex objects in non-first-normal-form models.

Complex events are derived from other - primitive or composite - events. They are specified in expressions (referred to as *event queries)*[2] using operators such as logical conjunction or disjunction. At a high level, the specification of such an event is, for instance, "temperature on Sensor S1 greater than 40C and temperature on Sensor S2 greater than 45C". Different applications consider different sorts of complex events. These can be more or less elaborate, for instance depending on whether they take time constraints into account or not (e.g., temperature on Sensor S1 greater than 40C and temperature on Sensor S2 greater than 45C within a five-minute interval). This can lead to sophisticated algebras and languages.

Basic event definitions To define the various types of events that are handled by the system, let us first consider the cases displayed on Figure 2. The figure displays four lines, which indicate different situations and events of interest.

[2] sometimes called *profiles* in the literature.

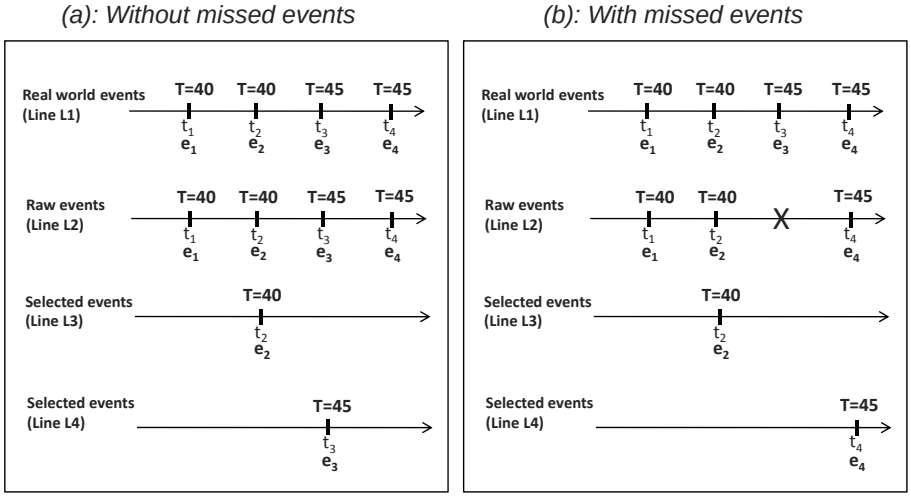

Fig. 2. Real-world events, raw events, and events of interest (without missed raw events (a) and with a missed raw events (b))

Real-word events: Real-world phenomena happen at certain times, e.g., e_1 at t_1 or e_2 at t_2 on the lines L1 in Figure 2. We denote them *real-world events*. These events may be sent to the system, in which case we denote them *raw events*. Those are the first events processed by the system and they are not necessarily "of interest".

Event occurrence: When a real world event takes place, we refer to its *occurrence*. An occurrence is associated with a time, the unit of which is expressed using a time referential. The latter itself depends on the application (it can be expressed in minutes, hours, days, and so on). As it is often the case in current modeling approaches, an event has no duration but a time stamp at which time it occurs.

Event model: Generally speaking, an event is modeled as a 3-tuple:

$$e = (e_{id}, t, < e_{val} >),$$

where e_{id} is the identifier of an event, t its time of occurrence, and $< e_{val} >$ a collection of attribute-value pairs (A_i, a_i), as for instance in the collection $< (Temperature, 40), (Humidity, 60) >$. We denote such collections *event instances*. This notion is important for handling duplicate events such as in the example of the ATM machine where three subsequent instances of a certain event trigger the blocking of a card.

Unexpected event: Several types of events can be defined. For instance, some applications need to consider *unexpected events*. They can correspond to an unusual value for a real world event (e.g., an unusual humidity value on a sensor) or be some aberration at the application level when detecting complex events.

Multiple occurrences of events: Sometimes one talks about *multiple occurrences* of an event, that is, using our simple model above, the same $<$ (*Attribute*, *value*) $>$ collection happening at different times. The event is theoretically the same, just its time stamp changes.
Event detection: When an event such as a storm, a fraud, or an empty shelf in the store is recognized by the system, we refer to its *detection*. There may be a time lapse between event occurrence and event detection. Note that the process of event detection occurs on different levels of abstraction, e.g., when capturing a sensor measurement or evaluating complex expressions. Generally speaking, detection is the match of an expression (query) that specifies an event of interest (the result of a query). This is implemented by a *filter*. Filters can build hierarchies and thereby implement hierarchies of query operations. The fact that an event is of interest depends on the considered hierarchy level. When an event matches a condition it is *selected*.
Obviously, noisy or faulty input data directly impacts the value of a collection of raw events, hence event selection. On lines L3 in Figure 2, if the interesting events correspond to the fact that the temperature stays constant for some time, then e_2 will be selected. Suppose now that the event of interest is the fact that the temperature value increases. On line L4, left-hand side of Figure 2, e_3 will be selected. If for some reason e_3 had been missed (e.g., defective sensor), then it would not have been included in the collection of raw events (compare lines L2 on both sides of Figure 2). In this case e_4 would have been selected instead, as indicated on line L4 on the right-hand side of Figure 2. In this case e_3 may be inferred later or even detected *at posteriori*. Some operations consider many successive events but do not need to keep them, e.g., when detecting the highest value in a sequence of events.
Event representation: An event representation is the encoding of an event (a sort of "materialization"). Within the EBS community it is customary to refer to this encoding as an *event message*. The encoding communicated to the subscriber is often denoted a *notification*.
Trace: Raw events may be kept in a history often called *trace* in order to detect history-related complex events (e.g., average temperature greater than 40C in the past three years). The trace can be compacted and values aggregated but such considerations are out of the scope of this section.
Subscriber/publisher: The entity that formulates a query on events is usually referred to as a subscriber. It can be for instance a person, as in our illustrating example, or a machine. It can also occur at a lower level of abstraction. The sources of information that encompass the events are denoted as publishers.
Event stream: An event stream is usually considered as a linearly ordered sequence of events of a particular type, where the order is given by the time of arrival in the system. Typical streams are stock values in financial applications.

2.3 General Mechanism of Event Handling and Alerting

In the general mechanism illustrated in Figure 1, a condition is specified in the form *IF condition THEN action*. If a situation matches the expression given

by the condition, a defined action takes place, e.g., alerting the supervisor. The active database community often refers to this mechanism as "Event - Condition - Action (ECA)" rules. That is, *events* occur and if they follow some *condition* some *action* is taken. Note that we consider actions taken as being outside of the scope of an EBS. In other words, without loss of generality, we consider only deductive rules that operate on event data. Reactive rules are realized by notifying components or actuators that execute the specified action.

The expression that specifies the condition is often called a **pattern** and has for instance the form "E1 or E2" or "E1 then E2 then E3", in which case it includes the consideration of operators (e.g., from temporal logics). Such an expression can lead to different semantics. Ei is a logical expression that is applied to values and that can be for instance (t > 40C). It is sometimes denoted as a *constraint*.

Note that EBS are sometimes reduced to "publish/subscribe" systems. In such systems (e.g., a digital library) sources *publish* new information - which constitutes events of interests - and users *subscribe* to new information. For example, in a digital library application, one may be interested in new publications with the term "Interoperability" and subscribe to a data source (e.g., a web portal) with this keyword. Then if a new article or special issue with keyword "Interoperability" is published the user will be notified. The major challenges in designing such systems include the expressiveness of the subscription language and the specification of efficient middleware to handle the filtering process. For more information on this topic the reader is referred to [40].

3 Layered Approach

In this section, we introduce a structure for event-based systems that follows the design pattern of layers. Layers describe system features on different levels of abstraction. Each layer is implemented using features of the next lower one. This principle allows stepwise refinement of the system from high level concepts to low level functionality. The interfaces between layers enable abstraction from concrete implementations. Thus, the layers are not bound to a specific technology and application designers can choose from different options without affecting the rest of the system. However, this requires a clear definition of interfaces and features for each layer. The challenge is to find an appropriate decomposition of the whole system into well-encapsulated, self-contained, layers.

Often, existing solutions for event-based system only cover a range of different aspects such as query definition, event processing, and communication. For a layered architecture we want to separate such aspects from each other to keep concrete implementations exchangeable. The Mātārere layers provide such a decomposition under consideration of different implementation options. This allows choosing the most suitable implementation option for each aspect in a given environment and helps with the orchestration of technologies for event-based systems.

This section first presents the five layers that can be seen in an event-based system. It then presents the layer model, i.e., the concepts that can be applied at each level. Note that this approach is conceptual and that we are not concerned with issues such as performance or scalability. Finally, this section ends with the description of the recursive mechanism to apply the layer model on each layer.

3.1 Five Layers in Event-Based Systems

We propose the decomposition of event-based system into five layers. These are the Application Layer, the Language Layer, the Execution Layer, the Communication Layer, and the Capturing Layer.

The *Application Layer* comprises high level end-user applications that use event data for serving application specific requests. An example is an application that displays a live chart of a stock. This application requests the needed events from a subsystem and visualizes the event data. The subsystem and its implementation are transparent to the user.

The *Language Layer* provides means to register event queries using a high level language. Queries on this level logically define events of interest in a unified format, often in an SQL-like fashion but we will see that other approaches are possible. The Language Layer thereby provides an abstract interface for "retrieving" events and a unified view on the event sources. It allows defining the query while abstracting from details regarding the query execution.

The *Execution Layer* orchestrates operations for queries on events. It defines the processing logic for generating the desired output from incoming event data. That is, it logically integrates different data sources and makes the operations for query execution explicit. It also instantiates or invokes concrete software components for retrieving and processing events. This layer thereby defines the concrete systems and resources that perform a given event query task.

The *Communication Layer* enables event-based interaction between event sources and processing operators. It provides interfaces for publishing/subscribing and implements the mechanisms for exchanging event messages. In a distributed systems this layer also covers methods for network communication, such as event routing and subscription routing.

The *Capturing Layer* comprises the software components that provide access to event sources. Depending on the application, this could be for instance the API of a sensor or an RSS feed. Events coming from the Event Capturing Layer are the most primitive events in a given system scope.

Table 1 presents an overview of the layers in Mātārere. Note that not all layers are necessarily in place in every application. In simple setups, the applications may directly access lower layers. For instance, not all systems require a high level query language and application programmers directly address operations of the execution model. However, leaving out layers comes at the cost of limited flexibility in the system design and increased complexity of applications.

3.2 The Layer Model

We specify all system layers using the same set of concepts. These are data, operations, and key actions on the layer. We subsequently introduce each concept briefly. Table 1 presents an overview of what the concepts refer to in the different layers.

Table 1. Abstraction layers for event-based systems

Layer	Data	Operations	Key Actions
Application Layer	Application specific representation of query results	Application specific operations on query results	Application specific orchestration of queries issued to the system
Language Layer	Data model of query language (e.g., tuples)	Declarative operators (e.g., "followed by" to order events in a pattern or filter predicates on attributes)	Query expression in the language format (e.g., rules in an ECA format)
Execution Layer	Data model of rule engine (implementation specific data structures)	Logical operators of rule engine (e.g., transitions between states in a state machine)	Orchestration of operators in execution model (e.g., states and transitions in a state machine)
Communication Layer	Source specific formats with common envelopes or layer specific event encoding (e.g., XML expression)	Publication of events, subscription to events, and routing mechanisms	Forwarding of events and implementation of routing strategies
Capturing Layer	Source specific formats (e.g., XML expression)	Interaction with event sources (e.g., an http request)	Making calls to event sources and providing event data (e.g., to the API of a sensor gateway)

Data The concept of data describes the information entities that a layer handles as well as the corresponding data models. For instance, RFID readers in the Capturing Layer may provide RFID read events in the PML format. In contrast, the Execution Layer may use tuple streams in the proprietary format of the chosen rule engine.

Operations Each layer supports a specific set of operations. These can be for example operations on event data, such as filtering or correlation. Other operations do not operate on event data but address system components. Examples for such operations are subscriptions and unsubscriptions at event sources.

Key Actions Each layer realizes functionality on a certain level of abstraction. The key actions describe a layer's means to implement a certain higher level functionality. That is, operations on layer N find their correspondence in an orchestration of actions on layer N-1. Each layer may use different primitives for defining these actions. For instance, the Language Layer may define a query with relational operators and predicates on streams, following a declarative paradigm. Lower layers may imperatively define a sequence of primitives that realize the corresponding higher level operation.

3.3 Cascading Event-Based Systems

The above-introduced layer structure covers all system levels in an event-based system. It is important to point out that the presented layer stack may run on different system levels (recursion). Complex solutions can build cascades of event-based systems, each supporting event handling on a different level of abstraction. We illustrate this concept along an example for monitoring business processes.

Consider a business intelligence application that monitors the production progress in several production plants (see Figure 1). The first (upper) stack of system layers ranges from the monitoring application to the production plants of the company. The upper end is the monitoring application in the Application Layer of the first stack. The first stack's lower end is populated by *Manufacturing Execution Systems* (MES) in the plants. MES provide events about the progress in the plant and thereby implement the Capturing Layer for the upper stack of layers. A system designer may just look at the scope above MES for implementing an event-based system. However, the Mātārere layer structure repeats at a lower level. The MES that implements the capturing Layer at the upper level also implements the Application Layer of the second (lower) layer stack. The lower stack of Mātārere layers goes down to the machine interfaces on the plant's shop floor which provide the functionality for capturing low level events. By using cascades of layer stacks, complex systems can drill down from very complex and semantically rich event structures to very primitive event sources.

4 Layers

In this section we describe the layers of our architecture. For each layer we focus on the key aspects that the layer must implement and point out the most important challenges. We align our discussion along the concepts of data, operations, and execution plans as introduced in Section 3.

4.1 Application Layer

This layer marks the upper end of our architecture. It comprises applications that *use* event data. An example is an application that displays a stock chart. The application retrieves the required input events through the lower layers of

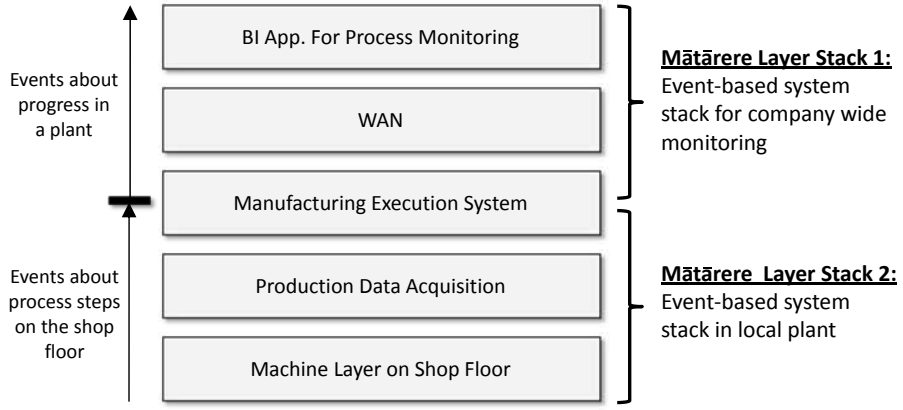

Fig. 3. Cascade of event-based systems

our architecture. In other words, there is no event detection on the application layer. Further details for implementing this layer are out of the scope of our framework.

4.2 Language Layer

The language layer as we define it provides means to express event queries in a declarative way. It is therefore supported by an event model and operators to be applied on event collections.

To specify the events of interest, two main approaches are distinguished (in the following we refer to these approaches as *event querylanguages*):

1. Algebraic approaches. Such approaches are independent from any language syntax and can be mapped onto existing languages at implementation time.
2. Language-based approaches. In this category, we distinguish *rule-based* approaches from approaches that are based on existing *database* languages (mostly SQL). The queries are then directly expressed in the database language such as SQL extended to time-related issues. The advantage is that one can use an existing compiler for this language. [58] refers to "event processing languages" and makes the distinction between imperative script languages, SQL extensions, and extension of inference rule language.

When defining an event language, the key issues to be considered are:

- its *expressiveness* - as some applications must be able to encompass a large number of situations,
- its *simplicity of use* - as one does not want to deal with complicated expression,
- the *semantics of the operators* which must be able to capture the desired events.

Relying on existing languages or paradigms allows one to take advantage of their expressiveness and their possible simplicity of use.

Data The data considered on this layer are the events themselves, modeled differently according to the approaches. Event models have been briefly introduced in Section 2. In that section, we chose a simple approach to describe an event, namely, the relational model in which an event was defined as a 3-tuple. There is a direct correspondence with its representation in relational tables when using SQL-like language to manipulate such events. When using an algebraic approach, an event is typically a logical expression. In many applications, unique identifiers (ID) are associated with events. However, applications that handle data streams often do not have this feature.

Many event models consider events without duration (e.g., [16, 70]). Some event models consider events with duration [24]. Common to all these models is the consideration of a discrete time model.

With some underlying simple paradigms some problems arise when considering composite events. Composite events are either derived from other events or composed of other events, in which case they are similar to complex events in the database vocabulary. The relational model in 1 NF does not consider nested relations, so an artifact has to be found to overcome this situation. In an object-oriented model, on the other hand, such a representation comes naturally.

Operations The operations at this levels are the operators defined on collections of events. The idea is to link possible events with operators from the logical and temporal domains (which can be also extended to other domains such as the spatial domain, but this is not our focus here). The collection of operators can be more or less elaborate and then encompass a large number of situations without ambiguity.

Their specification differs according to the underlying paradigm. We use below the simple classification of event languages given above:

1. Algebraic approaches. The typical operators are disjunction, conjunction, negation, sequence, and temporal restrictions such as "before", "after", "within *time interval i*", "at *time t*", *nth occcurrence of event e* (but many other temporal restriction can exist). An important point is the expressive power of the proposed language. Using regular expressions (such as in [33]) to describe events of interest gives the expressive power of a regular language. In this category we can cite [16], which is the basis of the Snoop system [17], YALES (Yet Another Language for Event Specification) [70], [11], or [64]. EVA (An EVent Algebra Supporting Adaptivity and Collaboration in Event-Based Systems) [39] is a high level algebra that allows one to define appropriate operators for applications in many domains.
2. Language-based approaches. The idea is to extend an existing database language for specifying events. Hence many approaches rely on SQL extension to event queries. The major introductions are time-dependent operators such as

"after" or "before" [6] but also different types of time windows (e.g., sliding, tumbling) [8].

Key Actions The key action at this level is the specification of the query expression that describes the events of interest. This expression can be elaborated. It encompasses in particular the notions of selection and consumption of events. When implementing event detection, if an event matches a condition, which is part of the query expression, - i.e., if it satisfies a criteria such as "temperature greater than 40C" - then it is *selected*. In a list of selected detected events, an event can be used (*consumed* using the vocabulary of the event community) instantaneously in a query (for instance, in a conjunctive expression) and either (1) "forgotten" (discarded) or (2) kept for future (re)use in a trace. For instance, if the application is interested in the median temperature value for the last hour and if the temperature is measured periodically with a granularity finer than one hour, then all the measurements will be stored in a trace to eventually detect the event or not. More information regarding event selection and consumption can be found in [70, 71, 29, 11]. These approaches illustrate the fact that selection and consumption can be specified at this high level of abstraction (the language layer) even though they are implemented by a filter in the next layer.

4.3 Execution Layer

The Execution Layer provides operations to *execute* event queries. This is in contrast to the Language Layer that *describes* the execution but provides no means to process the query. Software components that implement the functionality of the Execution Layer are often referred to as processing engines (in some contexts also called rule engines or stream engines). The Execution Layer gets an executable orchestration of operators, i.e., a graph of operators and communication links between them. It then invokes the needed operators on concrete processing resources. In a decentralized system, this may involve decomposition and distribution of query parts. The Execution Layer is also in charge of establishing communication links between processing operators and event sources. Therefore it uses functionality provided by the Communication Layer.

Data: The data model of the Execution Layer refers to the implementation specific data types required for event processing. This can cover models for the events themselves, data structures for intermediate and partial results (i.e., synopsis for handling traces), and models of the data sources (e.g., event streams).

The encoding of events is defined by the execution model. That is, the representation of events as they can be processed by the given implementation of operators. Examples of event encodings are XML messages [36, 54] or implementation specific encodings of tuples [41]. Ideally this data model is the same as in the models in the Communication Layer and the Language Layer. If this is not the case, data conversion is required for interaction with these layers.

The encoding of intermediate and partial results refers to temporary data storage as part of event processing. Often, synopsis data structures are used for this purpose. Examples are wavelets [34] or histograms [63] for capturing value distribution in a stream or an active stack for sequence extraction [66].

Models of the data sources can capture source properties (metadata) that are relevant for event processing. This can be for example information about the data rate and whether or not the incoming events are strictly ordered or ordered to a certain degree (e.g., k-sorted). Processing engines can use such information for optimizing their operations. It depends heavily on the implementation of the processing engines which information about sources is relevant.

Operations: Execution models of different processing engines vary in their capacities to detect events and the transformations which they can perform on events to detect event patterns. Implementations of an execution model are often referred to as rule engine or stream engine; dependent on the underlying paradigm. In general, processing engines facilitate (i) detection of events and (ii) generation of corresponding event notifications. Figure 4 illustrates the processing steps in a generic event processing operator. How these steps are conducted can vary between different implementations.

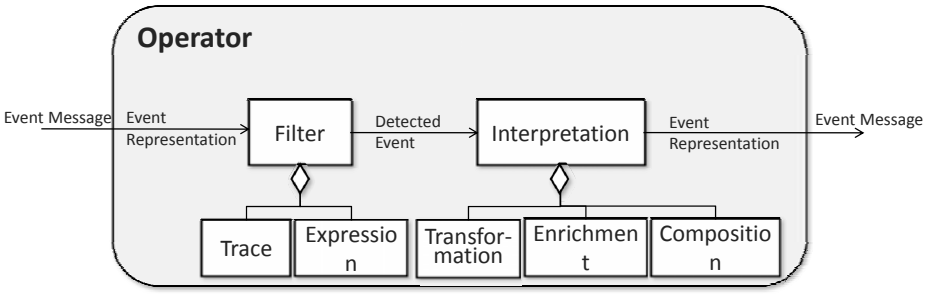

Fig. 4. Generic processing steps

Stream engines usually implement operations of the relational algebra over streams [7]. Stream engines convert parts of the streams (defined by windows) into relations. Operators keep such relations in buffers or synopsis data structures. The processing logic runs over these buffers and new incoming events. Intermediate results are passed on to subsequent operators in the query tree. Other approaches use variations of state machines [57, 9] or Petri nets [32] as their underlying processing paradigm. These approaches are tailored to the detection of complex event patterns. The implemented operations are state transitions based on events. The processing engine maintains a state that can change due to incoming events. The processing operations evaluate incoming events based on the current state and perform a state transition if query conditions are

met. A complex event pattern is detected when the state machine reaches one of its finite states. Subsequently the engine may apply predicate checks on the detected event patterns to account for constraints in the query condition. Note that some implementations already facilitate some constraint checks along transitions. A notification is issued if an event pattern and corresponding predicate checks are conducted successfully. Table 2 gives an overview of the operations that the Execution Layer provides.

Table 2. Operations provided by the Execution Layer

Operation	Mandatory	Functionality
Filters	Yes	Checking conditions on incoming events. (This refers to the pattern and constrain part in a PCA-style query)
Interpretation	Yes	Generating a representation of detected events. (This refers to the action part in a PCA-style query)
Orchestration	Yes	Defining links between operators and event sources (to build the query tree)

Internal Implementation: Implementation in the Execution Layer deals with the implementation of operators and - particularly in distributed systems - their deployment on processing resources. The implementation of operators for event processing must consider the special nature of event data. Challenges in this field include dealing with the streaming nature of input data, scalability issues, and challenges in handling time. Event data continuously arrive at the processing engine that may be out of order and come in bursts. The execution model must ensure seamless processing under these conditions. Such challenges were addressed in a broad range of projects (see for instance [7, 4, 3, 69, 18, 22]).

Scalability becomes a concern when processing events at a high rate or when pattern evaluation creates large intermediate results. The latter is particularly a challenge when event patterns span large time periods. In that case, a large amount of candidate events must be kept in memory before the pattern evaluation can be completed. Approaches to deal with this aim to reduce intermediate result sizes by applying predicate checks early in the query processing [66] or to drop outdated events [68, 61].

Actively handling time is relevant if event patterns have temporal constraints. In particular, detecting the non-occurrence of an event requires activities based on timers. In distributed systems this results in challenges of time-synchronization between different resources. Delay of event messages must also be taken into account.

Deployment deals with the invocation and allocation of concrete resources in the event-based system. This is particularly challenging in distributed environments. The underlying optimization problem is finding an optimal mapping

of the operators in a given query tree to resources in the network. Optimization goals include minimizing the network load in the system, minimizing delay, maximizing reliability, using processing resources efficiently, and a combination of these goals. The relative importance of these goals is heavily dependent on the application and the application domain. For instance, applications in sensor networks typically aim to reduce network load, as this saves power consuming communication and increases the application lifetime. Solving the optimization problem of optimal query distribution includes NP-hard problems. For instance, minimizing network load includes the NP-hard problem of finding a Steiner tree [30]. Thus, a range of heuristics exists for approximating an optimal solution [5, 60, 56]. These solutions differ in the targeted optimization goals and the factors they take into account. A major distinction can be made between approaches that are network aware (i.e., consider network properties in the optimization) and approaches that solely focus on the processing resources.

Related to the problem of operator mapping is remapping. Remapping operators can be reasonable if properties of the infrastructure changed such that the previous mapping decision has become suboptimal. For instance, network properties such as delay along certain links or available bandwidth can change over time. However, one must trade the overhead for remapping running queries against the expected gains [67].

Key Actions: The key actions of the Execution Layer can be described in terms of sessions. A session starts with registering a query. Figure 5 gives an overview of the process. At this point the query is represented as a graph that defines query operators and communication links between them. The Execution Layer deploys and invokes the binaries for executing the query. For establishing the communication links it uses the functionality of the Communication Layer. The Execution Layer subsequently passes back query results until the query is revoked and the session ends.

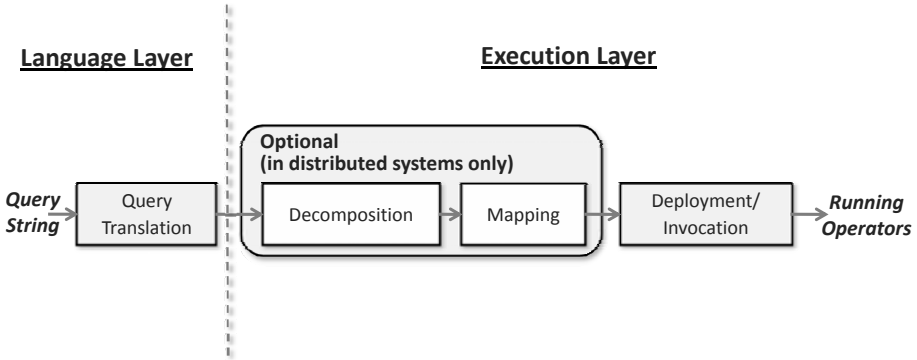

Fig. 5. Registering query operators

4.4 Communication Layer

The Communication Layer provides means for exchanging event messages. It facilitates indirect addressing and thereby enables the event-based interaction scheme. That is, it implements the functionality of notification service and underlying event bus [28]. Higher level components use the Communication Layer for interaction with each other and for retrieving events from the Capturing Layer.

Data: The data model of the Communication Layer must be uniform to the extent that it allows matching published event messages to corresponding subscriptions. For instance, in topic-based subscriptions the Communication Layer must be able to determine the topic of an event message. The required uniformity can be achieved in two ways. One way is that all event sources agree on a format for event encoding that is interpretable by the Communication Layer. Another way is that the sources wrap the event message in a uniform envelope that holds the required metadata for the Communication Layer.

Operations: The operations on the Communication Layer allow higher system layers to use anonymous, push-based communication. It must therefore provide methods to subscribe and unsubscribe for events of interest. It must also provide methods to publish an event, i.e., to push an event to the communication layer to notify the corresponding subscribers about this event.

Dependent on the implementation, the subscription mechanism may be more or less elaborate. That is, the subscription mechanisms may provide filters of different complexity for specifying events of interest. Options include the use of channels (or topics), subjects, types, or a general expression over the event data [28]. If channels are used, publishers associate notifications with a certain named channel. Subscribers subscribe to a channel name and get all notifications associated with the corresponding channel. A popular implementation of such a scheme is provided by the Java Message Service (JMS). If subjects are used, publishers associate notifications with a string that denotes a path in a subject tree. Subscribers can use a string filter to define the subjects of interest (e.g., "sports.football.*" for any event about football). If types are used, publishers assign a type to a notification that stems from an inheritance path of types that may include multiple inheritances. Subscriptions are defined by the type of interest and cover all subtypes. An implementation of this scheme can be found in the Hermes system [55]. The most expressive subscription mechanism is to use general expressions over the content of the message. For instance, subscribers may define events of interest by defining a threshold value for an event attribute. Such mechanisms exist with different expressive power. For instance, the subscription mechanism of JMS allows defining filters over the message content similar to a where clause in SQL. Table 3 gives an overview of the operations that the Communication Layer provides.

Table 3. Operations provided by the Communication Layer

Operation	Mandatory	Functionality
Subscription	Yes	Registering for certain events
Unsubscription	Yes	Unregistering for certain events
Publishing	Yes	Pushing an event message to the system
Notifying	Yes	Sending a notification about an event to a corresponding subscriber
Advertising	No	Announcing that certain events will be provided by a certain source
Removing Advertisement	No	Announcing that certain events will no longer be provided by a certain source

Internal Implementation: It is possible to implement the functionality of the Communication Layer with different architectures and different methods for internal dissemination of events. The simplest architecture is to use a central component that relays all event messages. Publishers push events to this component and subscribers receive events from the same component. However, while this architecture is simple, the central component yields a bottleneck for scalability. Alternative architectures therefore avoid a central component by using network of so called brokers. Such a decentralized architecture also enables to improve reliability [59]. Several approaches for such broker networks exists that vary in the network architecture and the mechanisms for internal message routing.

An early research prototype is the Rebeca system that structures brokers in a tree [27]. Publishers and subscribers interact with so called border brokers and the event messages flow through the system via so called internal brokers. Yet this system still yields bottlenecks at the root of the tree, a problem that is overcome by pure peer-to-peer architectures for broker networks (see for instance [55, 62, 52]).

Implementations further vary in the employed methods for disseminating event messages in the network. The simplest method is to flood the network with all event messages. More efficient methods use routing tables within the network to avoid communication overheard and direct the message flow to the interested parties. The table entries associate event filters with communication links in the network. Incoming events are forwarded along the links that correspond to matching filters and are thereby routed to the subscribers. The challenge is to update the tables in the network efficiently when subscriptions change. A range of different algorithms exist for building the routing tables. These algorithms are closely interrelated with the employed subscription method. The main strategies are based on (i) flooding subscriptions, (ii) identify-based routing, (iii) checks on the filter semantics, and (iv) matching subscriptions against advertisements.

With the use of flooding, each broker forwards incoming subscriptions filters and unsubscriptions filters to all its neighbors (except to the origin of the sub-

scription) and updates its routing table accordingly. With identity-based routing, brokers only forward filters if they have not forwarded an identical filter before [50]. This limits redundancy in subscription routing. With checks on filter semantics, brokers only update their neighbors if previous subscriptions do not already cover the new one [14]. They may further combine (merge) multiple subscriptions into one that covers all [49]. With matches against advertisements, brokers consider advertisements from event sources as well as subscriptions from subscribers. This allows reducing the overhead of communicating subscriptions toward event sources that do not publish corresponding events. Advertisements can be routed though the network in a similar fashion as subscriptions. Different strategies exist with regards to where and how advertisements and subscriptions meet within the network [12].

Key Actions: Key actions at the Communication Layer can be described as sessions from two perspectives. One is the perspective of publishers and one is the perspective of subscribers.

For *subscribers*, a session starts with subscribing for events of interest and ends with un-subscribing for these events at the Communication Layer. During a session, the subscriber gets all events that it subscribed for from the Communication Layer. Note that there may be different quality guarantees provided by different implementations of the Communication Layer. Different implementations can make different guarantees about the delay of event notifications, the completeness of notifications, and their temporal ordering.

For *publishers*, the session starts with registering at the Communication Layer. Dependent on the implementation this can go hand-in-hand with publishing an advertisement about the events this publisher will provide. The session ends with revoking the advertisement and un-registering at the Communication Layer. During the session, the publisher needs to push all events that match its advertisement to the Communication Layer.

4.5 Capturing Layer

The event Capturing Layer provides interfaces to the event sources in the system. That is, the event Capturing Layer is the layer where events enter the system. All operations below this layer are out of the scope of the Mātārere layers. The nature of the event sources depends heavily on the application domain. Examples include sensor nodes in a sensor network, Web services, or RSS feeds.

Data: The data model in the event Capturing Layer is the data model of the event sources in the system scope, e.g., the output of sensors. Note that this data model may be very primitive and simply contain the value of a sensed property (e.g., a temperature value). However, event sources often enrich the provided values with an identifier for the event source, a timestamp for the event, and possibly additional attributes. For instance, TinyDB can provide sensor measurements along with an ID for the corresponding sensor node and with a

timestamp [46]. Another example is the event model of EPCIS for retrieving RFID events [25]. This event model for RFID observations includes a timestamp and optimally an ID for the corresponding read point.

Operations: The operations on the event Capturing Layer allow higher system layers to implement the observer design pattern; a fundamental pattern in event-based systems. It must therefore provide methods to register and unregister for events and mechanisms to actively call the registered observer methods. If sources only support pull-based communication, the event Capturing Layer has to mask this by regularly polling these sources. In such cases, event sources can provide methods to define the frequency for polling. The higher layers can use this feature for dynamic adaption of the sample rate [51].

Dependent on the event source and application, the provided events may be very primitive (e.g., simple sensor measurements) or have rich sets of attributes (e.g., events about completed business steps). Event sources may provide options to project out event attributes during registration. Also, some sources may allow specifying filters along with the registration for events. For instance, one may register only for events about temperature measurements above a certain threshold. Another example is to configure RFID readers to report disappearance of an object only if it was not observed in a certain number of consecutive read attempts. Table 4 gives an overview of the operations that the Capturing Layer provides.

Table 4. Operations provided by the Capturing Layer

Operation	Mandatory	Functionality
Registration	Yes	Registering observers at an event sources
Unregistration	Yes	Unregistering observers at an event sources
Projection	No	Specifying the relevant event attributes
Filtering	No	Specifying operations on the raw events

Internal Implementation: Operators in the event Capturing Layer have access to raw event data from the local event sources and can apply filters on these events. Dependent on the type of source, raw events can include noise and errors, such as missed RFID tags in an RFID read or noise in a sensor signal. The event Capturing Layer may implement cleaning filters to reduce errors in the events before passing them on. For instance, RFID readers typically apply low pass filters to cancel out false events about appearance or disappearance of an object and to suppress so called flickering [10].

Furthermore, the Capturing Layer can enrich raw events with additional data before passing them on to higher layers. It therefore may correlate event data with data from additional sources, e.g., relational databases. Typically an event

is at least enriched with a timestamp (if not already included in the raw event). Furthermore, an event can have attributes that provide information about the source or the context in which it occurred. For instance RFID events in business applications are commonly enriched with the ID of the corresponding reader and possibly the business step during which it was captured [25].

Key Actions: Key actions at the Capturing Layer start with registering an observer at the callback interface for the event source and end with unregistering the observer. The Capturing Layer then collects the required events from its local sources (e.g., sensing devices). The observer receives all events that it registered for until the registration is revoked. The Capturing Layer may perform some preprocessing on the captured events - such as cleaning - before passing on to higher layers.

5 Cross-Layer Issues

The presented architecture encapsulates tasks in event-based systems and separates them into layers. In a clean layered design, the interdependency between layers is limited to the adjacent interfaces and each layer only interacts with its directly neighboring layers. However, there are situations where system designers may want to make exceptions to these strict design rules. This can be for the sake of cross-layer optimization or to avoid overhead in simple applications. In this section we discuss such issues that arise across layer boundaries.

5.1 Transitioning between Layers

In a layered architecture, layers receive requests from the adjacent upper layer and issue requests to the adjacent lower layer. Along this way the query tasks are split into subtasks and transformed into consecutively lower levels of abstraction. In the following we discuss this transformation throughout the Mātārere layers. Figure 6 gives an overview of the transitions.

The Language Layer interacts with the Application Layer. From there it receives event queries in the provided query language. In the next step, the query is compiled, i.e., translated into an executable model that can be passed on to the Execution Layer. The general form of a executable a model is a directed graph where nodes represent query operators and edges represent the data flow. Details of this model depend on particularities of the execution layer. The process of compiling facilitates the transitioning between the Language Layer and Execution Layer. Thus, the compiler provides abstraction from execution.

The Execution Layer receives an executable model from the Language Layer and invokes the respective operators. In a distributed setting this involves the decision which operator should run on which resource. Invoking operators corresponds to implementing the nodes in the directed graph which describes the execution. Implementing the edges in the graph corresponds to establishing the communication links between operators. The Execution Layer implements this

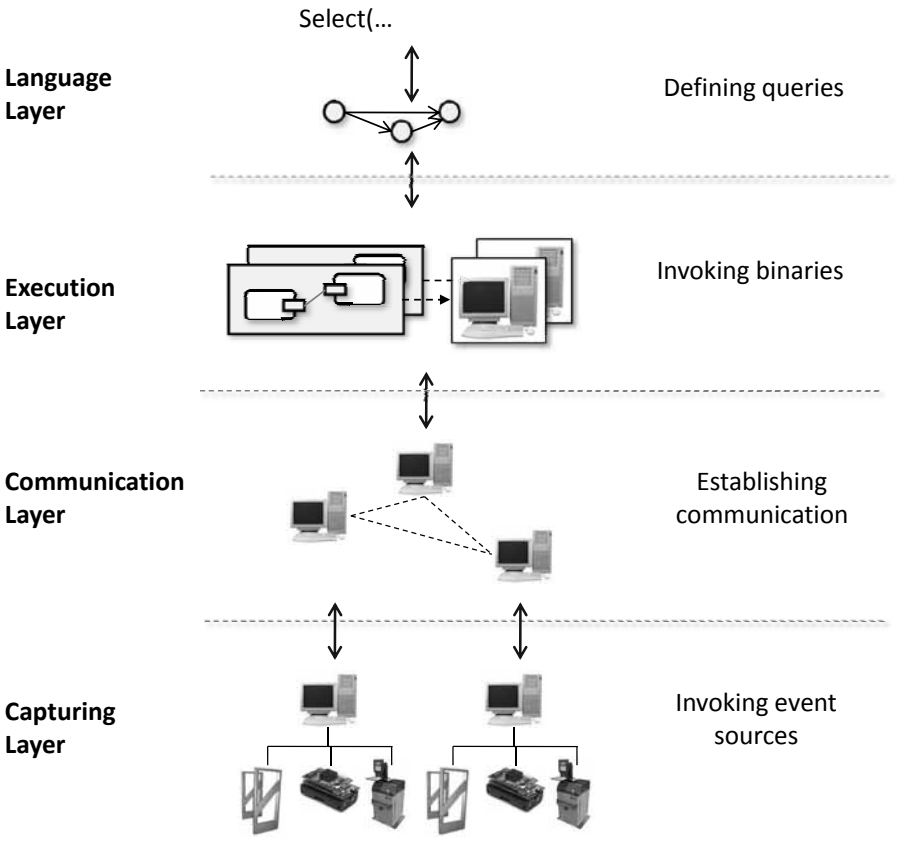

Fig. 6. Transitions between layers

using the functionality of the Communication Layer. That is, it issues subscriptions and possibly advertisements.

The Communication Layer receives subscriptions (and possibly advertisements) from the Execution Layer and realizes the exchange of events accordingly. In distributed systems this typically involves establishing routing tables. The Communication Layer is also responsible for invoking the event sources according to the subscriptions from the Execution Layer. That is, it must register at the corresponding callback interfaces. In distributed systems this is done by so called edge brokers that directly interact with publishers and subscribers.

The event Capturing Layer receives registrations from the Communication Layer. It is then in charge of capturing the corresponding events (e.g., by sampling sensor data) and providing them in a push-based manner.

5.2 Crosscutting Concerns between Layers

In a strict layered model, higher layers send request to lower layers and receive the corresponding results. Yet, in some situations it can be beneficial to disobey this design guideline for the sake of cross-layer optimization. In this context we discuss in the following (a) the distribution of filter operations across the lower three Mātārere layers and (b) cross-layer optimization of the query structure.

Event filters can run on the Execution Layer, the Communication Layer, and the Capturing Layer. The Capturing Layer can run filter operations over the local event data that it provides. This can allow for selective registration at the callback interface. For instance, a sensor source may only transmit values if they exceed a certain threshold. The aim is generally to push filtering as close to the sources as possible for reducing intermediate result sizes. However, dependent on the filter and application environment, this may push heavy computational load to resource constrained devices. Srivastava et al. describe the corresponding trade off [60]. Further filtering can take place in the Communication Layer. That is because the Communication Layer matches events against subscription filters. The expressiveness of such filters depends on the implemented subscription mechanism. Theoretically there is no limit to the expressive power that subscription filters may provide. However, typically this is limited to filter operations on single events and without considering traces. More complex filters are commonly realized in the Execution Layer.

Cross-layer optimization of the query structure refers to the problem of finding an optimal query deployment. This is the task of the execution layer. However, optimization may involve adapting the execution model for a better fit to the given network structure. For example one can calculate and consequently distribute aggregates in different ways (e.g., sum(sum(a1,a2), sum(a3,a4)) or sum(sum(a1,a4), sum(a2,a3))). Different execution models for the same query can result in different mappings to processing resources. Thus, finding the queries' execution model that allows for the best deployment involves the language layer and the execution layer.

5.3 Bypassing Layers

The functionality specified in the Mātārere layers support application developers in building event-based systems. This is similar to the support that the OSI model provides for developing networked applications. Yet, as with the OSI model, not all layers of Mātārere must always be in place. Top-down applications may cover any number of the Mātārere layers. For instance, a simple application may directly access event sources. Other applications may directly access the communication layer for retrieving events. Bypassing layers in this way can be reasonable if the applications requires only very simple versions of the respective layers' functionality. For instance, simple queries do not need high-level language support. However, bypassing layers comes at the cost of reduced flexibility in the system design.

6 Conclusion

The field of event processing has received a great deal of interest in the past two decades, from many communities. As a result, the same problems are sometimes tackled from different view points (e.g., from a database or a software engineering view point) and many different terminologies have been introduced for the various entities to be handled in event processing. Also, one refers, for instance, to Complex Event Processing (CEP), Even-Driven Architectures (EDA), event Notification Systems (ENS), Alerting Systems (AS), and more recently Event-based Systems (EBS), to cite the currently most common terms. The specific functionalities of these systems are often unclear and the borders between subsystems or modules are sometimes fuzzy - for instance, one would sometimes consider filtering as being inside an ENS. Given all these different view points, it seems necessary to consider a general approach for modeling event-based systems and defining properly all terms and entities to be considered in such systems.

This paper presented a structured framework and terminology for specifying event-based systems. We presented a layered architecture that encapsulates elementary tasks in event-based systems and discussed three major aspects: (1) data to consider, (2) operations to be performed, and (3) actions to be taken for each of the defined layers. We thereby provide a framework which shows similarities between event-based systems and other application domains and which can serve as underlying structure for system implementation. We need to emphasize that our approach is not a simple layered architecture as considered in software engineering because of the recursive aspect that takes place at all levels like in, for instance, standard network modeling. It is also worth noting that some aspects (such as filtering, publish/subscribe, and unexpected event definition and handling) are orthogonal to our approach and that they concern all layers. In addition, issues such as scalability and performance are not relevant to this conceptual framework. However, existing technologies to address scalability and performance fit in our model.

With our framework we aim to help researchers and developers to structure and direct their work more precisely along the various functionalities on differ-

ent levels of event-based systems. The presented architectural framework allows categorizing existing technologies and supports constructing complex systems from technological building blocks in the framework. We aim to contribute to establishing best practices and more standardization in the design of event-based systems.

Acknowledgements. We wish to thank Chris Switzer for his valuable help on the last version of this paper.

References

1. ACM Intl. Conf. on Distributed Event-Based Systems (DEBS). URL: http://www.informatik.uni-trier.de/ ley/db/conf/debs/ as of February 2010.
2. Complex Event Processing - Applications, products, research, and developments. URL: http://complexevents.com/events-workshop/, as of February2010.
3. D. J. Abadi, Y. Ahmad, M. Balazinska, U. Cetintemel, M. Cherniack, J. H. Hwang, W. Lindner, A. Maskey, A. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, and S. Zdonik. The design of the Borealis stream processing engine. In *Intl. Conf. on Innovative Data Systems Research (CIDR)*, pages 277–289, 2005.
4. D. J. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, C. Erwin, E. Galvez, M. Hatoun, A. Maskey, A. Rasin, A. Singer, M. Stonebraker, N. Tatbul, Y. Xing, R. Yan, and S. Zdonik. Aurora: a data stream management system. In *Proc. of the Intl. ACM Conf. on Management of Data (SIGMOD)*, pages 666–666, New York, NY, USA, 2003. ACM.
5. Y. Ahmad, J. Jannotti, E. Zgolinski, and S. Zdonik. Network awareness in internet-scale stream processing. *IEEE Data Engineering Bulletin*, 28:63–69, 2005.
6. J. Allen. Time and time again: The many ways to represent time. *International Journal of Intelligent Systems*, 6:341–355, 1991.
7. A. Arasu, B. Babcock, S. Babu, J. Cieslewicz, M. Datar, K. Ito, R. Motwani, U. Srivastava, and J. Widom. Stream: The stanford data stream management system. Technical report, Department of Computer Science, Stanford University, 2004. http://dbpubs.stanford.edu:8090/pub/2004-20.
8. A. Arasu, S. Babu, and J. Widom. The CQL continuous query language: Semantic foundations and query execution. *The VLDB Journal*, 15(2):121–142, 2006.
9. L. Brenna, J. Gehrke, M. Hong, and D. Johansen. Distributed event stream processing with non-deterministic finite automata. In *DEBS '09: Proceedings of the Third ACM International Conference on Distributed Event-Based Systems*, pages 1–12, New York, NY, USA, 2009. ACM.
10. J. Brusey, C. Floerkemeier, M. Harrison, and M. Fletcher. Reasoning about uncertainty in location identification with RFID. Workshop on Reasoning with Uncertainty in Robotics (in conjonction with IJCAI), 2003.
11. J. Carlson and B. Lisper. An event detection algebra for reactive systems. In *Proc. of the 4th ACM Intl. Conf. on Embedded Software (EMSOFT)*, New York, NY, USA, 2004. ACM.
12. A. Carzaniga. Architectures for an event notification service scalable to wide-area networks. PhD thesis, Politecnico di Milano, Milano, Italy, 1998.

13. A. Carzaniga, D. S. Rosenblum, and A. L. Wolf. Interfaces and algorithms for a wide-area event notification service. Technical Report CU-CS-888-99, Department of Computer Science, University of Colorado, October 1999.
14. A. Carzaniga, D. S. Rosenblum, and A. L. Wolf. Design and evaluation of a wide-area event notification service. *ACM Trans. Comput. Syst.*, 19(3):332–383, 2001.
15. S. Chakravarthy and Q. Jiang. *Stream Data Processing: A Quality of Service Perspective*. Springer Verlag, 2009.
16. S. Chakravarthy, V. Krishnaprasad, E. Anwar, and S.-K. Kim. Composite events for active databases: Semantics, contexts and detection. In *Proc. of the Intl. Conf. on Very Large Data Bases (VLDB)*, pages 606–617, San Francisco, CA, USA, 1994. Morgan Kaufmann Publishers Inc.
17. S. Chakravarthy and D. Mishra. SNOOP: An Expressive Event Specification Language for Active Databases. *Journal of Data and Knowledge Engineering*, 14(1), 1994.
18. S. Chandrasekaran, O. Cooper, A.Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. Madden, V. Raman, F. Reiss, and M. A. Shah. Telegraphcq: Continuous dataflow processing for an uncertain world. In *Intl. Conf. on Innovative Data Systems Research (CIDR)*, 2003.
19. S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. R. Madden, F. Reiss, and M. A. Shah. TelegraphCQ: Continuous dataflow processing. In *Proc. of the Intl. ACM Conf. on Management of Data (SIGMOD)*, pages 668–668, New York, NY, USA, 2003. ACM.
20. M. K. Chandy. Event-driven applications: Costs, benefits, and design approaches. Gartner Application Integration and Web Services Summit, 2006.
21. M. K. Chandy, O. Etzion, and R. von Ammon, editors. *07191 Abstracts Collection – Event Processing*, number 07191 in Dagstuhl Seminar Proceedings, Dagstuhl, Germany, 2007. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany.
22. J. Chen, D. J. DeWitt, F. Tian, and Y. Wang. NiagaraCQ: a scalable continuous query system for Internet databases. In *Proc. of the Intl. ACM Conf. on Management of Data (SIGMOD)*, pages 379–390, New York, NY, USA, 2000. ACM.
23. U. Dayal, B. Blaustein, A. Buchmann, U. Chakravarthy, M. Hsu, R. Ledin, D. McCarthy, A. Rosenthal, S. Sarin, M. J. Carey, M. Livny, and R. Jauhari. The HiPAC project: combining active databases and timing constraints. *ACM SIGMOD Record*, 17(1):51–70, 1988.
24. A. Demers, J. Gehrke, and P. Biswanath. Cayuga: A general purpose event monitoring system. In *Proc. of the Intl. Conf. on Innovative Data Systems Research (CIDR)*, pages 412–422, 2007.
25. EPCglobal. EPC Information Services (EPCIS) Version 1.01 Specification, September 2007. http://www.epcglobalinc.org/standards/epcis/, 2007.
26. EPTS. Event Processing Technical Society. available at http://www.ep-ts.com.
27. L. Fiege, G. Mühl, and F. C. Gärtner. A modular approach to build structured event-based systems. In *SAC '02: Proc. of the 2002 ACM symposium on Applied computing*, pages 385–392, New York, NY, USA, 2002. ACM.
28. L. Fiege, G. Muhl, and P. R. Pietzuch. *Distributed Event-based Systems*. Springer-Verlag, Berlin, Heidelberg, New York, June 2006.
29. A. Galton and J.-C. Augusto. Two approaches to event definition. In *In Proc. of the Intl. Conf. on Database and Expert Systems Applications (DEXA)*. Springer-Verlag, 2002.

30. M. R. Garey and D. S. Johnson. The rectilinear steiner tree problem is NP complete. *SIAM Journal of Applied Mathematics*, 32:826–834, 1977.
31. S. Gatziu and K. R. Dittrich. SAMOS: An Active Object-Oriented Database System. *IEEE Quarterly Bulletin on Data Engineering, Special Issue on Active Databases*, 15(1-4):23–26, 1992.
32. S. Gatziu and K. R. Dittrich. Detecting composite events in active database systems using petri nets. Proc. of the 4th Intl. workshop on research issues in data engineering: active database systems, Houston, 14-15 Feb 1994, 1994.
33. N. H. Gehani, H. V. Jagadish, and O. Shmueli. Composite event specification in active databases: Model & implementation. In *Proceedings of the VLDB*, 1992.
34. A. C. Gilbert, Y. Kotidis, S. Muthukrishnan, and M. Strauss. Surfing wavelets on streams: One-pass summaries for approximate aggregate queries. In *Proc. od the Intl. Conf. on Very Large Data Bases*, pages 79–88, San Francisco, CA, USA, 2001. Morgan Kaufmann Publishers.
35. R. E. Gruber, B. Krishnamurthy, and E. Panagos. Highlevel constructs in the ready event notification system. In P. Guedes and J. Bacon, editors, *8th ACM SIGOPS European Workshop*, New York, N.Y, 1998. ACM.
36. A. K. Gupta and D. Suciu. Stream processing of XPath queries with predicates. In *Proc. of the Intl. ACM Conf. on Management of Data (SIGMOD)*, pages 419–430, New York, NY, USA, 2003. ACM.
37. D. Gyllstrom, E. Wu, H. J. Chae, Y. Diao, P. Stahlberg, and G. Anderson. SASE Complex event processing over streams. In *Proc. of the Intl. Conf. on Innovative Data Systems Research (CIDR)*, 2007.
38. A. Hinze, K. Sachs, and A. Buchmann. Event-based applications and enabling technologies. In *Proc. of the Intl. Conf. on Distributed Event-Based Systems (DEBS)*.
39. A. Hinze and A. Voisard. EVA: An EVent Algebra supporting adaptivity and collaboration in event-based systems. Technical Report TR-09-006, International Computer Science Institute (ICSI), 2009. available at http://www.icsi.berkeley.edu/cgi-bin/pubs/publication.pl?ID=002680.
40. H.-A. Jacobsen. Publish/subscribe. In Ling Liu and M. Tamer Özsu, editors, *Encyclopedia of Database Systems*, pages 2208–2211. Springer US, 2009.
41. L. Gürgen L., C. Labbé, C. Roncancio, and V. Olive. Sstream: A model for representing sensor data and sensor queries. In *Intl. Conf. on Intelligent Systems And Computing: Theory And Applications (ISYC)*, 2006.
42. D. C. Luckham. *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems*. Addison-Wesley Longman Publishing Co., Inc., Boston,, 2001.
43. D. C. Luckham and B. Frasca. Complex event processing in distributed systems. Technical report, Stanford University, 1998.
44. D. C. Luckham and R. Schulte. Event processing glossary - version v1.1. available at http://www.ep-ts.com/component/.
45. S. Madden and M. J. Franklin. Fjording the stream: An architecture for queries over streaming sensor data. In *ICDE '02: Proceedings of the 18th International Conference on Data Engineering*, 2002.
46. S. R. Madden, Michael J. Franklin, Joseph M. Hellerstein, and W. Hong. TinyDB: An acquisitional query processing system for sensor networks. *ACM Transactions on Database Systems (TODS)*, 30(1):122–173, 2005.
47. R. Motwani, J. Widom, A. Arasu, B. Babcock, S. Babu, M. Datar, G. Singh Manku, C. Olston, J. Rosenstein, and R. Varma. Query processing, approximation, and resource management in a data stream management system. In *Proc. of the Intl. Conf. on Innovative Data Systems Research (CIDR)*, 2003.

48. G. Muehl, L. Fiege, and P. R. Pietzuch. *Distributed Event-based Systems*. Springer Verlag, Berlin/Heidelberg/New York, 2006.
49. G. Mühl. Generic constraints for content-based publish/subscribe. In *Proc. of the 6th Intl. Conference on Cooperative Information Systems (CoopIS), volume 2172 of LNCS*, pages 211–225. Springer-Verlag, 2001.
50. G. Mühl, L. Fiege, and A.P. Buchmann. Filter similarities in content-based publish/subscribe systems. In *Proc. of the Intl. Conference on Architecture of Computing Systems (ARCS)*, pages 224–240, New York/Heidelberg/Berlin, 2002. Springer-Verlag.
51. C. Olston, J. Jiang, and J. Widom. Adaptive filters for continuous queries over distributed data streams. In *Proc. of the Intl. ACM Conf. on Management of Data (SIGMOD)*, pages 563–574, 2003.
52. C. Pairot, P. García López, and A. F. Gómez-Skarmeta. Dermi: A decentralized peer-to-peer event-based object middleware. In *ICDCS*, pages 236–243, 2004.
53. A. Paschke and P. Vincent. A reference architecture for event processing (short paper). In *Proc. of the Intl. Conf. on Distributed Event-Based Systems (DEBS)*, pages 1–4, New York, NY, USA, 2009. ACM.
54. F. Peng and S. Chawathe. XPath queries on streaming data. In *Proc. of the Intl. ACM Conf. on Management of Data (SIGMOD)*, pages 431–442, New York, NY, USA, 2003. ACM.
55. P. R. Pietzuch. Hermes: A scalable event-based middleware. Ph.D. thesis, Computer Laboratory, Queens' College, University of Cambridge, 2004.
56. P. R. Pietzuch, J. Ledlie, J. Shneidman, M. Roussopoulos, M. Welsh, and M. Seltzer. Network-aware operator placement for stream-processing systems. In *ICDE '06: Proc. of the 22nd Intl. Conf. on Data Engineering*, page 49, New York, NY, USA, 2006. IEEE Computer Society.
57. P. R. Pietzuch, B. Shand, and J. Bacon. Composite event detection as a generic middleware extension. *IEEE Network*, 18(1):44–55, 2004.
58. J. Riecke, O. Etzion, F. Bry, M. Eckert, and A. Paschke. Event processing languages. Tutorial at the Intl. Conf. on Distributed Event-based Systems (DEBS) 2009.
59. R. Sherafat and H.-A. Jacobsen. Reliable and highly available distributed publish/subscribe service. In *28th IEEE International Symposium on Reliable Distributed Systems*, 2009.
60. U. Srivastava, K. Munagala, and J. Widom. Operator placement for in-network stream query processing. In *Proc. of the Intl. ACM symposium on Principles of Database Systems (PODS)*, pages 250–258, New York, NY, USA, 2005. ACM.
61. U. Srivastava and J. Widom. Flexible time management in data stream systems. In *Proc. of the Intl. ACM symposium on Principles of Database Systems (PODS)*, pages 263–274, New York, NY, USA, 2004. ACM.
62. W.W. Terpstra, S. Behnel, L. Fiege, A. Zeidler, and A. P. Buchmann. A peer-to-peer approach to content-based publish/subscribe. In *Proc. of the 2nd Intl. workshop on Distributed Event-Based Systems (DEBS)*, pages 1–8, New York, NY, USA, 2003. ACM.
63. N. Thaper, S. Guha, P. Indyk, and N. Koudas. Dynamic multidimensional histograms. In *Proc. of the Intl. ACM Conf. on Management of Data (SIGMOD)*, pages 428–439, New York, NY, USA, 2002. ACM.
64. S. Urban, I. Biswas, and S. W. Dietrich. Filtering features for a composite event definition language. In *Intl. Symposium on Applications and the Internet (SAINT)*. IEEE Computer Society, 2006.

65. A. Voisard and H. Schweppe. Abstraction and decomposition in interoperable gis. *Intl. Journal of Geographical Information Science*, 12(4):315–333, 1998.
66. E. Wu, Y. Diao, and S. Rizvi. High-performance complex event processing over streams. In *Proc. of the Intl. ACM Conf. on Management of Data (SIGMOD)*, pages 407–418, New York, NY, USA, 2006. ACM.
67. Y. Xing, S. Zdonik, and J.-H. Hwang. Dynamic load distribution in the Borealis stream processor. In *Proc. of the 21st Intl. Conference on Data Engineering*, pages 791–802, New York, NY, USA, 2005. IEEE Computer Society.
68. L. Xue, L. Jing, S. Quan, and Z. Weicai. Time To Live: Temporal Management of Large-Scale RFID Applications. Tech. report SSE-2008-02, School of Information Technology and Electrical Eng., Univ. of Queensland, 2008.
69. Y. Yao and J. Gehrke. The cougar approach to in-network query processing in sensor networks. In *Proc. of the Intl. ACM Conf. on Management of Data (SIGMOD)*, 2002.
70. R. Zhang and E. Unger. Event specification and detection. Technical Report 96-8, Kansas State University, June 1996. available at http://www.cis.ksu.edu/ schmidt/techreport/1996.list.html.
71. D. Zimmer and R. Unland. On the semantics of complex events in active database management systems. In *Proc. of the 15th Intl. Conf. on Data Engineering (ICDE)*, page 392, New York, NY, USA, 1999. IEEE Computer Society.
72. H. Zimmermann. OSI reference model–the ISO model of architecture for open systems interconnection. *IEEE Transactions on Communications*, 28(4):425–432, 1980.