Distributed Event Streams Composition

présentée le 20 février 2016
Université de Grenoble Alpes

pour l'obtention du grade de Docteur en Informatique
par

Orleant EPAL NJAMEN

acceptée sur proposition du jury:

Pr Homer SIMPSON, Rapporteur
Pr BART SIMPSON, Rapporteur
Pr MAGGI SIMPSON, Examinateur
Pr Christine COLLET, Directrice de thèse
Pr Vargas-Solar GENOVEVA, Co-directrice de thèse

Rien n'est plus insondable
que le système de motivations
derrière nos actions…
— Georg Christoph Lichtenberg

À vous que j'aime si fort…

Remerciements

Merci!

Abstract

The abstract...

Keywords: event stream processing, complex event processing, distributed systems, smart grids, Quality of service .

Résumé

Mots clefs : event stream processing, complex event processing, distributed systems, smart grids, Quality of service.

Contents

List of Figures

List of Tables

1 Introduction

Event-based systems have gained importance in many application domains, such as management and control systems, large-scale data dissemination, monitoring applications, autonomic computing, etc. The components of an event-based system communicate by producing and consuming events, where an event is the notification that a happening of interest has occurred [VC02]. An event service mediates producers and consumers enabling loosely coupled communication among them. Producers publish events to the service, and consumers express their interest in receiving certain types of events by issuing event subscriptions. A subscription is seen as a continuous query that allows consumers to obtain event notifications [DGP07, BKK07]. The service is then responsible for matching received events with subscriptions and conveying event notifications to all interested consumers [MFP06]. Several academic research and industrial systems have tackled the problem of event composition. Techniques such as complex pattern detection [GJS92, GD94, CC96, PSB04], event correlation [Duo96, YB05], event aggregation [Luc02], event mining [AS95, GTB02] and stream processing [WDR06, DGP07, BKK07], have been used for composing events. In some cases event composition is done on event histories (e.g. event mining) and in other cases it is done dynamically as events are produced (e.g. event aggregation and stream processing).

The extension of event models towards more flexible and Quality of Services (QoS) oriented event models requires an analysis of the semantics that should be given to the events, and of their associated processing strategies. This requires dissociating the modeling of events and the application design and, the proposal of methods that allow to define event types independently of the management issues (detection, production, notification). In addition, event composition varies according to the different application requirements, in other words, the semantics of composition operators and in the different production policies. Therefore, we need to adapt the event models to smart grid characteristics and requirements. We must define event types, composition operators with their associated semantics, and composition algorithms to produce QoS oriented complex events.

2 Background and related works

3 An Event Streams Composition Model

This chapter presents the building blocks to model an event composition system. It defines events, which are the entities manipulated by such a system. Then, it introduces event streams and stream based operations. Then it defines how to represent an event composition, with the related QoS dimensions.

Contents

3.1 Event, event type, event streams

A vast number of event management models and systems have been, and continues to be proposed. Several standardization efforts are being made to specify how entities can export the structure and data transported by events. Models proposed in the literature range from simple models which describe the notification of signals, to evolved models which take into account various policies to manage and compose events. Existing models have been defined in an ad hoc way, notably linked to the application context (active DBMS event models), or in a very general way in middleware (Java event service, MOMs). Of course, customizing solutions prevents systems to be affected with the overhead of an event model way too sophisticated for their needs. However, they are not adapted when the systems evolve, cooperate and scale, leading to a lack of adaptability and flexibility.

3.1.1 Definitions

Preamble

Let's assume a finite set of *domains*, each consisting of a possibly infinite set of values. In particular, we consider the domain $\mathbb{S}$ of strings, $\mathbb{B}$ of booleans ($\{true, false\}$), $\mathbb{Z}$ of integers, and $\mathbb{R}$ of real numbers. We also consider a domain $\mathbb{T}$ defined by the set $\mathbb{N} \bigcup \{0\}$, i.e. the set of natural numbers plus zero, which characterizes time values. These can be represented alternatively as *string, boolean, integer, real* and *time* respectively. In addition, we assume a set $\mathbb{A} = \{A_1, A_2, ...\} \subseteq \mathbb{S}$ of type names.

Complex value types

Complex value types are represented by lower-case letters with hats (e.g. $\hat{t}$) and are defined by a pair $A : def$, where A is the name of the type and def its definition. In order to enable access to both components we assume the functions *name* and *def*, which given a type will return the respective component of the type. For instance, for the type *Power : real, name(Power : real) = Power* whereas *def(Power : real) = real*. The set of all complex value types $\mathcal{T}$ is defined inductively as follows.

1. if D is a domain, then $A : D$ is an atomic type named A, where $A \in \mathbb{A}$;
2. if $\hat{t}$ is a type, then $A : \{\hat{t}\}$ is a set type named A;
3. if $\hat{t_1}, ..., \hat{t_n}$ are types with distinct names, then $A : \langle \hat{t_1}, ..., \hat{t_n} \rangle$ is a tuple type named A, and each $\hat{t_i}$ is an attribute type;
4. if $\hat{t_1}$ and $\hat{t_2}$ are types with distinct names, then $A : \hat{t_1} \oplus \hat{t_2}$ is the alternative type.

Every type $\hat{t} \in \mathcal{T}$ denotes a set of complex value instances $[\![\hat{t}]\!]$ which is defined inductively as follows.

1. For each atomic type $A : D$, $[\![A : D]\!] = \{d \mid d \in D\}$, where we assume the values of the domain D given;

2. for set types of the form $A : \{\hat{t}\}$, $[\![A : \{\hat{t}\}]\!] = \{S \mid S \in \mathscr{P}([\![\hat{t}]\!])\}$, where $\mathscr{P}$ denotes the power set;

3. for tuple types of the form $A : \langle \hat{t_1}, ..., \hat{t_n} \rangle$, $[\![A : \langle \hat{t_1}, ..., \hat{t_n} \rangle]\!] = \{\langle v_1, ..., v_n \rangle \mid : v_i \in [\![\hat{t_i}]\!] \bigwedge i \in [1..n]\}$;

4. for alternative type of the form $A : \hat{t_1} \oplus \hat{t_2}$, $[\![A : \hat{t_1} \oplus \hat{t_2}]\!] = \{v \mid v \in [\![\hat{t_1}]\!] \bigvee v \in [\![\hat{t_2}]\!]\}$

Access to components of complex value instances

We note $t \equiv \hat{t}$, to define an instance of a complex value type $\hat{t}$ named t. We define the function *val* to obtain the value v associated to an instance t. In addition, for tuple values t of the type $A : \langle \hat{t_1}, ..., \hat{t_n} \rangle$ such that $name(\hat{t_i}) = A_i$, we adopt the dot notation $t.A_i$ to access the instance of the attribute type $\hat{t_i}$ of t. In particular, if we have $t = \langle v_1, ..., v_n \rangle$, then $\forall i \in [1..n]$, $v_i = val(t.A_i)$. For example, $t \equiv Account : \langle id : string, balance : real \rangle$ denotes an instance t of the tuple type $Account : \langle id : string, balance : real \rangle$. In addition, if $t = \langle B14, 500 \rangle$ then, $val(t.id) = B14$ and $val(t.balance) = 500$.

For simplicity, we also adopt the notation t, to refer to the value $val(t)$ of an instance t. The difference between the value of the instance and the instance itself depends on the context. For example, for the tuple instance $t = \langle B14, 500 \rangle$ of the type $Account : \langle id : string, balance : real \rangle$, we have $t.id = B14$ and $t.balance = 500$.

3.1.2 Event

An event is something that happens at a particular place and time and that is particularly significant, interesting or unusual. In computing systems the notion of event has a major importance since it provides a powerful abstraction to model dynamic aspects of applications. For instance, events can represent state changes in databases; signals in message systems; changes of existing objects or the creation of new objects in object-oriented systems; or "real-world" events such as the departure or arrival of vehicles. In event-driven programming an event is a message that indicates a situation that happened, such as a keystroke or a mouse click. In process control an event is an occurrence that happened and that has been registered. Examples are a purchase order, an email confirmation of an airline reservation, a stock tick message that reports a stock trade.

The literature proposes different definitions of an event. For example, in [MsSS97] an event is a happening of interest, which occurs instantaneously at a specific time. Another definition given by [RW97] characterizes an event as the instantaneous effect of the termination of an invocation of an operation on an object. According to the first definition, an event exists because some entity is interested in it; the second one defines events independently of any

interested party. The second definition also subsumes an object model while the first one is neutral with respect to the model adopted for entities.

3.1.3 Event type

An event type is an expression that characterizes a class of significant facts (events) and the context under which they occur. Facts of the same nature are denoted by events that have the same type. An event model gives concepts and general structures used to represent event types.

According to the complexity of the event model, the event types are represented as sequences of strings [YBMM94], regular expressions [BGP+94] or as expressions of an event algebra [CM94, GD94, CC96]. Other models represent an event type as a collection of parameters or attributes, allowing the type itself to contains implicitly the content of the message. This model is useful when we need to reason about events content. For example, $MeterAlarm : \langle idMeter : string, voltage : real, current : real \rangle$ is an event type that represents a smart meter reading, where the current and voltage values observed are represented by attributes named *voltage* and *current*.
Event composition, or specifically event streams composition supports the idea of performing operations on events. Some of those operations needs to access the events content (see Section 3.1). For that reason, we represent an event type as a tuple type:
$EventType : \langle A_1 : D_1, ..., A_n : D_n \rangle$.
The definition of an event type includes the attributes presented in Table 3.1. The *producerID* attribute refers to the id of the entity that produced the event occurrence. The *detectionTime* attribute refers to the time at which the event occurrence has been detected by a source. The *productionTime* attribute refers to the time at which the event has been produced (as a result of processings on others events) by an event-processing unit. The *notificationTime* attribute refers to the time at which the event is notified to interested consumers. The *receptionTime* attribute refers to the time at which an interested consumer receives the event. In addition to those attributes that are common to all event types, each event type includes other attributes that capture the data that are particular to this event type. For example, a *MeterAlarm* event type is defined as $MeterAlarm : \langle$*producerID : string, detectionTime : time, productionTime : time, notificationTime : time, receptionTime : time, voltage : real, current : real*$\rangle$.
In addition to attributes presented in Table 3.1, the *MeterAlarm* event type includes the attributes *idMeter:string, voltage:real, current:real* that represents data that are specific to the MeterAlarm event type. In the rest of this document, we will refer to the attributes presented in Table 3.1 as *header attributes*. For writing simplicity, we will sometimes omit the header attributes in the description of event types and occurrences, when they are not relevant for the topic under consideration. For example, the $MeterAlarm$ event type previously defined can be simply defined as $MeterAlarm : \langle voltage : real, current : real \rangle$.

Name	Domain
producerID	string
detectionTime	time
productionTime	time
notificationTime	time
receptionTime	time

Table 3.1 – Event types attributes

Event occurrence

An event occurrence (or simply event) e is an instance of an event type *EventType*, that is $e \equiv EventType$. For example, considering the previously defined event type *MeterAlarm*, $e = \langle' meter5', 1, 1, 2, 4, 9, 1\rangle$ is an event instance produced by producer *meter5*, at time 1, notified at time 2, received at time 4, for which the voltage and current values are 9 and 1 respectively.

3.1.4 Event streams

An event stream is a continuous, append-only sequence of events $\{e_1, ..., e_n\}$. We note $Stream(s, T)$ the stream of events of type T generated by the source s. That is, $Stream(s, T) = \{e_1, ..., e_n \mid \forall e_i, e_i \equiv T \bigwedge e_i.producerID = s\}$. If S is a set of sources, then $\{\bigcup stream(s, T), s \in S\}$ defines a stream of events of type T, denoted $Stream(T)$.

3.2 Stream based composition operators

Event stream composition operators specify the types of operations that can be performed on event streams. An event stream composition operator op takes one or many input event streams of a given type, and produces an output stream of a given type (see Figure 3.1). In the following, we adopt the notation ES_i to denote the event stream of type E_i, that is $ES_i = Stream(E_i)$.

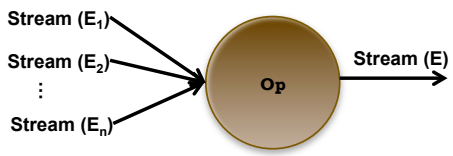

Figure 3.1 – Operator synopsis

Event stream composition operators considered here are : filter, logic operators, sequence, window operators, aggregation operators, selection operator and flatten.

3.2.1 Filter

The *filter* operator selects events in an input stream that satisfy a given predicate. The *filter* operator is denoted $filter_P(ES_i)$. It receives as parameters one input stream ES_i and a predicate P. The predicate P is defined according to the following rules:

- $A_i\ \theta\ v_i$ is a predicate, where

  - A_i is the name of an attribute of E_i, that is $\exists \hat{t_j}$ such that $\hat{t_j}$ is an attribute type of E_i and $name(\hat{t_j}) = A_i$

  - $\theta \in \{<, >, =, \leq, \geq\}$ is a comparison operator,

  - $v_i \in [\![\hat{t_j}]\!]$ is a value.

- if P_1 and P_2 are predicates, then

  - $P_1 \bigwedge P_2$ is a predicate. The symbol $\bigwedge$ denotes the conjunction.

  - $P_1 \bigvee P_2$ is a predicate. The symbol $\bigvee$ denotes the disjunction.

The output of the filter operator is an event stream having the same type as E_i. Its output is defined as $filter_P(ES_i) = \{e_j \mid e_j \in ES_i \bigwedge P(e_j) = true\}$.
For example, let's consider the event type $E_i = MeterMeasure$: $\langle$ *meterID : string, realPower : double*$\rangle$[1] and some event instances of type E_i, $e_1 = \langle$"$meter1$",7$\rangle$, $e_2 = \langle$"$meter1$",5$\rangle$, $e_3 = \langle$"$meter1$",4$\rangle$, $e_4 = \langle$"$meter1$",6$\rangle$.
Let's also consider the event stream ES_i =*Stream(MeterMeasure)* = $\{e_1, e_2, e_3, e_4, ...\}$.
Then, $filter_{realPower>5}(ES_i) = \{e_1, e_4, ...\}$. Events e_2 and e_3 have been filtered out by the filter operator since they don't satisfy the predicate "realPower > 5".

3.2.2 Disjunction

The disjunction operator merges the input streams into one output stream. The disjunction operator is denoted $OR(ES_1, ES_2, \ldots, ES_n)$. The output of the disjunction operator is an event stream of type $E_1 \oplus E_2 ... \oplus E_n$, defined as $OR(ES_1, ES_2 ..., ES_n) = \{e_j \mid \exists i \in [1..n],\ e_j \in ES_i\}$ $= \bigcup ES_i,\ i \in [1..n]$. The disjunction operator produces events in the output stream at the occurrence of events in any of the input streams $ES_i,\ i \in [1..n]$. For example, let's consider the event streams ES_1 and ES_2 in Figure 3.2. In this example, the output of $OR(ES_1, ES_2)$ is the sequence $\{e_{1,1}, e_{2,1}, e_{2,2}, e_{1,2}, ...\}$.

3.2.3 Conjunction

The conjunction operator is used to ensure that at least one event occured in all input streams. The conjunction operator is denoted $AND(ES_1, ES_2, \ldots, ES_n)$. The output of the disjunction

[1] We omitted the header attributes

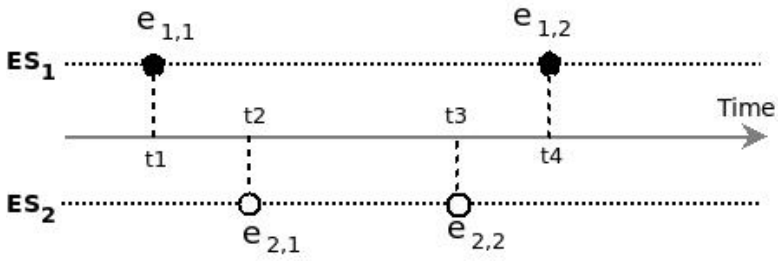

Figure 3.2 – Example situation 1. The time associated to events represents the time at which the events are received by the operator.

operator is an event stream of type $A : \langle context : \{E_1 \oplus E_2 \oplus ... \oplus E_n\}\rangle$[2], defined as follows: $e \in AND(ES_1,...,ES_2)$ iff $\forall i \in [1..n]$, $\exists e_{i,j} \mid e_{i,j} \in ES_i$. The *context* attribute is defined as follows: $e.context = \{e_{i,j} \mid \forall i, e_{i,j} \in ES_i\}$.
In other words, an event e is produced in the output stream if events $e_1, e_2, \ldots, e_n$ occur respectively in input streams $ES_1, ES_2, \ldots, ES_n$, regardless their occurrence order. The attribute *context* of the event e contains all events $e_1, e_2, \ldots, e_n$. For example, by considering again the event streams ES_1 and ES_2 in Figure 3.2, the output of $AND(ES_1, ES_2)$ will be:

- An event e_1 at time t2 with the event parameters $e_{1,1}$ and $e_{2,1}$, such that $e_1.context = \{e_{1,1}, e_{2,1}\}$
- An event e_2 at time t4 with the event parameters $e_{2,2}$ and $e_{1,2}$, such that $e_2.context = \{e_{2,2}, e_{1,2}\}$.

3.2.4 Sequence

The sequence operator captures precedence order of events in input streams. The sequence operator is denoted $SEQ(ES_1, ES_2, \ldots, ES_n)$, and receives two or more input event streams $ES_1, ES_2, \ldots ES_n$. The output of the disjunction operator is an event stream of type $A : \langle context : \{E_1 \oplus E_2 \oplus ... \oplus E_n\}\rangle$[3], defined as follows:
$e \in SEQ(ES_1,...,ES_2)$ iff $\exists\, e_1 \in ES_1,...,e_n \in ES_n \mid \forall i \in [1..n-1]$, $e_i.receptionTime < e_{i+1}.receptionTime$.
The *context* attribute contains the set of events e_i that occur. That is, $e.context = \{e_i \mid e_i$ occurred in $ES_i, \forall i \in [1..n]\}$.

In other words, the sequence operator produces an event e in output stream each time instances e_1 in ES_1, e_2 in ES_2,..., e_n in ES_n are received in the specified order. Then, sequence denotes that $\forall i$, occurrence e_i "is received before" occurrence e_{i+1}. The context attribute of the event e contains all events e_1 and e_2, ..., e_n. For example, let's consider the event streams

[2] We omitted the header attributes
[3] We omitted the header attributes

ES_1 and ES_2 in Figure 3.3, the output of $SEQ(ES_1, ES_2)$ will be:

- An event e_1 at time t2 with the event parameters $e_{1,1}$ and $e_{2,1}$, such that $e_1.context = \{e_{1,1}, e_{2,1}\}$;
- an event e_2 at time t5 with the event parameters $e_{1,2}$ and $e_{2,3}$, such that $e_2.context = \{e_{1,2}, e_{2,3}\}$

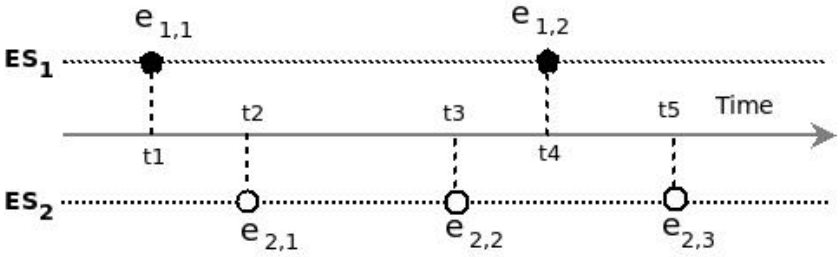

Figure 3.3 – Example situation 2. The time associated to events represents the time at which the events are received by the operator.

3.2.5 Window operators

Window operators partition an event stream into finite parts of the original event stream. Let's consider an input stream $Stream(E)$. We denote by $Stream_f(E)$, a finite part of $Stream(E)$. A window operator applied on $Stream(E)$ results in a stream of finite streams, which we denote $Stream(Stream_f(E)) = \{ES_{f,1}, ES_{f,2}, ES_{f,3}, ...\}$. The way each finite stream is constructed depends on the window specification, which can be *time-based* or *tuple-based*.

Time based windows

Time based windows define windows using time intervals.

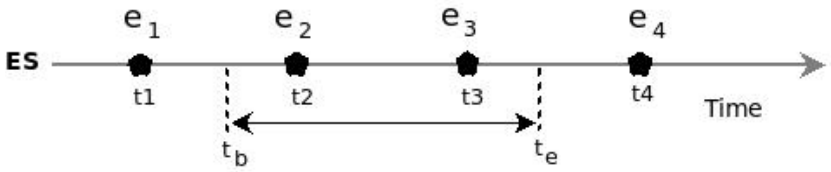

Figure 3.4 – Example situation 3. The time associated to events represents the time at which the events are received by the operator.

- **Fixed windows**: $win : within_{(t_b,t_e)}(ES)$. Defines a fixed time interval $[t_b, t_e]$. The output stream contains a single finite event stream $ES_{f,1}$ such that $\forall e \in ES, e \in ES_{f,1}$ iff $t_b \leq$

$e_i.receptionTime \leq t_e$. In the example depicted in Figure 3.4, the output of $win : within_{(t_b,t_e)}(ES)$ is the finite event stream $\{e_2, e_3\}$

- **Landmark windows**: $win : since_{t_b}(ES)$. Defines a fixed lower bound time t_b. The output stream is a sequence of finite event streams $ES_{f,i}, i \in \{1,2,\ldots,n\}$, such that each $ES_{f,i}$ contains events e_i received since the time lower bound t_b. That is, $\forall i, e \in ES_{f,i}$ iff $t_b \leq e_i.receptionTime$. In the example depicted in Figure 3.4, the output of $win : since_{t_b}(ES)$ is the finite event stream $\{e_2, e_3, e_4\}$

- **Sliding windows**: $win : sliding_{(t_w,t_s)}(ES)$. Defines a time duration t_w and a time span t_s. The output stream is a sequence of finite event streams $ES_{f,i}, i \in \{1,2,\ldots,n\}$, such that each $ES_{f,i}$ contains events from stream ES produced during last t_w time units. The finite event streams in the sequence are produced each t_s time unit. That is, if $ES_{f,i}$ is produced at time t, then $ES_{f,i+1}$ will be produced at time $t + t_s$. In the example depicted in Figure 3.5, the output of $win : sliding_{(t_w,t_s)}(ES)$ are the finite event streams $\{e_1, e_2\}$ and $\{e_4\}$.

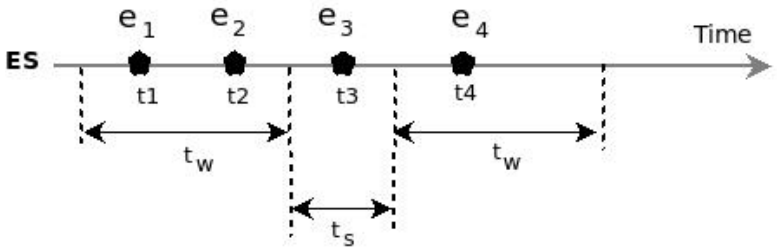

Figure 3.5 – Example situation 4. The time associated to events represents the time at which the events are received by the operator.

Size bounded windows

A size bounded window defines the number of events for each window. We distinguish between size fixed windows and sliding size fixed windows.

- **Fixed size windows**: $win : batch_{nb}(ES)$. Specifies a fixed size nb of each finite stream. The output stream is a sequence of finite event streams $ES_{f,i}, i \in \{1,2,\ldots,n\}$, such that each finite event stream $ES_{f,i}$ contains nb most recent events from ES, and are non-overlapping. If we consider the event stream $ES = \{e_1, e_2, e_3, e_4, e_5, e_6, \ldots\}$, $win : batch_3(ES)$ will result in finite event streams $\{ES_{f,1}, ES_{f,2}, ...\}$ such that $ES_{f,1} = \{e_1, e_2, e_3\}$, $ES_{f,2} = \{e_4, e_5, e_6\}$, and so on.

- **Moving fixed size windows**: $win : mbatch_{(nb,m)}(ES)$. Defines a fixed size nb of each finite stream, and a number of events m after which the window moves. The output stream is a sequence of finite event streams $ES_{f,i}, i \in \{1,2,\ldots,n\}$, such that each

$ES_{f,i}$ contains nb most recent events from ES. $ES_{f,i+1}$ is started after m events are received in $ES_{f,i}$ (moving windows). As result, an event instance may be part of many finite event streams. This is the case if $m \leq nb$. For example, if we consider windows of size nb=3 moving after each m = 2 events, that is $win : mbatch_{(3,2)}(ES)$, the event stream $ES = \{e_1, e_2, e_3, e_4, e_5, e_6, e_7, \ldots\}$ will be partitioned into finite event streams $\{ES_{f,1}, ES_{f,2}, ES_{f,3}, \ldots\}$ such that $ES_{f,1} = \{e_1, e_2, e_3\}$, $ES_{f,2} = \{e_3, e_4, e_5\}$, $ES_{f,3} = \{e_5, e_6, e_7\}$, and so on.

3.2.6 Aggregation operators

Applied on a stream of finite streams $Stream(Stream_f(E)) = \{ES_{f,1}, ES_{f,2}, ES_{f,3}, ...\}$, an aggregator operator $aggregate_{attr,\ aggrAttr}(Stream(Stream_f(E)))$ computes for each finite stream $ES_{f,i}, i \in \{1, 2, \ldots, n\}$, the aggregated value of the attribute *attr* over events occurrences in $ES_{f,i}$.
The attribute *attr* is the name of an attribute of E, that is $\exists \hat{t}_j$ such that $\hat{t}_j$ is an attribute type of E and $name(\hat{t}_j) = attr$.
The output is a event stream of type $A : \langle aggrAttr : D \rangle$[4] where $D = def(\hat{t}_j)$.
Aggregate operators include *max, min, count, avg* and *sum.*

- The max operator compute the maximum value of the attribute over the event instances in $ES_{f,i}$.
- The min operator compute the minimum value of the attribute over the event instances in $ES_{f,i}$.
- The $count$ operator compute the number of event instances in $ES_{f,i}$ with the specified attribute.
- The avg operator compute the average value of the attribute over the event instances in $ES_{f,i}$.
- The sum operator compute the sum of the attribute's values over the event instances in $ES_{f,i}$.

3.2.7 Selection operators

A selection operator takes as input a stream of finite streams $Stream(Stream_f(E)) = \{ES_{f,1}, ES_{f,2}, ...\}$ and produce as output an event stream of type E, with values $e_1, e_2, \ldots$ such that for all i, each e_i is a selection of a particular event from $ES_{f,i}$. The choice of that particular occurrence depends on the selection operator:

- **First occurrence**: $first(Stream(Stream_f(E)))$.

[4] We omitted the header attributes

For each finite stream $ES_{f,i}, i \in \{1,2,\ldots,n\}$, selects the first event occurrence. For example, if we apply the first operator to the stream of finite streams $\{\{e_1,e_2\}, \{e_3,e_4,e_5,e_6\}, \{e_7,e_8\},\ldots\}$, we obtain as output the stream $\{e_1,e_3,e_7,\ldots\}$.

- **Last occurrence**: $last(Stream(Stream_f(E)))$.
  For each finite stream $ES_{f,i}, i \in \{1,2,\ldots,n\}$, selects the last event occurrence. For example, if we apply the last operator to the stream of finite streams $\{\{e_1,e_2\}, \{e_3,e_4,e_5,e_6\}, \{e_7,e_8\},\ldots\}$, we obtain as output the stream $\{e_2,e_6,e_8,\ldots\}$.

3.2.8 Flatten

The flatten operator is denoted $flatten(Stream(Stream_f(E)))$. The *flatten* operator takes as input a stream of finite event streams $Stream(Stream_f(E)) = \{ES_{f,1}, ES_{f,2}, ES_{f,3}, ...\}$ and produces as output an event stream of type E, which contains the concatenation of events in $ES_{f,1}, ES_{f,2}, ES_{f,3}$, and so on. For example, if we consider the stream of finite streams $\{\{e_1\},\{e_2,e_3\},\{e_4,e_5,e_6\},\ldots\}$, the flatten operator produces the output stream $\{e_1,e,e_2,e_3,e,e_4,e_5,e_6,e,\ldots\}$ where e is a special "empty" event that indicates the end of each finite stream in the output stream. We assume that for all event type E, the empty event e satisfies $e \equiv E$.

3.3 Event streams composition

Stream based operators can be chained in order to produce complex event streams that capture particular situations. This is called event stream composition. An event stream composition is expressed by an event stream composition expression. An event stream composition expression is defined as:

- $A = Op(ES_1, ...ES_n)$ is an event stream composition expression named A, where Op denotes a stream based operator and $\forall i \in [1..n]$, ES_i denotes an event stream.
- $A = Op_1(Op_2(...(Op_n(A_1, ...A_m))...))$ where $\forall i \in [1..n]$, $\forall j \in [1..m]$, Op_i is a stream based operator and A_j is an event stream composition expression.

The event stream composition expression defines an event stream named A An event stream composition expression A defines an event stream with the same name A. For example, if we consider the event type $MeterMeasure$: ⟨ *meterID : string, realPower : double*⟩ and the event stream $ES = Stream(MeterMeasure)$, we can define an event stream *ComplexStream* that computes the aggregated real power of meter *'AMI100'* between time 10 and 50 as follows: $ComplexStream = avg_{realPower,\ avgP}$*(win:within(*$filter_{meterID='AMI100'}(ES)$*, 10, 50))*. The definition of the *ComplexStream* starts by filtering the event stream ES on the predicate *meterID='AMI100'*. Then, the result is used to compute a fixed windows (*win:within*) of events between time 10 and 50. The output stream, which is a finite stream is then aggregated on the attribute *realPower*. The aggregated value can be retrieved by accessing the *avgP* attribute of the event *e* in the *ComplexStream* output stream.

An event stream composition expression must be well formed. More precisely, the input of each stream based operator Op_i implicated in the event stream composition expression must be consistent with the operator definition. For example, the input of an aggregate operators is a stream of finite streams, which can be computed using windows operators as in the *ComplexStream* example. Moreover, the result of a windows operator, which is a stream of finite event streams, cannot be given directly as input to a filter operator as in the expression $FilteredStream = filter_P(win : sliding_{(t_w,t_s)}(ES))$. This is due to the fact that the filter operator takes as input an event stream, an not a stream of finite event streams. In order to be consistent with the filter operator definition, the *FilteredStream* expression can be written $FilteredStream = filter_P(flatten(win : sliding_{(t_w,t_s)}(ES)))$.

3.4 Conclusion

In this chapter, we presented a model for event streams composition. We focused on the definition of concepts that are manipulated by an event stream composition system. First, we presented the concepts of event, event types, event occurrence and event streams. Then, we presented operators applicable to event streams with their associated semantic. After that, we introduced event stream composition and event stream composition expressions, as mechanisms for combining stream based operators in order to produce complex event streams.

In the next chapter, we will go one step forward, by presenting how the presented model can be leveraged to define an approach for distributed event streams composition that deals with QoS.

A French Summary

Contents

A.1 Introduction

Bibliography

[BGP+94] M L Bailey, Burra Gopal, Michael Pagels, L L Peterson, and Prasenjit Sarkar. PATHFINDER: A Pattern-Based Packet Classifier. (JANUARY 1994):115–123, 1994.

[CC96] Christine Collet and T. Coupaye. Primitive and Composite Events in O2 Rules. In *Actes des 12ièmes Journées Bases de Données Avancées*, 1996.

[CM94] S. Chakravarthy and D. Mishra. Snoop: An expressive event specification language for active databases, 1994.

[GD94] S. Gatziu and K.R. Dittrich. Detecting composite events in active database systems using Petri nets. *Proceedings of IEEE International Workshop on Research Issues in Data Engineering: Active Databases Systems*, 1994.

[MsSS97] Masoud Mansouri-samani, Morris Sloman, and Morris Sloman. Gem - a generalised event monitoring language for distributed systems. *IEE/IOP/BCS Distributed Systems Engineering Journal*, 4, 1997.

[RW97] David S Rosenblum and Alexander L Wolf. A design framework for Internet-scale event observation and notification. *ACM SIGSOFT Software Engineering Notes*, 22(6):344–360, 1997.

[YBMM94] Masanobu Yuhara, Brian N Bershad, Chris Maeda, and J Eliot B Moss. Efficient Packet Demultiplexing for Multiple Endpoints and LargeMessages. In *WTEC'94: Proceedings of the USENIX Winter 1994 Technical Conference on USENIX Winter 1994 Technical Conference*, page 13, 1994.