Vargas-Solar, Genoveva; Bucciol, Paolo; Collet, Christine
On a Framework for Complex and ad hoc Event Management over Distributed Systems
Polibits, vol. 41, 2010
Instituto Politécnico Nacional
Distrito Federal, México

On a Framework for Complex and ad hoc Event Management over Distributed Systems

Genoveva Vargas-Solar, Paolo Bucciol, and Christine Collet

Abstract—Internet-based communications have amazingly evolved in recent years. As a consequence, the number – and complexity – of distributed systems which provide access to services and applications has dramatically increased. As long as these services have been extended to support an increasing number of communication media (voice, audio, video, ...) and systems, ad hoc communication protocols and methodologies have been designed and developed. Given the autonomy of available services and applications, distributed systems generally rely on event-based communications for integrating these resources. However, a general model for the management of event-based communications, suitable for complex and ad hoc event processing as well as for the generic publish/subscribe messaging paradigm, is still missing. This paper presents[1] a general and flexible event detection and processing framework which can be adapted based on specific requirements and situations. Within the framework, the main aspects of event management over distributed systems are treated, such as event definition, detection, production, notification and history management. Other aspects such as event composition, are also discussed. The goal of the paper is to provide a common paradigm for event-based communications, providing at the same time new advantages with respect to the existing standards such as composition, interoperability and dynamic adaptability.

Index Terms—Event management, modeling, distributed systems, interoperability, adaptability.

I. Introduction

THE "fully connected world" is perhaps the most remarkable step in the evolution of the communication system in the last century. Through the Internet, each user can virtually communicate *events* almost instantaneously with any other user. The concept of *event* is of major importance in the communications field, since it provides an enough general abstraction layer through which dynamic aspects of applications can be modeled. Events produced in the real world are converted into a sequence of bits, sent through one or more networks to reach their destination and then elaborated and presented following the consumer's indications. Events are well fitted to represent the dynamic aspects of applications

and distributed characteristics of systems: the sequence of their execution, the state of a given process or of a data structure, the communication between entities. Examples of events range from the evolution of data (*the price of Euro has varied in the United States*), to the change in the execution state of a given process (*a web page has been modified*), to the communication and interaction between its components (*a print request has been performed*).

A vast number of event management models and systems has been, and continues to be, proposed [1]–[3]. Several standardization efforts are being made to specify how entities can export the structure and contents of the events [4]. 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 the events. Existing models have been defined in an *ad hoc* way, notably linked to the utilization 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 heavy weight 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.

The event management framework proposed in this paper, based on an in-depth event characterization, is targeted to the definition of a common event management model.

Through the definition of the conceptual mechanisms and the general semantics for the integration of distributed systems and heterogeneous networks, the proposed model aims at providing modular, adaptable and extensible mechanisms, which are well adapted when building applications and systems.

A list of *dimensions* is proposed [2], that characterize the *detection*, the *production*, the *notification* of events and the *history management* respectively. The proposed dimensions allow to: (i) highlight the aspects inherent to the management of events; (ii) propose a meta-model of event management to give a general characterization of management models independently of their applicative context [1].

The remainder of this paper is organized as follows. The generic structure of event-based systems is presented in Section II, while more specific aspects concerning events are described in Section III (event definition), Section IV

(event detection), Section V (event production), Section VI (event notification), and Section VII (event history). A generic event service which supports event management in distributed systems is introduced in Section VIII. Finally, conclusions and future work are presented in Section IX.

II. EVENT-BASED SYSTEMS

Event-based communication, based on a cause-effect principle, provides the basis for anonymous and asynchronous communication. The *event*, the cause, represents the change in the state of a system which leads to the production of a message. On the other hand, the *reaction*, the effect, corresponds to a set of reactions within the system, which react to the cause and can lead to the production of more events. This principle allows to model the evolution of a distributed system based on events asynchronously detected.

An event is modeled as an object of a specific type. The type is specified by the programmer, as, for instance, an object of type "event". The production environment is represented by the object attributes and methods, however finer categorization may be defined by the programmer.

An event channel is an object which allows multiple producers to communicate with multiple consumers in an asynchronous way. The event channel, at the same time producer and consumer of events, can notify changes related to a certain object. In this sense, it acts as an intermediary between objects which have been modified and an object interested (or involved) in such changes. When a change has taken place, an event can be notified to all the objects interested. In this vision, the underlying communication channel and network (e.g. multi-hop network) is transparent to the event channel.

Producers and consumers can communicate through the event channel by means of the *push and pull* model. The producer notifies some events to the channel; then, the channel notifies the events to the consumers. The consumer consumes the events through the event channel, which, in turn, detects them from the producer. The decoupled communication between producers and consumers is highly relevant in contexts where consumers would not be able to receive and interpret the messages, which is, for instance, the case of sensor networks. When a sensor needs to be awake to receive the events in real time (*on* status), it stays continuously connected to the network and thus wastes significant amounts of energy. This communicating strategy is not feasible in the common situation where sensors have strong computational and power constraints [5]. Event channels are therefore of deep importance to sensor networks, in which the consumers are given the possibility to receive information (events) asynchronously. Moreover, event channels determine how to propagate the changes between producers and consumers. For instance, an event channel determines the persistence of an event: it is the channel which decides for which period to hold an event, to which to send it and when. The producers generate

events without having to know the identity of consumers and vice versa.

Common event models are considered an actual challenge also in the field of multimedia stream management. especially when the considered system is distributed over heterogeneous networks ("There is the need of querying and event processing architectures, simple schemes for sensor networks are not designed for multimedia sensors" [6]). Among the most interesting new approaches to multimedia event management, it is worth noting the six-dimensional "5W+1H" approach, derived from the journalism [7], [8] and particularly related to the emerging field of multimedia event management.

Other recent challenges in the field of multimedia and stream event management and modeling include multimedia over sensor networks [6], which introduce interesting dimensions related to multimedia requirements such as class type, data type, and bandwidth. In [9], event management for RFID sensors is discussed. Events are characterized as {event time, ID of RFID tags, ID of RFID readers} tuples, and XML-based communication is foreseen. A similar approach is discussed in [10]. However, this is not applicable to other types of nodes (e.g. sensors), where the low communication bandwidth foresees more efficient communication modes.
In [11] several aspects of multimedia applications are considered (structural, temporal, informational, experiential, spatial and causal). A common event model is foreseen to provide interoperability to multimedia applications of different areas, such as multimedia presentations and programming frameworks. Chang *et al.* [12] consider multimedia elements as a primary data type (*micon*), and define apposite operators like Ψ (conversion between media formats).

In addition, in practical situations, events produced by sensors such as wireless motes and RFID readers, are not significant enough for consumers. They must be combined or aggregated to produce meaningful information. By combining and aggregating events either from multiple producers, or from a single one during a given period of time, a limited set of events describing meaningful situations may be notified to consumers. Therefore, academic research and industrial systems have tackled the problem of event composition. Techniques such as complex patterns detection [13]–[16], event correlation [17], [18], event aggregation [19], event mining [20], [21] and stream processing [22]–[24], have been used for composing events. In some cases event composition is done on event logs (e.g. data mining) and in other cases it is done dynamically as events are produced (e.g. event aggregation and stream processing). Nevertheless, to the best of our knowledge, there is no approach that integrates different composition techniques. Yet, pervasive and ubiquitous computing, network and environment observation, require to observe behavior patterns that can be obtained by aggregating and mining statically and dynamically huge event logs or histories.

III. Event Definition

The definition of an event type is described by a dimension and a domain. We consider the word "dimension" as *a parameter or coordinate variable assigned to such a property*[3]. The identified dimensions are not independent, but instead they are organized in layers and are cross-referenced to characterize the events. Table 6 summarizes the main event dimensions identified in the work for characterizing events' definition (Dimensions 1-7). Dimensions 1-3 are related to the event representation: if the event is represented by a type or not, the structure of the type, if it has a production environment or not. Dimensions 4-6 characterize the types of operators which can be associated to the event types to represent composite event types. Finally, dimension 7 (net effect) considers the net effect by means of: (i) inverse types, and (ii) an algebra of production environments.

The *net effect of a sequence of operations* represents the balance of those operations after the end of the sequence. If, for instance, a sequence of operations creates an entity, and destroys it afterwards, the net effect will be considered as zero. If two entities are created, a and b, and afterwards a is destroyed, the net effect will be the creation of b.

The definition of event types is strongly related to time, taking into account the temporal nature of the notion of event. Indeed, the definition of an event type often integrates concepts like interval and occurrence instant. Certain types can also represent other temporal dimensions such as duration, dates, periods, etc. In general, event models are based on the time models inherited from the context in which they have been designed: programming languages, data models, etc. The following characterizes the dimensions to be considered for defining events.

Dimensions 8-12 (ref. Table 2) characterize a time model. The occurrence instant of an event is represented as a point on a timeline which can be discrete or continuous (dimension 8). Granularity and temporal types, defined in dimensions 9 and 11 respectively, can be used to define event types. Granularity also characterizes the operations which may be executed on the temporal types, and which can be also used to describe event types – like, for example, *5 minutes after an user connection*: *BeginConnection + 5 minutes with delta(login:string, instantconnection:date)*. The possibility to have conversion functions is foreseen in dimension 10, while dimension 12 describes the types of temporal operators available.

The most basic concept regarding time is the *timeline*. Abstractly speaking, a timeline is a pair $(D, <_T)$ composed by a finit set of *chronons* [4] D [25] and a binary relation $<_T$, which defines a complete and linear order over D. From the event management point of view, a timeline serves to model a discretized position in the production of a succession of events linearly ordered. This notion is particularly important, since the types conceptually represent occurrences produced within a timeline. The characteristics of the timeline allow then to choose, for instance, specific ordering algorithms, but also to specify the temporal relations between events, such as: an event e_1 was produced after e_2, or an event e_1 was produced between $9:00$ and $17:00$.

We refer to *granularity* as one partition of the set of chronons of a given timeline and its convex subsets, named *particles*, and to *minimal granularity* as the granularity obtained by dividing a timeline in singletons. Weeks, months and years correspond to *granularities*. The partial order relation *finer than* $\prec$ defines a hierarchical structure on a same timeline, which for instance allows to define that the granularity *seconds* is finer than *hours*.

The $\prec$ relation also allows to convert particles belonging to different granularities by means of conversion functions like *approximation*, which allows to rough guess a particle of a granularity G_1 by means of a particle G_2 which contains it (*zoom in*), and *expansion*, which allows to associate a set of granularities G_1 to each particle of granularity G_2 (*zoom out*). Interested readers can refer to [26] for further details.

Starting from the concept of granularity, a set of types which get involved directly in the definition of event types has been identified:

- An *instant* is a point within a timeline which can be represented by an integer, when a discrete time representation is adopted;
- A *duration* is a number of particles used as a distance measure between two instants, to allow the expression of movements in time with respect to a given instant. In general, it is characterized by a positive integer (its measure) and by a granularity, like *4 seconds*;
- An *interval* is represented by the bias between two instants, or by an instant (the lower bound of the interval) and its duration. Given that the lower and upper bounds of an interval are both of the same granularity, the interval can be represented by means of a granularity and two positive integers (the positions of the bounds).

The temporal types of the programming languages and the query languages are generally provided with operators. In our opinion, the four basic following operators on time models which should be included in any software implementation are:

- *selectors* of the maximum/minimum instant and of the duration of a set of instants with the same granularity,
- *order relations* on the instants ($<, >, =$),
- *arithmetic operators* between instants and durations like addition and subtraction of a duration to an instant, or between two durations, and
- *conversion operators* between two temporal values observed with different granularity levels. Interested readers can refer to [27] for more details.

---

[3]Merriam-Webster English dictionary. The geometric meaning of dimension, for space definition (e.g. *a three-dimensional space*), will not be considered here.

[4]A chronon is a proposed quantum of time in the Caldirola's discret time theory.

TABLE I
DIMENSIONS OF EVENT TYPES.

	Dimension	Domain
1	Event	{*with type, without type*}
2	Event type	{*string, expression, object*}
3	Production environment	{*yes, no*}
4	Operator types	{*selection, algebraic, temporal*}
5	Validity interval	{*interval, period, none*}
6	Filtering	{*regular expressions, predicates*}
7	Net effect	{*inverse events, algebra of production environments, none*}

TABLE II
DIMENSIONS CHARACTERIZING THE TIME MODEL.

	Dimension	Domain
8	Time	{*discrete, continuous*}
9	Granularity	{*day, month, year, hour, minute, second*}
10	Conversion function	{*yes, no*}
11	Temporal types	{*instant, interval, duration*}
12	Temporal operators	{*comparison, selection, join*}

IV. DETECTION

DETECTION is the process through which an event is recognized and associated to a point in time named *instant of occurrence*. The events may be observed by a process external and independent from the producer, or signaled by the producer itself.

The detection is characterized by the conditions in which the events are observed, and can be modeled by dimensions from 13 to 16 described in Table 3 and explained in the remainder of this section. The production unit is the interval during which producer can observe events (dimension 13). The observation point (dimension 14) can be located at the operation start or end points. The interaction protocols used to retrieve them, finally, are described by type (dimension 15) and detection mode (dimension 16).

The PRODUCTION UNIT identifies the interval during which the events can be detected. More precisely, it specifies the interval within the execution of a producer in which events are produced. The interval can be defined by the duration of the execution of a program, a transaction, an application, an user connection.

For example, the transaction is the production unit in most of centralized active DBMS [28]. Few active DBMS allow event detection mechanisms without transactions. The so-called external events, that is, the event which do not represent operations on the base and aren't inevitably produced inside transactions, are also detected in the context of a transaction. Distributed active systems [29]–[31] allow detection of events coming from different DBMS and applications. In this case, the events are observed by the detectors within the transactions. The detectors are synchronized by a global detection mechanism, which builds a global view of events produced within different transactions – and without transactions.

It is possible to associate an implicit production unit to a set of producers. In this case, event detection is active as long as there is any producer subscribed, even if there are no consumers. The Microsoft event service [32] defines the production unit as implicit and bound to the duration of the execution of the producer objects, while the Java event service [33] bounds the production unit to the duration of the execution of the producer objects, "reducing" the management to a *multicast* notify mechanism: the producer objects notify events to the subscribed consumer objects. The production unit in streams in sensor networks and also managed by Data Stream Management Systems (DSMS) is determined by explicit time intervals.

Event detection mechanisms face a granularity issue when the processes to be observed have a duration and the events which represent them are instantaneous. To this concern, certain systems distinguish between physical and logical events. A PHYSICAL EVENT is the instance which has been detected, while a LOGICAL EVENT is its conceptual representation (expressed by an event type).

We refer to the physical events which have been detected as an *occurrence of an event type*. A logical event which represents an observed operation can be split in two physical events detected repetively *before* and *after* the operation, according to the observation point of the detection process. For instance, the update operation on the *balance* variable of an entity with type ACCOUNT may be represented by an event type associated to an observation point named "point_obs": $<$ *point_obs $> UpdateAccount$ with delta(accountnumber:integer, newbalance:real, oldbalance:real)*.

TABLE III
DIMENSIONS OF EVENT DETECTION.

	Dimension	Domain
13	Production unit	{*duration of execution of the producer, transaction, connection, application*}
14	Observation point	{*before, after*}
15	Detection protocol	{*pull, push* }
16	Detection mode	{*synchronous, asynchronous*}

The operation can be handled by associating it to a time interval $[t_0, t_1]$ where t_0 correspond to the operation start and t_1 to its end. An event can be observed at t_0, corresponding to the transition:

update operation inactive → update operation running[5],

and at t_1, corresponding to the transition:

update operation running → update operation finished[6].

Both cases refer to the same event, but detected in two different instants. The modifier *before* and *after* allow to state the events observation point with respect to the operation executed within a production unit.

The DETECTION PROTOCOL identifies the way of interaction with the producer in order to retrieve the events. In general, events are detected with a protocol of type *push* if the producer explicitly reports the events. In case of type *pull*, the detection mechanism queries or observes the producer to retrieve the events. The choice of one of the two protocols depends on the characteristics of the producers.

The DETECTION MODE relates the event detection mechanism to the execution of the producer. In the *synchronous* detection mode, the producer stops its execution to signal the event. On the contrary, the *asynchronous* detection mode assumes that the producers report the events to the detection mechanism without having to interrupt their execution.

In general, asynchronous detection is achieved by means of a shared memory space. The *asynchronous pull* detection mode assumes that there is a monitoring mechanism implemented in the producer which observes its execution and stores the events in a shared memory space accessible by the detection mechanism. In case of *synchronous pull* detection, the execution of the producer can be interrupted instead.

V. PRODUCTION

The PRODUCTION process corresponds to the time stamping process of a detected event – taking into account the instant at which the event occurs – and to its insertion within an event history. Production is based on read and write access to an history of produced events, as well as on the computation

of the *production instant* of the events. The dimensions of the production specify policies for ordering and composing detected events (see Dimensions 17 - 20 in Table 4). Such policies determine how to time stamp events, and which events of the history should be used to produce composite events.

The TIME STAMPING process is the process through which events are labeled with information regarding their instant of occurrence. This process is based on the notion of *clock*, that is, a function $C(e)$ which associates an event e to its instant of occurrence I_{occ}. A *time stamp* specifies the position of an event on a timeline. The structure of time stamps varies depending on the observation of events with respect to a local or global reference.

In a centralized system, time can be described as a completely ordered sequence of points [7], where instants correspond to readings of the system local clock. Let e_1 and e_2 be two events, detected respectively at the instants $I_{occ}(e_1)$ and $I_{occ}(e_2)$. It is then possible to establish a total or partial order between the two events by ordering their instants of occurrence. Consequently, e_1 is produced before, after or at the same time than e_2.

In a distributed system, events are generally produced at points in time identified by different clocks. According to the observation point, the relative order between two events can vary depending on the observer's position. As a consequence, when events are produced by multiple producers and observed by multiple consumers, it is necessary to choose a reference clock in order to have a global perception of the events. The time point which an event is associated is then "associated" with a point of the reference clock, taking into account the drift of producer and reference clock with respect to an universal global reference point.

The GLOBAL TIME g_{t_k} of the instant I_{local_k}, read in a local clock k, is described by a point of the Gregorian calendar (*Universal Time Coordinated*) truncated to a global granularity g_g:

$$g_{t_k}(I_{local_k}) = TRUNC_{g_g}(clock_k(I_{local_k})),$$

where $TRUNC()$ is a rounding function like $round()$, $ceil()$, $floor()$ depending on the application context.

The "global" time stamp of a event allows to determine its production instant in a global timeline, knowing its position on another temporal reference called *local* with respect to a

---

[5]The transition is represented by the following event:
< *Before* > *UpdateAccount with delta(*
accountnumber:integer, newbalance:real, oldbalance:real).

[6]The transition is represented by the following event:
< *After* > *UpdateAccount with delta(*
accountnumber:integer, newbalance:real, oldbalance:real).

[7]Declaring that the time points are completely ordered implies that, for any pair of points t_1 and t_2, the temporal relation between them is either $t_1 < t_2$, $t_1 = t_2$ or $t_1 > t_2$.

TABLE IV
DIMENSIONS OF EVENT PRODUCTION.

	Dimension	Domain
17	Time stamping	$\{ < I_{occ}global >, < site, I_{occ}local >, < I_{occ}local, site, I_{occ}global > \}$
18	Granularity	$\{instance,\ set\ of\ the\ same\ type,\ set\ of\ different\ type\}$
19	Consumer range	$\{local, global\}$
20	Production mode	$\{continuous,\ recent,\ chronological,\ cumulative\}$

given *site*. The time stamp of an event $T(e)$ is thus a tuple of the form $< I_{occ}(e)$, *site*, $I_{occ_{global}}(e) >$, where $I_{occ}(e)$ is the instant of occurrence with respect to the local clock of the producer *site*, and $I_{occ_{global}}(e)$ is the *global instant of occurrence* with respect to a reference clock. For example, the time stamp of an event e produced in the site k may be of the following form:

$T(e)=<23991548127,\ k,\ ('19/10/95',\ 2:32:27.32)>$

The time stamping of a composite event is determined by the most recent component event. In a distributed context, the notion of "most recent" is not unique, that is, multiple component events exist which are virtually produced at the same time and which may contribute to triggering a composite event. In this case, all events contribute to determine the time stamp of the composite event. The time stamping process is summarized in Equation 1.

Given a set of time stamps ES and a time stamp $st \in ES$, the maximum can be defined as:

$st = Max(ES) \iff (\nexists\ st_1 \in ES, st < st_1).$

A set of maximum time stamps is therefore defined as:

$Max(ES) = \{st\ \in ES : st\ is\ a\ maximum\ of\ ES\}.$

The time stamp $T(e)$ of a composite event is the set $Max(ES)$, where ES is the set of time stamps of the component events. The production instant, as a function of the semantics of composition operators, can then be computed by using the set $Max(ES)$. For example, given the following join and sequence semantics:

$(E_1 \wedge E_2)(st) = \exists st_1, st2\,[E_1(st1) \wedge E_2(st2)] \wedge$
$[st = Max(st_1, st_2)]$ *(intersection)*
$(E_1; E_2)(st) = \exists st_1\,[E_1(st1) \wedge E_2(st2)] \wedge$
$(st_1 < st)$ *(strict sequence)*

To establish an order between events, it is necessary to compare their time stamps. The ordering procedure concerns the management of the event history, which will be discussed in Section VII.

It is possible to distinguish between events of different granularities according to the number of occurrences they are composed of. In general, the following two PRODUCTION GRANULARITIES can be identified:

– *instance*: an event is produced every time that an occurrence of event type is detected; and
– *set*: an event is produced at the moment of the detection and composition of a set of events of the same type, or of different type.

For example, let us consider an event of type *creation of a new bank account* detected at the moment of an insertion in a relation *Accounts*. With an *instance-oriented* granularity, the event is produced every time that the operation "a tuple is inserted into the *Account* relation" is executed. On the contrary, what happens if N bank accounts are created? Would it be needed to produce an event *creation of a new bank account* for each insertion or just one event which represents the *creation of N bank accounts*? The choice is determined based on what the consumer wants to observe and according to the application context.

A set of granularities *of the same type* allows to produce events which group N occurrences of the same type. For example, all occurrences of type *creation of a new bank account* produced in the context of a single transaction or all the purchases made by a client within the last month.
A set of granularities *of different types* allows to produce composite events by combining the events with operators such as join, disjoint, sequence, etc. For example, *creation of a new bank account followed by a deposit of more than 1000 EUR*:

CreateAccount with delta(accountnumber:string,
owner:string, balance:real); [8]
UpdateAccount with delta(accountnumber:string,
oldbalance:real, newbalance:real)
where
 CreateAccount.accountnumber=
  UpdateAccount.accountnumber
 and $oldbalance - newbalance > 1000$

In the two cases, it is necessary to specify the production conditions of the component events. For example, the component events should be produced by the same producer or within the same production unit. Such aspects are related to the semantics of the composition operators, but also to the *construction of the production environment*, as explained in the following.

The production process begins with the detection of basic events[9]. When a basic event is detected, it is necessary to verify if its occurrence initializes or triggers the production of a composite event. The production of such event depends on the occurrence of its component events and on the order with which the occurrences are produced.

---

[8]The operator ";" represents the *sequence* operator.
[9]We talk about *detection* of basic events and *production* of composite events.

$$Max\left(T(e_1), T(e_2)\right) = \begin{cases} T(e_1) & T(e_2) < T(e_1) \\ T(e_2) & T(e_2) < T(e_1) \\ T(e_1) \cup T(e_2) & T(e_2)\ and\ T(e_1)\ cannot\ be\ compared \end{cases} \tag{1}$$

The production of composite events is based on a *production mechanism*. This mechanism "knows" the structure of the composite event and the order with which its component events should have taken place for the composite event to be triggered. Everytime a simple event is detected, it is notified to the production mechanism. If the production can move forward, a new production state is derived. A certain *final state* indicates the triggering of a *composite event*.

The production of composite events has been deeply studied within the active databases domain [34], [35]. To date, most of the production mechanisms of composite events are based on the evaluation of *finite state automata*, *Petri nets* and *graphs*.

Let the reader remind that all events convey information which differentiates them and informs on the conditions under which they are produced: production instant, producer identifier, real parameters inherent to its type (see *production environment* within Section II. The *construction of the production environment* of an event is determined by the production granularity chosen to produce it. For example, let us consider an history $h = \{e_{12}, e_{13}, e_{24}, e_{15}, e_{26}, e_{27}, \ldots\}$, containing also occurrences of type *operation made on a bank account*:

E_1 = Deposit with delta(accountnumber:integer, amount:real)

The type E_1 conveys information concerning the account number and the amount of the operation which has been made. With a production granularity of *instances*, three events e_{12}, e_{13}, e_{15} will be produced and the production context of each contains the instant of production as well as the real parameter associated to its type, that are, bank account number and the amount of the operation which has been made.

When the construction of the production environment is by *set*, it is necessary to specify which events – produced previously and available in the history – participate in its construction. For instance, an event representing *n deposits made on the same bank account between* $[9 : 00, 17 : 00]$:
E_1 = Deposit with n × delta(accountnumber:integer,
  amount:real)
within [9:00,17:00] where same(accountnumber)

The CONSUMPTION SCOPE defines the selection policies of the events belonging to the history used for building the production environment of an event e_i. The scope of consumption can be:

– *local* with respect to (i) a producer, for example, when events of the same producer are selected; (ii) a specific set of producers, for example, when events produced by processes running on the same server are selected; (iii) to a time interval, for example, when only events

produced within the same production unit are used (e.g. a transaction); or
– *global*, if all instances present in the history are used to build the production environment of an event (e.g. when the event contains information regarding bank transactions effectuated during the day).

The PRODUCTION MODE determines the consumption protocol of events to build composite events. It indicates the combinations of primitive events that participate in the composition of an event and clarifies the semantics of composite event types. The notion of *production mode* of events has been introduced by Snoop [36] with the name of *parameter context*.

Four production modes have been proposed in the database domain. *(i) Continuous:* all occurrences which time stamp the begin of an interval of a composite event type are considered as initiator events of composite events. *(ii) Recent:* only the most recent events are used to trigger occurrences of E. *(iii) Chronological:* the occurrences of events are considered in their chronological order of appearance. The component events are used with a "FIFO"–type strategy. *(iv) Cumulative:* when an occurrence of E is recognized, the context to which it is associated includes — cumulates — the parameters of all occurrences of events which participate to its construction. For instance, NAOS [35] implements the *continuous* mode, Sentinel [37] implements all the four modes, Chimera [38] supports the *recent* mode, SAMOS [39] implements the *chronological* mode.

VI. NOTIFICATION

The NOTIFICATION process deals with the notification of events to the consumers. The notification of an event can be made at specific instants in relation to their instant of production and considering temporal constraints. The events can also be filtered before being notified. The dimensions of the notification, corresponding to dimensions 21-27 in Table 5, characterize the aspects to be taken into account for the notification of events to the consumers. Such aspects are determined by the specific consumer needs in terms of information (validity interval, instant of notification, history management, communication protocol), but also by the autonomy and the isolation needs of the producers (granularity and range of selection, visibility of events in the history).

The VALIDITY INTERVAL (dimension 21), specifies an observation window on events belonging to the history. This interval allows to specify in which period of time a consumer is interested on events of a specific type. For example, the following expression:

TABLE V
DIMENSIONS OF EVENT NOTIFICATION.

	Dimension	Domain
21	Validity interval	{*implicit, explicit*}
22	Notification instant	{*immediate with [Δ]C [Comp], immediate without C, [Δ] differed wrt. an event*}
23	Selection granularity	{*instance, set*}
24	Selection range	{*local, interval, global*}
25	Visibility	{*local, global*}
26	Notification protocol	{*pull, push*}
27	Notification mode	{*synchronous, asynchronous*}

after UpdateAccount with delta(accountnumber:integer, oldbalance:real, newbalance:real), [9:00,17:00]

allows to focus on an *update on a bank account executed between* 9 : 00 *and* 17 : 00. Only events which take place in the active production unit(s) (a transaction, a program, etc.) in the interval [9 : 00, 17 : 00] are considered. If the validity interval and the production unit correspond, the events are then relevant during the execution of the transaction (production unit by default) in which they are produced and are not visible outside such interval. Treating separately the event production unit (on the detection side) and the validity interval (on the notification side) contributes to the flexibility of the event notification mechanism, while allowing the consumers to decide the periods during which they want to observe events.

The NOTIFICATION INSTANT (dimension 22) specifies the instant when the notification mechanism must retrieve the events from the history to notify them to the consumers. The events are delivered at different instants according to the degree of reliability of the conveyed information, the notification rate required by the consumers, and so on. It is possible to notify them:

– *immediately after the production*;
– at a *precise instant* with respect to the production of another event, for instance at the end of the stock market session, every two minutes, on 10/09/00 at 14:30; or
– with respect to a *latency time* defined as follows: it exists a constant Δ such as, if the instant of production of an event e_i is t, then the notification mechanism sends the event after $t + \Delta$.

It is also possible to associate a *confirmation* to the event notification, to allow the validation – or invalidation – of the notified occurrences. For example, the event e_1 "an operation of purchase of actions has been executed" can be confirmed by the event e_2 "authorization confirmed by the bank". It is possible to specify a temporal constraint (*latency time*) associated to the notification of the confirmation event. For example, e_2 should be notified with a *15 minutes delay*. If the constraints are never verified, it can also be defined a *compensation event* notified instead of the confirmation event. An instant of notification is associated to the compensation event too.

The EVENT SELECTION process specifies the policies to be used to choose the events belonging to the event history when it is necessary to notify them to the consumer. These policies are essentially determined by the consumers' needs and the characteristics of the producers (for instance, autonomy), but also by the characteristics of the applications common to producers and consumers.

The SELECTION GRANULARITY (dimension 23) describes the selection criteria of history events:
(i) *instance–oriented*: only the last occurrence of an event is notified;
(ii) *set–oriented*: all instances available in the history are notified.

The SELECTION SCOPE (dimensions 24, 25) describes the visibility of history events with respect to the consumers. Two scopes of selection are possible based on applications' needs:

– *local*: each consumer is granted access to a history events subset. There can be two different criteria: selection of events produced within a specified time interval (e.g., selection of the events detected in the same production unit (*intra–production unit*)), or considering a content–based filtering;
– *global*: events produced within other executions (*inter–production unit*) are selected, that is, beyond the limitations of applications and transactions. For example, the events produced within a given transaction are visible after the validation or before it.

The *inter–production unit* approach poses issues which still have to be treated, like: (i) what to do of history events which production unit has terminated?, and (ii) until which moment are they visible to the consumers?

The dimensions of notification (26 and 27) characterize also the communication protocol and mode used to notify events to the consumers. The NOTIFICATION PROTOCOL describes the way in which the notification mechanism interacts with the consumer during the event notification process, while the NOTIFICATION MODE define the way the operations are executed. Asynchronous notification assumes consumers and notification mechanism exchange events through a shared memory space. The *pull asynchronous* notification mode assumes that the notification mechanism stores the events in a memory space which may be queried by the consumer to retrieve the events. In case of a *push synchronous* notification, the notification mechanism must be able to stop the consumer(s) execution.

VII. History

An EVENT HISTORY is an ordered set of instances of events. The event history plays a fundamental role in the event production and notification. The management policies of the event history, summarized by means of dimensions 28-31 in Table 6 determine which events have to be inserted in the history and for how long, and when to insert and delete them. The management of the event history specifies then the insertion, selection and history update policies.

EVENTS INSERTION (dimension 28) in the event history is determined by an ordering function which allows to determine the position of an event (just occurred) in the history. According to specific algorithms like the precedence model $2g_g$ proposed by [40], we assume that it is always possible to determine an ordering (total or partial) between history events, based on comparison of timestamps. Let the reader refer to Appendix A for more details concerning timestamps comparison.

The HISTORY SELECTION is determined by the events *visibility* adopted for the production of events and by the *selection scope events* adopted for the notification. The *visibility* of events adopted for the production describes the policies to be used to consume the events part of the history during the construction of the production environment of an event e_i. With a policy:

- *intra–occurrence*, the event e_i is produced and ignored in the construction of the production context of other occurrences;

- *inter–occurrence*, after having been produced, the event e_i is kept visible for the construction of production environments of other occurrences.

The UPDATE of the history (dimension 29) is performed in the following cases:

- *Invalidation*: An event stored in the history can be invalidated by the production of a second event. The *net effect* allows to determine this situation. To be able to compute the net effect of an event of type E, its inverse type E^{-i}, as well as the net effect computation rules (defined by the model of event types) should initially be known. The invalidation of events affects the event notification process. In particular, the cancellation of events already notified but that wait for a confirmation. If the event is cancelled before the production of the confirmation event, strategies to notify such situation should be planned.
- *Expiration*: The consumers determine a validity interval for the various event types. When such deadline expires, event can be deleted from the history if they have been received by all consumers. A "sophysticated" system where the consumers grant different validity intervals to the same event type can be imagined. In this case, the definition of views within the history can be interesting.

- *Cancellation*: The production of an event can be cancelled by a particular situation. For example, when events are produced within transactions, their production may be cancelled if the transaction fails. In this case, it is necessary to delete from the history all cancelled events.
- *Explicit invalidation*: Instances of events can be deleted as a consequence of explicit requests formulated by clients or users [10].

A *transient* event has a duration equal to zero. On the contrary, a *persistent* event has a longer duration. So, for example, an event may persist as long as the control stream by which it has been generated exists, the object from which has been produced persists or another event still has to be produced. The PERSISTENCE (dimension 30) describes the time interval during which the events are kept in the history. Four policies can be adopted. An event e_i is kept:

1) *until its notification*: e_i is kept in the history until it has been notified to all its consumers, then it is deleted;
2) *during the validity interval*: events are kept in the history until the end of their validity interval;
3) *for the production unit*: events do not survive the execution of their producer. For example, event are kept in the history until the end of the transaction, as long as a program is running or an user is connected. As mentioned earlier, in many active DBMS event models the production unit (transaction) corresponds to the validity interval. In this case, events are kept until the end of the transaction;
4) *until the production of the next event of the same type*: a new occurrence of an event causes the deletion of the previous occurrence. In this case, two situations are possible: the new event can add up to the production environment of its predecessor (at the condition that it refers to the same data), or can ignore it.

Taking into account the net effect has a few implications on the persistence policies, since in the case of a persistence policy *until the notification*, an event e_i can be cancelled by another event of "inverse type" between the time of its production and the moment when the notification has taken place. In the other cases, an event of inverse type can always cancel e_i.

The events describe changes, transitional by nature and which can be assimilated to a transition between two states, generated within a system and which determine its evolution. PERMANENCE aims at making the evolutions permanent by storing in the disk the history of corresponding events, in validation points explicitly set by the application programmers.

A number of applications exist, such as workflow or data mining applications, for which the permanence of events could be useful — as when the so–called "durative" events have to be detected. For example, the detection of an event like *5 days after the creation of a bank account* requires the storage of the event *creation of a bank account* for at least five

---

[10]Which is undoubtedly an effective method to tamper with event history.

TABLE VI
DIMENSIONS OF EVENT HISTORY.

		Dimension	Domain
28		Insertion	{*ordering function*}
29		Updating	{*invalidation, expiration, cancellation, explicit cancellation, none*}
30		Persistence	{*validity interval, production unit, until a more recent e_i is produced* }
31		Permanence	{*validation point, explicit, none*}

consecutive days. The migration of processes can also need the permanence of events. A sequence of events reflects indeed the execution state of one or more processes. It is potentially desirable to store this information, migrate the process, then recover the events by means of which it can be relaunched with its last state.

VIII. EVENT SERVICE FRAMEWORK

An event service is a mediator in event-based systems enabling loosely coupled communication among producers and consumers. 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 [23], [24]. The service is then responsible for matching received events with subscriptions and conveying event notifications to all interested consumers [41]. Within our framework, the challenge is to design and implement an event service that implements event composition by querying distributed histories ensuring scalability and low latency.

We propose a generic event service that provides support for managing events in distributed systems. The service architecture consists of cooperative and distributed event detectors, which receive, process and notify events. The event service detects the occurrence of primitive and composite events and consequently notifies all the applications or components that have declared their interest in reacting to such events. Two main detector types are provided by the service: *Primitive Event Detectors*, which gather events from producers; and *Composite Event Detectors*, which process and produce events from multiple, simpler detectors.

Event Detectors are connected forming an event composition network. Events obtained from producers are propagated through the composition network and incrementally processed at each Composite Event Detector involved. Event processing functions can be separately configured for each Composite Event Detector. The most important configurable functions include event composition, correlation, aggregation, filtering according to the dimensions described in the previous sections.

Event Detectors are managed by the ***Service Manager*** component. They are dynamically created and configured in base on advertisements and subscriptions from producers and consumers respectively. Then, the Service Manager manages event advertisements and subscriptions. It maintains a list of all the detectors of the system and the event types (primitive or composite) that each one of them manages (receives and notifies).

The Service Manager implements the `DataCollector` interface, receiving advertisements and subscriptions from producers and consumers respectively, in order to create and configure Event Detectors. The Service Manager uses `EDManagement` interfaces in order to (re-)configure Event Detectors.

We propose a distributed event composition approach, done with respect to subscriptions managed as continuous queries, where results can also be used for further event compositions. According to the type of event composition strategy (i.e., aggregation, mining, pattern look up or discovery), event composition results can be notified as data flows or as discrete results. Event composition is done with respect to events stemming from distributed producers and ordered with respect to a timestamp computed with respect to a global time line. Provided that our event approach aims at combining different strategies that can enable dynamic and postmortem event composition, we assume that different and distributed event histories can be used for detecting composite event patterns. For example combining a history of events representing network connections with another history that collects events on the access to a given address in the Internet for determining the behavior of users once they are connected to the network.
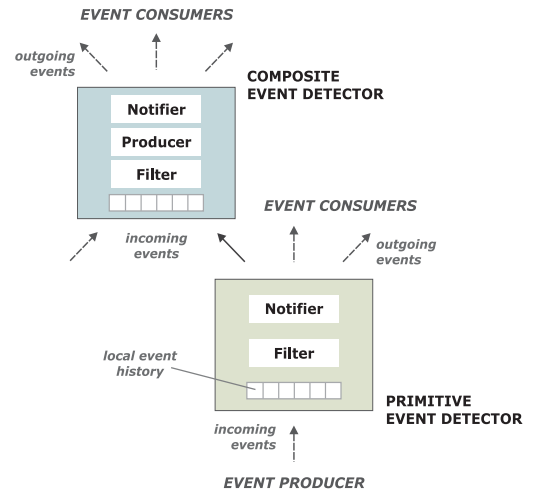

Fig. 1. *Primitive and Composite Event Detectors*

Therefore, both histories must be analyzed for relating the connection of a user to the sites he/she visits at different moments of the day.

The implementation of the event service extends the implementation of the Composite Probes framework [42]. The prototype implementation is based on Julia [43], a Java implementation of the Fractal component model [44], and on additional Fractal utilities, including Fractal ADL [45]. The service prototype implements Primitive and Composite Detectors. The service of communication is implemented via JORAM [46], a JMS implementation. Additionally, Fractal RMI and Fractal JMX should be used for the distribution and managing of the Fractal components. The event service implementation uses Fractal bindings for interconnections among Event Detectors. Using bindings allows both local communication inter-components via direct method calls, and distributed communication based on Fractal RMI [47]. In addition, maintaining the binding principle provides a true architecture among Event Detectors.

IX. CONCLUSIONS

A general framework for processing events and thereby supporting the communication among producers and consumers has been presented in the paper. In our approach we consider that event based communication must be adapted according to the kind of producers and consumers. For example, continuous detection/notification of events in sensor networks is not the same as detecting and notifying events concerning RSS flows of a web site or monitoring middleware infrastructures for supporting business services. Our framework and its associated event service can be personalized thanks to a general meta-model that provides dimensions for programming ad-hoc event management policies. The paper also describes our implementation experience of an event service that implements the meta-model and that is based on composite probes [42] and that can be configured for detecting, composing and notifying events.

We are currently developing a general event composition framework that can act in a completely distributed way and support dynamic event composition and event mining on post-mortem event histories. We are particularly interested in supporting monitoring for cloud-computing computing and business services, middleware and computer services monitoring, and for transmission of multimedia content over ad-hoc networks.

ACKNOWLEDGMENTS

REFERENCES

[1] *ADEES: An Adaptable and Extensible Event Based Infrastructure*. London, UK: Springer-Verlag, 2002.

[2] Y. Yao and J. Gehrke, "The cougar approach to in-network query processing in sensor networks," vol. 31, no. 3.   New York, NY, USA: ACM, 2002, pp. 9–18.

[3] D. J. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik, "Aurora: a new model and architecture for data stream management," *The VLDB Journal*, vol. 12, no. 2, pp. 120–139, 2003.

[4] D. D. et al. (2010, February) Web services event descriptions (ws-eventdescriptions), w3c working draft. [Online]. Available: http://www.w3.org/TR/2010/WD-ws-event-descriptions-20100209/

[5] I. A. Ismail, I. F. Akyildiz, and I. H. Kasimoglu, "Wireless sensor and actor networks: Research challenges," 2004.

[6] T. M. I. F. Akyildiz and K. R. Chowdury, "Wireless multimedia sensor networks: A survey," *IEEE Wireless Communications*, vol. 14, no. 6, pp. 32–39, December 2007.

[7] L. Xie, H. Sundaram, and M. Campbell, "Event mining in multimedia streams," *Proceedings of the IEEE*, vol. 96, no. 4, pp. 623–647, 2008. [Online]. Available: http://dx.doi.org/10.1109/JPROC.2008.916362

[8] X.-j. Wang, S. Mamadgi, A. Thekdi, A. Kelliher, and H. Sundaram, "Eventory – an event based media repository," in *ICSC '07: Proceedings of the International Conference on Semantic Computing*.   Washington, DC, USA: IEEE Computer Society, 2007, pp. 95–104.

[9] J. Xu, W. Cheng, W. Liu, and W. Xu, "Xml based rfid event management framework," nov. 2006, pp. 1 –4.

[10] S. Bose and L. Fegaras, "Data stream management for historical xml data," in *SIGMOD '04: Proceedings of the 2004 ACM SIGMOD international conference on Management of data*.   New York, NY, USA: ACM, 2004, pp. 239–250.

[11] U. Westermann and R. Jain, "Toward a common event model for multimedia applications," *IEEE MultiMedia*, vol. 14, no. 1, pp. 19–29, 2007.

[12] S.-K. Chang, L. Zhao, S. Guirguis, and R. Kulkarni, "A computation-oriented multimedia data streams model for content-based information retrieval," *Multimedia Tools Appl.*, vol. 46, no. 2-3, pp. 399–423, 2010.

[13] O. S. N.H. Gehani, H.V. Jagadish, "Composite event specification in active databases: model & implementation," in *18th*, Septembre 1992.

[14] S. Gatziu and K. R. Dittrich, "Detecting composite events in active database systems using Petri nets," in *4th International Workshop on Research Issues in Data Engineering: Active Database Systems*, 1994.

[15] C. Collet and T. Coupaye, "Composite Events in NAOS," in *7th(DEXA'96)*, Zurich - Switzerland, 9-13 Septembre 1996.

[16] P. P, S. B, and B. J, "Composite event detection as a generic middleware extension," *Network*, vol. 18, no. 1, pp. 44–55, 2004.

[17] D. C, "Alarm driven supervision for telecommunication networks. on line chronicle recognition," *Annales des Telecommunications*, pp. 501–508, 1996.

[18] E. Yoneki and J. Bacon, "Unified semantics for event correlation over time and space in hybrid network environments," in *OTM Conferences*, 2005.

[19] D. Luckham, *The Power of Events: An Introduction to Complex Event Processing in Distributed Systems*.   Addison Wesley Professional, 2002.

[20] R. Agrawal and R. Srikant, "Mining sequential patterns," in *PROC 11th International Conference on Data Engineering*, IEEE, Ed., Taiwan, 1995.

[21] A. Giordana, P. Terenziani, and M. Botta, "Recognizing and Discovering Complex Events in Sequences," in *ISMIS 02: Proceedings of the 13th International Symposium on Foundations of Intelligent Systems*, London, 2002.

[22] E. Wu, Y. Diao, and S. Rizvi, "High-Performance Complex Event Processing over Streams," in *SIGMOD*, 2006.

[23] A. J. Demers, J. Gehrke, B. Panda, M. Riedewald, V. Sharma, and W. M. White, "Cayuga: A General Purpose Event Monitoring System," in *CIDR*, 2007.

[24] M. Balazinska, Y. Kwon, N. Kuchta, and D. Lee, "Moirae: History-Enhanced Monitoring," in *CIDR*, 2007.

[25] P. Caldirola, "The introduction of the chronon in the electron theory and a charged-lepton mass formula," *Lettere Al Nuovo Cimento (1971-1985)*, vol. 27, no. 8, pp. 225–228, February 1980.

[26] G. Iqbal-A, Y. Leontiev, M.-T. Ozsu, and D. Szafron, "Modeling temporal primitives: Back to basics," 1997.

[27] M. Dummett, *Elements of intuitionism*, 2nd ed., ser. Oxford logic guides 39.   Clarendon Press, 2000.

[28] N. W. Paton, *Active Rules for Databases*. Springer Verlag, 1998.

[29] S. Berchtold, D. A. Keim, and H.-P. Kriegel, "Using extended feature objects for partial similarity retrieval," *The VLDB Journal*, vol. 6, no. 4, pp. 333–348, 1997.

[30] Y. Li, M. Potkonjak, and W. Wolf, "Real-time operating systems for embedded computing," in *International Conference on Computer Design*, 1997, pp. 388–392.

[31] J. pieter Katoen and L. Lambert, "Pomsets for message sequence charts," in *1st Workshop of the SDL Forum Society on SDL and MSC, SAM98*, 1998, pp. 291–300.

[32] D. A. Menascé, "Mom vs. rpc: Communication models for distributed applications," *IEEE Internet Computing*, vol. 9, no. 2, pp. 90–93, 2005.

[33] D. Flanagan, *Java In A Nutshell, 5th Edition*. O'Reilly Media, Inc., 2005.

[34] S. Chakravarthy, V. Krishnaprasad, Z. Tamizuddin, and R. H. Badani, "ECA Rule Integration into an OODBMS: Architecture and Implementation," University of Florida, Department of Computer and Information Sciences, Technical Report UF-CIS-TR-94-023, Mai 1994.

[35] C. Collet, "NAOS," in *In Active Rules for Databases*, N. W. Paton, Ed. Springer Verlag, 1998.

[36] S. Chakravarthy and D. Mishra, "Snoop: An Expressive Event Specification Language For Active Databases," University of Florida, Gainesville, Tech. Rep. UF-CIS-TR-93-007, Mars 1993.

[37] S. Chakravarthy, "Sentinel: an object-oriented dbms with event-based rules," in *SIGMOD '97: Proceedings of the 1997 ACM SIGMOD international conference on Management of data*. New York, NY, USA: ACM, 1997, pp. 572–575.

[38] S. Ceri and R. Manthey, "Consolidated Specification of Chimera," IDEA Esprit Project, Politecnico di Milano, Milano - Italy, Tech. Rep. IDEA.DE.2P.006.01, 1993.

[39] S. Gatziu, H. Fritschi, and A. Vaduca, "SAMOS an Active Object-Oriented Database System: Manual," Zurich University, Tech. Rep. Nr 96.02, Février 1996.

[40] S. Schwiderski, "Monitoring the behaviour of distributed systems," Tech. Rep., 1996.

[41] M. G, F. L, and P. P, *Distributed Event-Based Systems*. Springer-Verlag, 2006.

[42] "Composite probes," http://forge.objectweb.org/projects/lewys.

[43] "Julia," http://fractal.objectweb.org/julia/.

[44] "Fractal," http://fractal.objectweb.org/.

[45] "Fractal," http://fractal.objectweb.org/fractaladl/.

[46] "Joram," http://joram.objectweb.org/.

[47] "Fractal," http://fractal.objectweb.org/fractalrmi/.

[48] S. Chakravarthy and S. Yang, "Formal semantics of composite events in distributed environments," 1999.

APPENDIX

Let the reader remind that a timestamp $T(e)$ of an event e_i is a tuple $< I_{occ}(e_i), site, I_{occ_{global}}(e_i) >$ [11]. Yang and Chakravarty define in [48] the temporal ordering relationship between the timestamps of primitive events as follows:

1) e_i before e_j:

$T(e_1) < T(e_2) \iff$

$(T(e_1).site = T(e_2).site) \land (I_{occ}(e_1) < I_{occ}(e_2)) \lor (T(e_1).site \neq T(e_2).site) \land (Iocc_{global}(e_1) < Iocc_{global}(e_2))$

Event e_1 is produced before event e_2 if:

(i) the two events are produced at the same location and the instant of production of e_1 is smaller than that of e_2; or

(ii) if they are produced at different locations and the global instant of production of e_1 is smaller than that of e_2.

2) Simultaneous:

$T(e_1) = T(e_2) \iff (T(e_1).site = T(e_2).site) \land (I_{occ}(e_1) = I_{occ}(e_2))$

Events e_1 and e_2 are simultaneous if the two are produced at the same location and they have the same instant of production.

3) Concurrent:

$T(e_1) \sim T(e_2) \iff (T(e_1) < T(e_2)) \lor (T(e_2) < T(e_1))$

Events e_1 and e_2 are concurrent if e_1 was is produced before e_2 or vice–versa.

The relation $<$ defines a strict partial order [12] on a set of timestamps of primitive events. Then two events e_1 and e_2 can be ordered as follows:

e_1 *before* e_2:

$T(e_1).local < T(e_2).local \rightarrow T(e_1).global \leq T(e_2).global$

e_1 *simultaneous to* e_2:

$T(e_1).local = T(e_2).local \rightarrow T(e_1).global = T(e_2).global$

e_1 *concurrent to* e_2:

$T(e_1).local \sim T(e_2).local \rightarrow |\ T(e_1).global - T(e_2).global\ | \leq 1g_g$

(g_g represents the granularity of the global reference clock.)

It is worth noting that the difference between simultaneous and concurrent events consists in that simultaneous events are produced within the same location. For the ordering of composite events, the previous definitions are modified as follows [48]:

1) $T(e_1) < T(e_2) \iff (\forall t_2 \in T(e_2), \exists t_1 \in T(e_1))$ *such that* $(t_1 < t_2)$

The composite event e_1 is triggered before the composite event e_2 if and only if for all events which trigger e_2 exists an event, which triggers e_1, that has been produced before.

2) Concurrent:

$T(e_1) \sim T(e_2) \iff (\forall t_1 \in T(e_1), \forall t_2 \in T(e_2))$ *such that* $(t_1 \sim t_2)$

The composite events e_1 and e_2 are concurrent if and only if all the triggering events of e_1 and e_2 are concurrent.

3) Not comparable:

$T(e_1) \bowtie T(e_2) \iff \neg ((T(e_1) < T(e_2)) \lor (T(e_1) > T(e_2)) \lor (T(e_1) \sim T(e_2)))$

The timestamps $T(e_1)$ et $T(e_2)$ of the composite events e_1 and e_2 are not comparable if it is impossible to determine an ordering relationship between e_1 and e_2.

4) $T(e_1) <\sim T(e_2) \iff T(e_1) \sim T(e_2) \lor T(e_1) < T(e_2)$

The composite event e_1 is triggered approximately before the composite event e_2 if and only if they are concurrent or if e_1 is produced before e_2.

---

[11] The notation $T(e).< attribute >$ is used to denote the local (global) instant of occurrence and the timestamp location.

[12] Let the reader remind that an ordering relation $<$ on a set A defines a strict partial order if it is transitive and non–reflexive. It defines a total ordering if, moreover, $\forall\ x,y,z \in A$ *either* $x < y$, or $x = y$, or $y < x$. In centralized systems, a strict total order of type $<$ is non-reflexive, transitive and asymmetric; on the contrary, a total order $\leq$ is reflexive, transitive and asymmetric.