Formalization and Detection of Events Using Interval-Based Semantics

Raman Adaikkalavan
IT Laboratory and Department of CSE
The University of Texas at Arlington
Arlington, TX 76019-0015
(+1) - 817-272-5188
adaikkal@cse.uta.edu

Sharma Chakravarthy
IT Laboratory and Department of CSE
The University of Texas at Arlington
Arlington, TX 76019-0015
(+1) - 817-272-2082
sharma@cse.uta.edu

ABSTRACT

Active databases utilize Event-Condition-Action rules to provide active capability to the underlying system. An event was initially defined to be an instantaneous, atomic occurrence of interest and the time of occurrence of the last event in an event expression was used as the time of occurrence for an entire event expression (detection-based semantics), rather than the interval over which an event expression occurs (interval-based semantics). This introduces semantic discrepancy for some operators when they are composed more than once. Currently, all active databases detect events using the detection-based semantics rather than the interval-based semantics. SnoopIB is an interval-based event specification language developed for expressing primitive and composite events that are part of active rules. Algorithms for event detection using interval-based semantics pose some challenges, as not all events are known (especially their starting points). In this paper, we address the following: 1) briefly explain the need for interval-based semantics, 2) formalization of events accumulated over a semantic window and 3) how diversified events (e.g., sliding window, accumulated) are detected using interval-based semantics in the context of Sentinel – an active object oriented database.

1. INTRODUCTION

There is consensus in the database community on Event-Condition-Action (or ECA) rules as being one of the most general formats for expressing rules in an active database management system. As the event component was the least understood (conditions correspond to queries, and actions correspond to transactions) part of ECA rule, there is a large body of work [1-11] on the language for event specification. Snoop [1, 2] was developed as the event specification component of the ECA rule formalism used as a part of the Sentinel project [12-15]. Snoop supports expressive ECA rules that include coupling modes and parameter contexts or event consumption modes.

An *event* is an occurrence of interest, which can be either *primitive* (e.g., depositing cash in bank) or *composite* (e.g., depositing cash in bank, *followed by* withdrawal of cash from bank). Primitive events occur at a point in time (i.e., time of depositing) and composite events occur over an interval (i.e., interval starts at the time cash is deposited and ends when cash is withdrawn). Thus, primitive events are detected at a point in time, whereas the composite events can be detected either at the end of the interval (i.e., detection-based semantics, where start of the interval is not considered) or can be detected over the interval (i.e., interval-based semantics). Event consumption modes are needed while detecting events, since, not all the detected events using unrestricted context (i.e., none of the event occurrences are discarded after participating in event detection) are meaningful for an application.

In all event specification languages used in Active DBMSs (Snoop [1, 2], COMPOSE [3, 4], Samos [5, 6], ADAM [7, 8], ACOOD [16, 17], event-based conditions [9], and Reach [10, 11, 18]), events are considered as "instantaneous", although an event occurs over an "interval". Because of this, all these event specification languages detect a composite event at the *end of an interval* over which it occurs (i.e., detection-based semantics). When events are detected using the detection-based semantics, where *event occurrence* and *event detection* are not differentiated, it leads to certain unintended semantics as pointed out in [19, 20] when certain operators, such as sequence are composed more than once.

SnoopIB [21-23] is an event specification language based on interval-based semantics. Active rules have been shown to provide active capability to object oriented databases, relational databases, and so on. Lately, they have been shown to support diverse application areas such as information filtering [24], information security [25], data stream processing [26], XML processing [27], semantic web [28], and sensor databases [29]. Interval-based semantics has far-reaching applications and has been utilized in information filtering [24], XML processing [27], and cooperative information system [30].

ADVANCES IN DATA MANAGEMENT 2005
Jayant Haritsa, T.M. Vijayaraman (Editors)

SnoopIB event operators were formally defined in the recent context (i.e., for applications where the events are happening at a fast rate and multiple occurrences of the same type of event only refine the previous data value e.g., sensor applications.) using interval-based semantics in [22] and over a sliding window (i.e., for trend analysis and forecasting applications e.g., stock market, after-the-fact diagnosis) using interval-based semantics in [23]. Interval-based semantics has substantial differences as compared to detection-based semantics and is explained below using a simple example (refer [21, 22, 30] for more critical examples).

"Department of transportation needs to find whether there is a traffic jam in (*Road_2* AND *Road_3*) after *Road_1* has encountered a traffic jam". Thus, the condition (time of traffic jam in *Road_1* < **(**time of traffic jam in *Road_2* AND time of traffic jam in *Road_3***)**) is checked in order to detect a traffic jam. Let us assume that there is a traffic jam at the following time: *Road_1* (10.00 a.m.), *Road_2* (9.30 a.m.) and *Road_3* (12.00 p.m.). Thus, we should check the following condition (10.00 a.m. < (9.30 a.m. AND 12.00 p.m.)). As the detection-based semantics uses the end time of the entire expression, 12.00 p.m. is treated as the time of (*Road_2* AND *Road_3*) traffic jam. Thus, the condition whether (10.00 a.m. < 12.00 p.m.) is checked, and since it is true traffic jam is notified. This is not as intended, since traffic jam in *Road_2* occurs at 9.30 a.m. way before *Road_1*.

When interval-based semantics is used, the time for *Road_1* traffic jam is treated as (10 a.m., 10 a.m.), where the first entry represents the start time and the second entry represents the end time. Thus, traffic jam in (*Road_2* AND *Road_3*) occurs over an interval (9.30 a.m. to 12.00 p.m.), where 9.30 a.m. starts the event and 12.00 p.m. ends the event. When the condition (10 a.m. < 9.30 a.m.) (i.e., whether traffic jam in *Road_1* has occurred before the start time of the composite event (*Road_2* AND *Road_3*)) is checked, it returns false and traffic jam is not notified.

Detection-based semantics was adopted as *begin* and *end* events were of significance in most of the database related work. From our example above, it is evident that events are detected as intended when interval-based semantics is used in place of detection-based semantics. Thus, event detection using interval-based semantics is a *trusted way* and not just another way of detecting events.

1.1 Our Contributions

SnoopIB event operators were formally defined in the recent context using interval-based semantics in [22] and over a sliding window using interval-based semantics in [23]. In this paper, we have formally defined event operators for detecting accumulated events over a semantic window (i.e., applications where multiple occurrences of a constituent event needs to be grouped and used in a meaningful way when the event occurs e.g., banking application) using interval-based semantics. Algorithms for event detection using interval-based semantics pose some challenges, as not all events are known (especially their starting points). We discuss the implementation issues and show how events are detected in various event consumption modes using interval-based semantics in the context of Sentinel – an active object oriented database.

1.2 Outline

The rest of the paper is organized as follows. Section 2 refers to related work on event specification. Section 3 explains the interval-based semantics of Snoop. Section 4 extends the above to the accumulated events that are detected over a semantic window. Section 5 provides the implementation details along with the algorithms. Section 6 has conclusions and future work. Appendix A has additional algorithms.

2. RELATED WORK

There has been a considerable amount of work done in the interval-based semantics. Why the interval-based semantics is needed for event detection is explained with *concrete* examples in [30], using Snoop operators, but does not deal with formal semantics, algorithms and implementation for any of the context in Snoop. [31] explains the event detection using the duration-based (i.e., interval-based) semantics, but why it is needed, what operators are supported, how it is implemented and the formal semantics is not explained.

Snoop [1, 2] uses event graphs to detect the composite event, whereas Samos [5, 6] uses Petri-nets to detect the composite events, likewise all the aforementioned event specification languages detects the composite event using different approaches, but all of them use detection-based semantics, which has some problems as we have seen before. Details of event detection by other event specification languages and why they are not sufficient can be found in [21].

Algorithms for event composition and event consumption, which make use of accuracy interval based time stamping is illustrated along with a window mechanism to deal with varying transmission delays when composing events from different sources, are dealt in [32]. The paper claims that event consumption modes like recent and chronicle can be unambiguously defined by using an accuracy interval order that guarantees the property of time consistent order. Even though this system uses "accuracy interval based time stamping" guaranteeing the time consistent order for the event arrival, it uses the detection-based semantics for the composite event detection, which has the same drawbacks.

3. INTERVAL-BASED SEMANTICS

For the purpose of this paper, we assume an equidistant discrete time domain having "0" as the origin and each time point represented by a non-negative integer as shown in Figure 1.

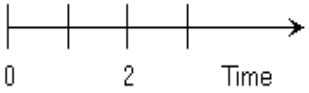

Figure 1. Time Line.

3.1 Primitive Events

Events can be file operations (i.e., opening, closing, etc.) in operating systems, method execution by objects in OODBMs, data manipulations such as insert, delete and update in RDBMSs, events based on system clock of the underlying system (i.e., absolute or relative temporal events), external events (i.e., based on the data from sensors), occurrence of regular expressions or

keywords in text streams, and so on. These events that are predefined in the underlying system (i.e., domain-specific) are known as primitive or simple events (for more detail refer to [1, 2, 33]). For example, *a method execution* by an object in an object-oriented database is a primitive event. These *method executions* can be grouped into *before* and *after* events (or *event types)* based on when they are detected (*immediately before* or *after the method call*).

An event occurs over a time interval and is denoted by O (E [t_1, t_2]) (see Figure 2, where O represents the interval-based semantics, E is the event, t_1 is the start interval of the event denoted by ↑E, and t_2 is the end interval of the event denoted by E↓). In the case of primitive events, the start and the end interval are assumed to be the same (i.e., $t_1 = t_2$). For events that span over an interval, the event *occurs* over the interval [t_1, t_2] and is *detected* at the end of the interval.

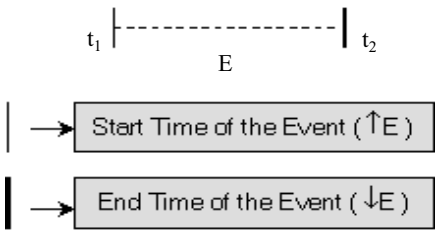

Figure 2. Event Notations.

3.2 Event Expressions

For many applications, supporting only primitive events is inadequate. In many real-life applications, there is a need for specifying more complex patterns of events such as, *arrival of a report followed by a detection of a specified object in a specific area*. The above shown complex pattern cannot be expressed with a language that does not support expressive event operators along with their semantics. An appropriate set of operators along with the closure property allows one to construct complex composite events by combining primitive events and composite events in ways meaningful to an application interested in situation monitoring. To facilitate this, we have defined a set of event operators along with their semantics. Snoop(IB) is an event specification language that is used to specify combinations of events. Motivation for the choice of these operators and how they compare with other event specification languages can be found in [1, 2]. Some of the event operators are AND (two events occur together in any order), Sequence (two events occur together in a particular order), NOT (one event does not occur in between two other events), OR (any one of the two events occur), Aperiodic (aperiodic occurrence of a event between two other events), Periodic (periodic occurrence of an event between two other events), Aperiodic* and Periodic* (cumulative versions of Aperiodic and Periodic operators), Frequency or Cardinality (number of times an event should occur), and Plus (event occurrence based on another event plus some time).

3.3 Composite Events

Composite events are composed of more than one primitive or composite event using event operators. These events are constructed using primitive events and event operators in a recursive manner. A composite event consists of a number of primitive events and operators; and the set of primitive events of a composite event are termed as constituent events of that composite event. A composite event is said to occur *over an interval*, but is *detected* at the point when the last constituent event of that composite event is detected. The detection and occurrence semantics is clearly differentiated and the detection is defined in terms of occurrence as shown in [19, 20]. Note that occurrence of events cannot be defined in terms of detection which was the problem with the earlier detection-based approaches.

We introduce the notion of an *initiator*, *detector*, and *terminator* for defining event occurrences. A composite event occurrence is based on the initiator, detector and terminator of that event which in turn are constituent events of that composite event. An *initiator* of a composite event is the first constituent event whose occurrence starts the composite event. *Detector* of a composite event is the constituent event whose occurrence detects the composite event, and *terminator* of a composite event is the constituent event that is responsible for terminating the composite event. For example, when a stock trading agent requests for a stock quote every hour from 9 a.m. to 5 p.m., then 9 a.m. starts the event (i.e., initiator), 5 p.m. terminates the event (i.e., terminator) and every hour (i.e., 10 a.m., 11 a.m., …) detects an event (i.e., detector). For some operators, the detector and terminator are different (e.g., Aperiodic), while for other operators, detector and terminator are the same (e.g., Sequence).

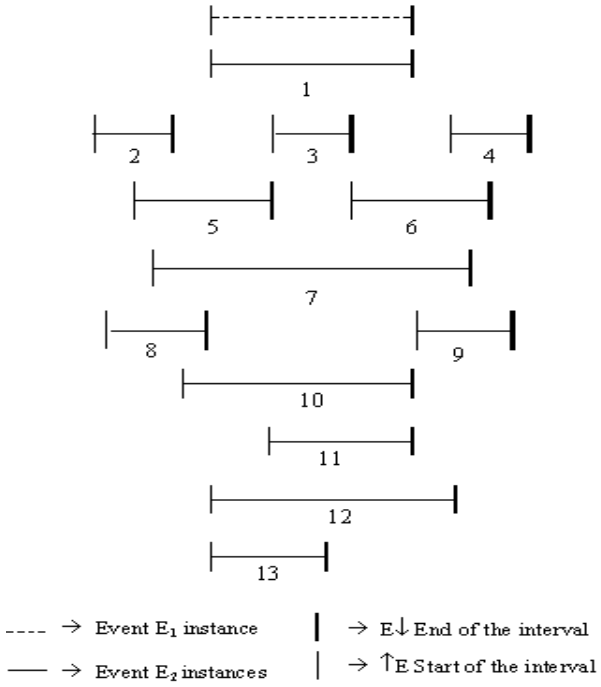

Figure 3. Overlapping Event Combinations.

A composite event E occurs *over a time interval* and is defined by E [t_1, t_2] where E is a composite event, t_1 is the start time of the composite event occurrence and t_2 is the end time of composite event occurrence (t_1 is the starting time of the first constituent event that occurs (*initiator*) and t_2 is the end time of the detecting or terminating constituent event (*detector or terminator*) and they

are denoted by ↑E and E↓ respectively). Below, "O" represents the occurrence-based or interval-based semantics.

Start of an event: $\mathbf{O(\uparrow E, t) \triangleq \exists t' (t \leq t' \wedge O(E, [t, t']))}$

End of an event: $\mathbf{O(E\downarrow, t) \triangleq \exists t' \leq t (O(E, [t', t]))}$

Event Combinations: Nature of constituent event occurrences of a composite event is another important aspect as they can be either overlapping or disjoint.

Overlapping Event Combinations: When events are allowed to overlap, all the possible combinations in which two events can occur [34, 35] are shown in Figure 3. All operators formally defined in this paper assume that events occur in an overlapping fashion.

Disjoint Event Combinations: When events are not allowed to overlap, we have fewer combinations. This may be meaningful for many applications where the same event should not participate in more than one composite event or when only one of the overlapping events is of interest. The possible disjoint event combinations are shown in Figure 4.

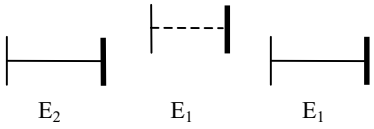

Figure 4. Disjoint Event Combinations.

3.4 Event Histories

In real world, events occur over a time line (or online). Events can be detected as and when it occurs as far as the events are predefined in the system (i.e., primitive events). Even though the time of occurrence of a composite event is over an interval in which it occurs, it is detected only when the last constituent event occurs. Thus, history of an initiator and other constituent events should be maintained so that they can be paired when detector/terminator occurs. An event history maintains a history of event occurrences up to a given point in time. Suppose e_1 is an event instance of type E_1, then E_1 [H] represents the event history that stores all the instances of the event E_1 (namely e_1^i). In the following sections, using the notion of *event histories,* we formalize SnoopIB operator definitions taking event consumption modes (or parameter contexts) into account. In order to extend these definitions to event consumption modes following notations are used.

$\mathbf{E_i [H] = \{e_i^j [t_{si}, t_{ei}]\}}$

E_i [H] → Event history for event E_i,

t_{si} – Start time of an event instance e_i^j of event E_i

t_{ei} – End time of an event instance e_i^j of event E_i

For example, event histories for the event occurrences shown in Figure 5 are shown below.

E_1 [H] = {e_1^1 [3, 5]}

E_2 [H] = {e_2^1 [1, 2], e_2^2 [4, 6]}

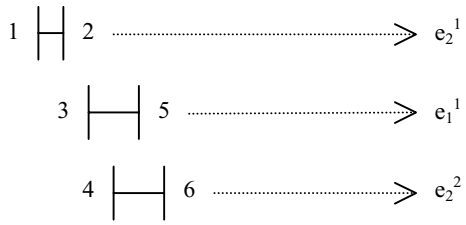

Figure 5. Event Occurrences.

3.5 Event Consumption Modes

Events in the ECA rules are detected in unrestricted (or general) context. This means events, once they occur, cannot be discarded at all. For a ";" (Snoop sequence operator) event, all event occurrences that occur after a particular event will get paired with that event as per the unrestricted context semantics. In the absence of any mechanism for restricting event usage (or consumption), events need to be detected and parameters for those composite events need to be computed using the unrestricted context definitions of the Snoop event operators. However, the number of events produced (with unrestricted context) can be large and not all event occurrences may be meaningful for an application. In addition, detection of these events has *substantial computation and space overhead*, which may become a problem for situation monitoring applications. Thus, Snoop(IB) has five event consumption modes based on the application domains and they are: Recent, Recent-Unique, Chronicle, Continuous, and Cumulative.

Motivations behind the recent, continuous, and cumulative contexts are given below in an intuitive way. In addition, semantics used for event detection in these contexts are also discussed.

Recent Context: In applications where events are happening at a fast rate and multiple occurrences of the same event only refine the previous value can use this context. Only the most recent or the latest initiator for any event that has started the detection of a composite event is used in this context. This entails that the most recent occurrence just updates (summarizes) the previous occurrence(s) of the same event type. In this context, *not all occurrences* of a constituent event will be used in the composite event detection. An initiator will continue to initiate new event occurrences until a *new initiator* or a *terminator* occurs.

Continuous Context (Sliding Window Events): In applications where event detection along a moving time window is needed, continuous context can be used. This context is especially useful for tracking trends of interest on a sliding time point governed by the initiator event. For example, computing *change of more than 20% in DowJones average in any 2-hour period* requires each change to initiate a new occurrence of an event. In this context, each initiator starts the detection of that composite event, and a single detector or terminator may detect one or more occurrences of that same composite event. In other words, each initiator starts a new window, and the events are detected until (or when) a terminator occurs. For binary SnoopIB operators, all the constituent events (initiator, detector and/or terminator) are deleted once the event is detected. For ternary SnoopIB operators detector and terminator are different. Detectors detect the event occurrence (e.g., Aperiodic) and are deleted once detected. Terminator terminates the event (e.g., Aperiodic*) and deletes

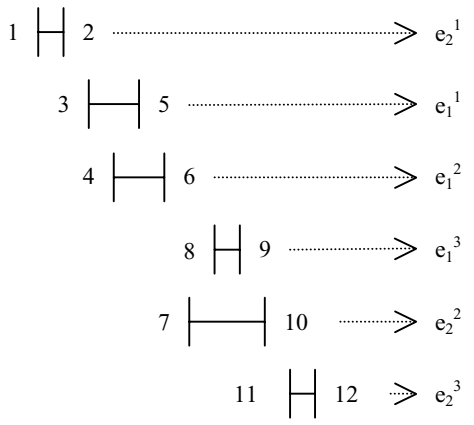

Figure 6. Examples for SEQUENCE Operator.

corresponding initiator and terminator pair along with the constituent events that cannot be used in future events. Future events are the events that are initiated by the initiators that are not paired with this terminator.

Cumulative Context (Semantic Window Events): Applications use this context when multiple occurrences of constituent events need to be grouped (or accumulated) and used in a meaningful way when the event occurs (e.g., banking application). In this context, all occurrences of an event type are accumulated as instances of that event until the event is terminated (i.e., forming a semantic window based on the earliest initiator that was not terminated and a terminator). An event occurrence does not participate in two distinct occurrences of the same composite event. In both the binary and ternary operators, detector and terminator are same, and once detected and terminated all constituent event occurrences that were part of the detection are deleted. Other events that can act as a constituent event for some future event are preserved.

4. EVENT OPERATOR FORMALIZATION

In this section, we provide the formalization of event operators in cumulative context using event histories (explained in Section 3.4) based on the formal semantics defined for unrestricted [19, 20], recent [22] and continuous [23] contexts.

Below, event operators are defined intuitively, examples for event detection over cumulative context using interval-based semantics are provided, and the formal definitions of the operators in cumulative context are given. "O" represents the occurrence-based or interval-based semantics.

We will use the start and end of an event defined earlier for formally defining the event operators. To enable us to express this more concisely the predicate O_{in} is defined as follows [19, 20].

$$\mathbf{O_{in}\ (E\ [t_1, t_2]) \triangleq \exists t_1', t_2'\ (t_1 \le t_1' \le t_2' \le t_2 \wedge O\ (E, [t_1', t_2']))}$$

4.1 SEQUENCE (;) Event Operator

Sequence Operator (;): O (E_1; E_2, [t_1, t_2]). Sequence of two events E_1 and E_2, denoted by E_1; E_2, occurs when E_2 occurs provided E_1 has already occurred. This implies that the end time of occurrence of E_1 is guaranteed to be less than the start time of occurrence of E_2. E_1 is the *initiator* and E_2 is the *detector/terminator* of the sequence event.

Example: In this context, a detector or terminator produces only one event. Event histories are used for the detection of the ";" operator defined above. Event histories corresponding to the event occurrences shown in Figure 6 are given below, where E_1 [H] is the initiator event history and E_2 [H] is the terminator even history.

E_1 [H] = {e_1^1 [3, 5], e_1^2 [4, 6], e_1^3 [8, 9]}

E_2 [H] = {e_2^1 [1, 2], e_2^2 [7, 10], e_2^3 [11, 12]}

When terminator event e_2^1 occurs there is no initiator event in E_1 [H] that satisfies the ";" operator condition. Event e_1^1 occurrence initiates a sequence event. Event e_1^2 occurrence is accumulated. When the event e_2^2 occurs, E_1 [H] has events {e_1^1 [3, 5], e_1^2 [4, 6], e_1^3 [8, 9]}. Thus, e_2^2 detects the event initiated by event e_1^1 generating the following event (e_1^1, e_1^2, e_2^2) [3, 10], since it satisfies the sequence condition ($t_{s1} \le t_{e1} < t_{s2} \le t_{e2}$) (i.e., ($3 \le 5 < 7 \le 10$) for pair ($e_1^1$, e_2^2)). As shown, all the events in between the pair (e_1^1, e_2^2), in this case e_1^2, are accumulated. Even though e_1^3 occurred before e_2^2, it is not detected since it does not satisfy the condition ($9 < 7$). According to the cumulative context definition, events e_1^1, e_1^2 and e_2^2 are deleted as they have already participated in event detection and cannot act as constituent events for future detections. In addition, event e_1^3 is also deleted as it has occurred before the start time of e_2^2 and does not satisfy the sequence condition. As there are no events after end time of e_2^2, event e_2^3 does not detect any event. Event pairs detected by sequence operator in continuous context are: (e_1^1, e_1^2, e_2^2) [3, 10]

Formal Definition in Cumulative Context:

$$
\begin{aligned}
&O\ (E_1\ ;\ E_2, [t_{s1}, t_{e2}]) \triangleq \\
&\forall E_2 \in E_2\ [H] \\
&\{O\ (E_2, [t_{s2}, t_{e2}]) \wedge (\nexists E_2'\ [t_{s,}\ t_e] \mid (t_e < t_{e2}) \wedge E_2' \in E_2\ [H]) \\
&\ \wedge \{\forall E_1 \in E_1\ [H]\ (O\ (E_1, [t_{s1,}\ t_{e1}]) \wedge (t_{s1} \le t_{e1} < t_{s2} \le t_{e2}))\} \\
&\} \\
&\vee \\
&\forall E_2 \in E_2\ [H] \\
&\{O\ (E_2, [t_{s2}, t_{e2}]) \wedge ((\exists E_2'\ [t_s, t_e] \mid (t_e < t_{e2}) \wedge E_2' \in E_2\ [H]) \\
&\ \wedge (\nexists E_2''\ [t_{s'}, t_{e'}] \mid (t_{e'} > t_e) \wedge (t_{e'} < t_{e2}) \wedge E_2'' \in E_2\ [H])) \\
&\ \wedge \{\forall E_1 \in E_1\ [H]\ (O\ (E_1, [t_{s1,}\ t_{e1}]) \wedge (t_{s1} \le t_{e1} < t_{s2} \le t_{e2}) \\
&\ \wedge (t_{s1} > t_e) \wedge (\nexists E_1'\ [t_{s1'}, t_{e1'}] \mid (t_{s1}' > t_e) \wedge (t_{s1}' < t_{s1}) \\
&\qquad \wedge E_1' \in E_1\ [H])) \\
&\quad \} \\
&\}
\end{aligned}
$$

Two events $e_1 \in E_1$ [H] and $e_2 \in E_2$ [H] are said to occur in sequence in the cumulative context only when there is no occurrence of $e_2' \in E_2$ [H] before the occurrence of e_2 and all the other occurrences of $e_1' \in E_1$ [H] that occurs in between the pair e_1 and e_2 are accumulated. There are two cases to formally define the operator (refer the formal definition above). First case handles when there is no other terminator is available in the terminator history (i.e., first occurrence of the terminator). In other words, there should be no occurrence of other terminators before this terminator and this terminator should be in sequence with all initiators till that point. In this case, all the event occurrences of the initiator are accumulated, and the cumulative event is

detected. Second case handles when there is more than one terminator present in the history. For this case, there should be no occurrence of other terminators in between start of the initiator and end of the terminator or a terminator can occur only if its end time is less than start time of the initiator. In other words, an initiator starts an event occurrence and a terminator terminates and detects the ";" event with events in between as constituent events and there should be no other instance of the terminator.

4.2 OR Event Operator

OR Operator (∇): O ($E_1 \nabla E_2$, [t_1, t_2]). Disjunction of two events E_1 and E_2, denoted by $E_1 \nabla E_2$, occurs when E_1 occurs or E_2 occurs. Occurrences of one of E1 or E2 act as both *initiator* and *terminator*. The semantics of "∇" does not change with cumulative context as each occurrence is detected individually.

4.3 PLUS Event Operator

Plus Operator: O (Plus (E_1, E_2) [t, t]). A Plus operator is used to specify a relative time event [36]. A Plus operator combines two events E_1 and E_2 where E_1 can be any type of event and E_2 is a time string [t]. E_1 is the initiator and E_2 is the terminator. The Plus event occurs *only once* after time [t], after the event E_1 occurs. Plus operator's unrestricted context definition [21] holds for the cumulative context, since Plus operator is detected only once after the occurrence of the event E_1 and there is only one terminator for an initiator.

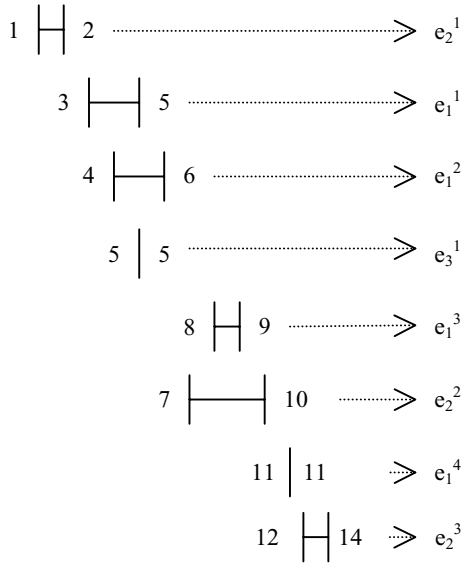

Figure 7. Examples for NOT Operator.

4.4 NOT Event Operator

NOT Operator (¬): O (¬ (E_3) [E_1, E_2], [t_1, t_2]). NOT operator detects the non-occurrence of the event E_3 in the closed interval formed by $E_1\downarrow$ and $E_2\uparrow$.

Example: "¬" Operator can be expressed as the sequence of E_1 and E_2 where there is no occurrence of the event E_3 in the interval formed by these events. Event histories corresponding to the event occurrences shown in Figure 7 are given below, where E_1 [H] is the event e_1 history, E_2 [H] is the event e_2 history, and E_3 [H] is the event e_3 history.

E_1 [H] = {e_1^1 [3, 5], e_1^2 [4, 6], e_1^3 [8, 9], e_1^4 [11, 11]}

E_2 [H] = {e_2^1 [1, 2], e_2^2 [7, 10], e_2^3 [12, 14]}

E_3 [H] = {e_3^1 [5, 5]}

When terminator event e_2^1 occurs there is no initiator event in E_1 [H] that can pair with e_2^1. Event e_1^1 occurrence initiates a sequence event. Event e_1^2 occurrence is accumulated. When the event e_2^2 occurs, E_1 [H] has events {e_1^1 [3, 5], e_1^2 [4, 6], e_1^3 [8, 9]}. But, event e_1^1 [3, 5] cannot combine with event e_2^2 [7, 10] since there is an occurrence of e_3^1 [5, 5] in between e_1^1 and e_2^2 (i.e., $5 \le 5 \le 7$), thus a NOT event is not detected. Event e_1^4 [11, 11] initiates the next NOT event. When event e_2^3 [12, 14] occurs, it pairs with event e_1^4 detecting (e_1^4, e_2^3) [11, 14] as there is no occurrence of event e_3 in the interval [11, 12]. The event pair generated by NOT operator in cumulative context is: {(e_1^4, e_2^3) [11, 14]}

Formal Definition in Cumulative Context:

$$
\begin{aligned}
&O\,(\neg\,(E_3)\,[E_1, E_2], [t_{s1}, t_{e2}]) \triangleq \\
&\quad \forall E_2 \in E_2\,[H] \\
&\quad \{O\,(E_2, [t_{s2}, t_{e2}]) \wedge (\nexists E_2'\,[t_s, t_e] \mid (t_e < t_{e2}) \wedge E_2' \in E_2\,[H]) \\
&\quad \wedge \{\forall E_1 \in E_1\,[H] \wedge \forall E_3 \in E_3\,[H] \\
&\qquad (O\,(E_1, [t_{s1}, t_{e1}]) \wedge (t_{s1} \le t_{e1} < t_{s2} \le t_{e2}) \\
&\qquad \wedge \neg O_{in}\,(E_3, [t_{e1}, t_{s2}]))\} \\
&\quad \} \\
&\quad \vee \\
&\quad \forall E_2 \in E_2\,[H] \\
&\quad \{O\,(E_2, [t_{s2}, t_{e2}]) \wedge ((\exists E_2'\,[t_s, t_e] \mid (t_e < t_{e2}) \wedge E_2' \in E_2\,[H]) \\
&\quad \wedge (\nexists E_2''\,[t_s', t_e'] \mid (t_e' > t_e) \wedge (t_e' < t_{e2}) \wedge E_2'' \in E_2\,[H])) \\
&\quad \wedge \{\forall E_1 \in E_1\,[H] \wedge \forall E_3 \in E_3\,[H] \\
&\qquad (O\,(E_1, [t_{s1}, t_{e1}]) \wedge (t_{s1} \le t_{e1} < t_{s2} \le t_{e2}) \wedge (t_{s1} > t_e) \\
&\qquad \wedge (\nexists E_1'\,[t_{s1}', t_{e1}'] \mid (t_{s1}' > t_e) \wedge (t_{s1}' < t_{s1}) \\
&\qquad\quad \wedge E_1' \in E_1\,[H]) \\
&\qquad \wedge \neg O_{in}\,(E_3, [t_{e1}, t_{s2}])) \\
&\quad \} \\
&\}
\end{aligned}
$$

Formal definition above has two cases similar to the sequence operator formal definition. Non occurrence of event $e_3 \in E_3$ [H] between two events $e_1 \in E_1$ [H] and $e_2 \in E_2$ [H] is said to occur in the cumulative context only when there is no occurrence of $e_2' \in E_2$ [H] before the occurrence of e_2 and all the other occurrences of $e_1' \in E_1$ [H] that occurs in between the pair e_1 and e_2 are accumulated. First case handles when there is no other terminator is available in the terminator history (i.e., first occurrence of the terminator). In other words, there should be no occurrence of other terminators before this terminator and this terminator should be in sequence with all initiators till that point and there should not be any occurrence of event e_3 in between the initiator and terminator as specified by the condition ($\neg O_{in}$ (E_3, [t_{e1}, t_{s2}])). In this case, all the event occurrences of the initiator are accumulated, and the cumulative event is detected. Second case handles when there is more than one terminator present in the history. For this case, there should be no occurrence of other terminators in between start of the initiator and end of the terminator or a terminator can occur only if its end time is less than start time of the initiator. In addition, there should not be any occurrence of event e_3 in between the initiator of the composite

event and the terminator as specified by the condition ($\neg O_{in}$ (E_3, [t_{e1}, t_{s2}])).

5. INTERVAL-BASED EVENT DETECTION

In section **4** formal definitions using event histories were given for event operators in cumulative context. In real world applications such as monitoring applications we *cannot assume* to have complete event histories to detect the composite event since the events occur online. In this section we will show the events detected based on histories, and explain how composite events are detected using event graphs in recent, continuous and cumulative contexts from the implementation perspective when events occurs online. It poses some challenges as the start of the event is not known before hand. We also show that the events detected using event histories (or formalization of event operators) and event graphs (or implementation of operators) are the same.

Let us take a simple composite event (E_1; E_2) as an example (please refer [21] for more complex/detailed examples). Event (E_1; E_2) represents the SEQUENCE between event E_1 and E_2, occurs when E_2 occurs provided E_1 has already occurred. This implies that the end time of occurrence of E_1 is guaranteed to be less than the start time of occurrence of E_2. E_1 is the *initiator* and E_2 is the *terminator* of the sequence event. Event occurrences that will be used for detecting events using both event histories and event graphs are shown in Figure 6.

5.1 Event Detection Using Event Histories

Event histories corresponding to the event occurrences shown in Figure 6 are given below, where E_1 [H] is the initiator event history and E_2 [H] is the terminator even history.

E_1 [H] = {e_1^1 [3, 5], e_1^2 [4, 6], e_1^3 [8, 9]}

E_2 [H] = {e_2^1 [1, 2], e_2^2 [7, 10], e_2^3 [11, 12]}

Events detected in Cumulative Context: Based on the formal semantics provided in Section 4.1 and the event histories above, all the event occurrences for the composite event (E_1; E_2) detected using cumulative context is given below.

(e_1^1 [3, 5], e_1^2 [4, 6], e_2^2 [7, 10]) [3, 10]

Events detected in Recent Context: Formal definition for the sequence operator in this context is provided in [22]. According to the definition of the recent context in section 3.5 a terminator pairs only with a recent initiator, and there should be no other instance of the terminator between them. From the event history, we can see that for terminator event e_2^1 [1, 2] there are no initiators, and for terminator event e_2^2 [7, 10] the recent initiator is e_1^3 [8, 9]. Terminator event e_2^2 [7, 10] cannot be paired with event e_1^3 [8, 9] as there it does not satisfy the sequence condition. Thus, for these event occurrences composite event (E_1; E_2) is not detected using recent context.

Events detected in Continuous Context: Formal definition for the sequence operator in this context is provided in [23]. According to the definition of the continuous context in section 3.5 a terminator terminates more than one initiator. This context is similar to cumulative context, except that the number of events generated equals the number of initiators. Thus, the event occurrences for the composite event (E_1; E_2) detected using continuous context are given below.

{(e_1^1 [3, 5], e_2^2 [7, 10]) [3, 10], (e_1^2 [4, 6], e_2^2 [7, 10]) [4, 10]}

5.2 Event Detection Using Event Graphs

Sentinel [12-15] uses an event graph or event detection graph (EDG) for representing an event expression in contrast to other approaches such as Petri nets used by Samos [5, 6] and an extended finite state automata used by COMPOSE [3, 4]. By combining event trees on common sub expressions, an event graph is obtained. Data flow architecture is used for the propagation of primitive events to detect composite events. By using event graphs, the need for detecting the same event multiple times is avoided since the event node can be shared by many events. In addition to reducing the number of detections, this approach saves substantial amount of storage space (for storing event occurrences and their parameters), thus leading to an efficient approach for detecting events.

As mentioned earlier, primitive events are detected by the underlying system and composite events are detected using the occurrence of its constituent events. The time of occurrence of a composite event depends on the event operator semantics and detection semantics (either detection-based or interval-based). Interval-based semantics uses the time of occurrence of both the first and last constituent event in an event expression as the time of occurrence for the entire event expression.

Algorithms and Implementation: Semantics of the event operators are defined using the event history in the previous section. In this section, we will provide algorithms that detect events according to the interval-based semantics. In the manner in which ECA rules are used for monitoring situations, events occur over a time line and are sent to the event detector. All events in the form of an event history are not submitted to the event detector. In fact, as part of event detection, the event detector at any point sees only a partial history in time. Algorithms presented in the following subsections detect events according to interval semantics although they do not see the complete history at any given point in time. How the start interval is handled is shown in the algorithm. The algorithms defined in the following subsections are implemented in Sentinel. The formal definitions and algorithms have been designed for all contexts and are detailed in [21]. Notations that are used while writing the algorithms are shown in Table 1.

Table 1. Notations used in Algorithms

e_i (e.g., e_1, e_2)	Primitive or Composite event instance or occurrence
$\mathbf{E}_i$ (e.g., E_1, E_2)	An event List that maintains the partial history of the occurrences of event **e***i*
t_s	Start time of the event (Start Interval)
t_e	Ending time of the event (End Interval)

Event Detection Graph: In an EDG, leaf nodes represent primitive events and internal nodes represent composite events (or event operators) and event occurrences flows in a bottom-up fashion. When a primitive event occurs and is detected, it is sent from its node to the parent node (if necessary) for detecting a composite event. Figure 8 shows a composite event SEQUENCE with two events E_1 and E_2. Leaf nodes, E_1, and E_2, represent the

primitive events and node "A" represent the composite event SEQUENCE. Whenever there is an event E_1 or E_2 occurrence it is propagated to node "A". SEQUENCE event is detected whenever both its constituent events occur, where E_1 precedes E_2 in time. As described in section 3.5, introduction of event consumption modes make event detection more meaningful for diverse applications.

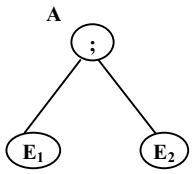

Figure 8. Event Graph for E_1 ; E_2.

Even though Snoop has 5 contexts, we explain the event detection only in recent, continuous, and cumulative contexts using the EDG shown in Figure 8. In order to provide more meaningful examples while discussing the algorithms, we consider the events E_1 and E_2 shown in Figure 8, and their start time and end time are same as shown in Figure 6. With each node, there are 5 counters indicating whether that event should be detected in that particular context. The counter is also used to keep track of number of composite events an event participates in. When this counter reaches zero, there is no need to detect that event in that context, as there are no events dependent on that event.

5.2.1 Event Detection in Cumulative Context

Algorithm for Sequence operator in Cumulative Context:

/* e_i can be recognized as coming from the left or right branch of the operator tree, and parameter_list represents event properties*/

PROCEDURE **seq_cumulative** (e_i, parameter_list):
    If e_i is the left event /* 1 */
        Append e_1 to E_1 /* 2 */

    If e_i is the right event /* 3 */
        If E_1 is not empty /* 4 */
            For every e_1 in E_1 and if ($t\_s(e_2) > t\_e(e_1)$) /* 5 */
                Append e_1 to $tempE_1$ /* 6 */
            If $tempE_1$ is not empty /* 7 */
                Pass <$tempE_1$, e_2> to parent with t_s ($tempE_1$'s EarliestStartTime) and t_e (e_2) /* 8 */
                Remove all event occurrences from $tempE_1$
            Remove all event occurrences from E_1 /* 9 */

Explanation of the algorithm:

/ 1 */* If the event is from the left child (i.e., initiator of this operator) then continue

/ 2 */* Accumulate event e_1 occurrences in list E_1

/ 3 */* If the event is from the right child (i.e., terminator of this operator) then continue

/ 4 */ W*hen there is an initiator in the list, then continue

/* 5 */ Check whether each event occurrence of e_1 has preceded the e_2 occurrence

/ 6 */* if above step is true, then add the event e_1 to a list $tempE_1$

/ 7 */* if there is at least one initiator then perform /* 8 */

/* 8 */ Pass the accumulated event occurrences of e_1 and e_2 along with the time of occurrence. Start time of the composite event is the start time of the first occurrence of e_1 (initiator) and End time for the composite event is the end time of the terminator.

/* 9 */ Terminator has occurred and all the event occurrences in the left child has to be removed

Event Detection: Event occurrences shown in Figure 6 are used to explain the event detection using EDG. Event e_2^1 occurs over [1, 2] and is propagated from node E_2 to "A". As specified in the algorithm this event enters /* 3 */ as it is propagated from the right child. As there are no previous occurrences of E_1 this event is not consumed. Event e_1^1 occurs over [3, 5] and enters /* 1 */ where it is appended to list E_1. Event e_1^2 [4, 6] occurrence also enters /* 1 */ and gets appended in E_1. From Figure 6 you can see that event e_2^1 has started, but it is not propagated to the node "A" as it is not yet detected. In the mean time event e_1^3 [8, 9] occurs and appended to E_1. When event e_2^2 is detected it enters /* 3 */ and steps /* 4 */, /* 5 */, and /* 6 */ are performed. Events e_1^1 [3, 5] and e_1^2 [4, 6] are appended to the list $tempE_1$. Event e_1^3 [8, 9] is not added as the condition in /* 5 */ (i.e., 7 > 8) fails. Single occurrence of the composite event (E_1 ; E_2) is detected with events $\{e_1^1, e_1^2, e_2^2\}$ and timestamp [3, 10]. Figure 9 shows the partial event history that is maintained in the node "A" when event e_2^2 occurs.

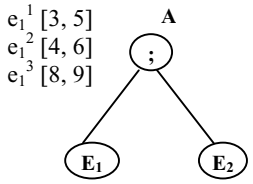

Figure 9. Partial History for Cumulative and Continuous Context.

5.2.2 Event Detection in Recent Context

Algorithm for Sequence operator in Recent Context:

/* e_i can be recognized as coming from the left or right branch of the operator tree, and parameter_list represents event properties */

PROCEDURE **seq_recent** (e_i, parameter_list):

    If e_i is the left event /* 1 */
        Replace e_1 in E_1 /* 2 */

    If e_i is the right event /* 3 */
        If (E_1 is not empty and ($t\_s(e_2) > t\_e(e_1)$)) /* 4 */
            Pass <e_1, e_2> to parent with t_s (e_1) and t_e (e_2) /* 5 */
        Remove all event occurrences from E_1 /* 6 */

Explanation of the algorithm:

/ 1 */* If the event is from the left child (i.e., initiator of this operator) then continue

/ 2 */* Make this occurrence as the most recent initiator

/ 3 */* If the event is from the right child (i.e., terminator of this operator) then continue

/ 4 */ W*hen there is an initiator in the list, check whether start time of terminator is greater that end time of initiator

/* *5* */ Pass the event e_1 and e_2 along with the time of occurrence. Start time of the composite event is the start time of the initiator and End time for the composite event is the end time of the terminator.

/* 6 */ Terminator has occurred and all the event occurrences in the left child has to be removed

Event Detection: Event occurrences shown in Figure 6 are used to explain the event detection using EDG. Event e_2^1 occurs over [1, 2] and is propagated from node E_2 to "A". As specified in the algorithm this event enters /* 3 */ as it is propagated from the right child. As there are no previous occurrences of E_1 this event is not consumed. Event e_1^1 occurs over [3, 5] and enters /* 1 */ where it stays as there is no other occurrence to replace. Event e_1^2 [4, 6] occurrence also enters /* 1 */ and it replaces e_1^1 as the recent initiator. From Figure 6 you can see that event e_2^1 has started, but it is not propagated to the node "A" as it is not yet detected. In the mean time event e_1^3 [8, 9] occurs and it acts as the recent initiator. When event e_2^2 is detected it enters /* 3 */ and it checks for the condition. As the condition in /* 4 */ fails, the composite event is not detected. Figure 10 shows the partial event history that is maintained in the node "A" when event e_2^2 occurs.

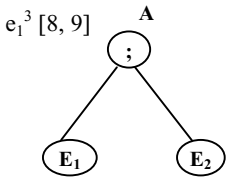

Figure 10. Partial History for Recent Context.

5.2.3 Event Detection in Continuous Context

Algorithm for Sequence operator in Continuous Context:

/* e_i can be recognized as coming from the left or right branch of the operator tree, and parameter_list represents event properties*/

PROCEDURE **seq_continuous** (e_i, parameter_list):
    If e_i is the left event /* 1 */
        Append e_1 to E_1 /* 2 */

    If e_i is the right event /* 3 */
        If E_1 is not empty /* 4 */
            For every e_1 in E_1 and if (t_s (e_2) > t_e (e_1)) /* 5 */
                Pass <e_1,e_2> to parent with t_s (e_1), t_e (e_2) /* 6 */
            Remove all event occurrences from E_1 /* 7 */

Explanation of the algorithm:

/* *1* */ to /* 4 */ are same as the algorithm for cumulative context

/* 5 */ For each event occurrence in E_1 check whether it has preceded the e_2 occurrence

/* *6* */ if above step is true, then pass the event occurrences of e_1 and e_2 along with the time of occurrence. Start time of the composite event is the start time of e_1 and End time for the composite event is the end time of the terminator.

/* 7 */ Terminator has occurred and all the event occurrences in the left child has to be removed

Event Detection: Event occurrences shown in Figure 6 are used to explain the event detection using EDG. Event e_2^1 occurs over [1, 2] and is propagated from node E_2 to "A". As specified in the algorithm this event enters /* 3 */ as it is propagated from the right child. As there are no previous occurrences of E_1 this event is not consumed. Event e_1^1 occurs over [3, 5] and enters /* 1 */ where it is appended to list E_1. Event e_1^2 [4, 6] occurrence also enters /* 1 */ and gets appended in E_1. From Figure 6 you can see that event e_2^1 has started, but it is not propagated to the node "A" as it is not yet detected. In the mean time event e_1^3 [8, 9] occurs and appended to E_1. When event e_2^2 is detected it enters /* 3 */ and steps /* 4 */, /* 5 */, and /* 6 */ are performed. Events e_1^1 [3, 5], e_1^2 [4, 6], and e_1^3 [8, 9] are checked for the condition. Two occurrences of the composite event (E_1 ; E_2) are detected with events {(e_1^1, e_2^2) [3, 10], (e_1^2, e_2^2) [4, 10]}. Figure 9 shows the partial event history that is maintained in the node "A" when event e_2^2 occurs.

5.3 Comparison of Events

Events that are generated based on formal definitions using event histories (section 5.1) and based on algorithms using event graphs (section 5.2) are shown in Table 2. As shown below all the events that are generated using these two approaches are same, and it shows that the formal definitions and the implemented system produce the same set of events.

Table 2. Comparison of events detected using event histories and event graphs

Event Detection (E_1; E_2)	Using Event Histories (section 5.1)	Using Event Graphs (section 5.2)
Cumulative	{(e_1^1, e_1^2, e_2^2) [3, 10]}	{(e_1^1, e_1^2, e_2^2) [3, 10]}
Recent	None	None
Continuous	{(e_1^1, e_2^2) [3, 10], (e_1^2, e_2^2) [4, 10]}	{(e_1^1, e_2^2) [3, 10], (e_1^2, e_2^2) [4, 10]}

6. CONCLUSIONS AND FUTURE WORK

Detection-based semantics was adopted as *begin* and *end* events were of significance in most of the database related work. From this paper, it is evident that events are detected as intended when interval-based semantics is used as opposed to detection-based semantics. Thus, event detection using interval-based semantics is a *trusted way* and not just another way of detecting events. SnoopIB [21-23] is an event specification language based on interval-based semantics. Interval-based semantics has far-reaching applications and has been utilized in diverse areas [24, 27, 30]. SnoopIB event operators were formally defined in the recent context in [22] and over a sliding window in [23].

In this paper, we have briefly explained the need for interval-based semantics. Cumulative context is necessary for applications where multiple occurrences of a constituent event need to be grouped and used in a meaningful way when the event occurs (e.g., banking application). We have formalized SnoopIB event operators for the events accumulated over a semantic window (or in cumulative context). Algorithms for event detection using interval-based semantics pose some challenges, as not all events are known (especially their starting points). We have shown how events are detected using event detection graphs and starting points of composite events are computed. We have also provided algorithms (for additional algorithms refer appendix A) for event

operators in recent, continuous and cumulative contexts using interval-based semantics in the context of Sentinel. Finally, we have shown that events detected using event histories based on formal definitions and event graphs based on algorithms are same.

All the operators defined in this paper assume that events can overlap and it would be interesting to extend the semantics of operators to detect composite events that are disjoint using interval-based semantics.

7. REFERENCES

1. Chakravarthy, S. and D. Mishra, *Snoop: An Expressive Event Specification Language for Active Databases.* in Data and Knowledge Engineering, 1994. **14**(10): p. 1--26.
2. Chakravarthy, S., et al., *Composite Events for Active Databases: Semantics, Contexts and Detection*, in *Proc. Int'l. Conf. on Very Large Data Bases VLDB*. 1994: Santiago, Chile. p. 606--617.
3. Gehani, N.H., H.V. Jagadish, and O. Shmueli, *Event Specification in an Active Object-Oriented Database*, in *Proc. of the ACM SIGMOD Conference on Management of Data*. 1992: San Diego. p. 81--90.
4. Gehani, N.H., H.V. Jagadish, and O. Shmueli, *Composite Event Specification in an Active Databases: Model & Implementation*, in *Proc. of the VLDB Conference*. 1992: Vancouver, British Columbia, Canada. p. 327--338.
5. Gatziu, S. and K.R. Dittrich, *Events in an Active Object-Oriented Database System*, 1993: in Proc. of the 1st Intl Conference on Rules in Database Systems.
6. Gatziu, S. and K.R. Dittrich, *Detecting Composite Events in Active Database Systems Using Petri Nets*, in *IEEE RIDE Proc. 4th Int'l. Workshop on Research Issues in Data Engineering*. 1994: Houston, Texas, USA.
7. Diaz, O., N. Paton, and P. Gray, *Rule Management in Object-Oriented Databases: A Unified Approach*, in *Proceedings 17th International Conference on Very Large Data Bases*. 1991: Barcelona (Catalonia, Spain).
8. Paton, N., et al., *Dimensions of Active Behaviour*, in *Rules in Database Systems.*, N. Paton and M. Williams, Editors. 1993, Springer. p. 40--57.
9. Bertino, E., E. Ferrari, and G. Guerrini. *An Approach to model and query event-based temporal data*. in *Proceedings of TIME*. 1998.
10. Buchmann, A.P., et al., *REACH: A REal-Time, ACtive and Heterogenous Mediator System.* in IEEE Bulletin of the Technical Committee on Data Engineering, 1992. **15**(1-4).
11. Buchmann, A.P., A. Deutsch, and J. Zimmermann, *The REACH Active OODBMS*, 1995. Technical University Darmstadt.
12. Anwar, E., L. Maugis, and S. Chakravarthy, *A New Perspective on Rule Support for Object-Oriented Databases*, in *1993 ACM SIGMOD Conf. on Management of Data*. 1993: Washington D.C. p. 99-108.
13. Chakravarthy, S., *Early Active Databases: A Capsule Summary.* in IEEE Transactions on Knowledge and Data Engineering, 1995. **7**(6): p. 1008--1011.
14. Chakravarthy, S., et al., *Design of Sentinel: An Object-Oriented DBMS with Event-Based Rules.* in Information and Software Technology, 1994. **36**(9): p. 559--568.
15. Chakravarthy, S., et al. *ECA Rule Integration into an OODBMS: Architecture and Implementation*. in *ICDE*. 1995.
16. Engstrom, H., M. Berndtsson, and B. Lings, *ACOOD Essentials*, 1997. University of Skovde.
17. Berndtsson, M. and B. Lings, *On Developing Reactive Object-Oriented Databases.* in IEEE Bulletin of the Technical Committee on Data Engineering, 1992. **15**(1-4): p. 31--34.
18. Buchmann, A.P., et al., *Rules in an Open System: The REACH Rule System*, in *Rules in Database Systems.*, N. Paton and M. Williams, Editors. 1993, Springer. p. 111--126.
19. Galton, A. and J. Augusto, *Two Approaches to Event Definition*, 2001. University of Exeter: Technical Report 401, Department of Computer Science.
20. Galton, A. and J. Augusto. *Two Approaches to Event Definition*. in *proceedings of 13th International Conference on Database and Expert Systems Applications*. 2002. Aix en Provence, France.
21. Adaikkalavan, R., *Snoop Event Specification: Formalization, Algorithms, and Implementation using Interval-based Semantics*, in MS Thesis, *Department of Computer Science and Engineering*. 2002, The University of Texas at Arlington: Arlington. On-line: http://itlab.uta.edu/ITLABWEB/Students/sharma/theses/raman.pdf
22. Adaikkalavan, R. and S. Chakravarthy. *SnoopIB: Interval-Based Event Specification and Detection for Active Databases*. in *Advances in Databases and Information Systems*. September 2003. Germany: Lecture Notes in Computer Science 2798.
23. Adaikkalavan, R. and S. Chakravarthy. *Formalization and Detection of Events Over a Sliding Window in Active Databases Using Interval-Based Semantics*. in *Advances in Databases and Information Systems*. September, 2004. Hungary.
24. Elkhalifa, L., R. Adaikkalavan, and S. Chakravarthy, *InfoFilter: Complex Pattern Specification and Detection Over Text Streams*, 2004. Technical Report CSE-2004-1, Department of Computer Science and Engineering, The University of Texas at Arlington. On-line: http://www.cse.uta.edu/Research/Publications/Downloads/CSE-2004-1.pdf.
25. Adaikkalavan, R. and S. Chakravarthy, *A Framework for Supporting and Enforcing RBAC and its Extensions in a Seamless Manner*, 2004. Technical Report CSE-2004-2, Department of Computer Science and Engineering, The University of Texas at Arlington. On-line: http://www.cse.uta.edu/Research/Publications/Downloads/CSE-2004-2.pdf.
26. Jiang, Q., R. Adaikkalavan, and S. Chakravarthy, *Estreams: Towards an Integrated Model for Event and Stream Processing*, 2004. Technical Report CSE-2004-4, Department of Computer Science and Engineering, The University of Texas at Arlington. On-line:

http://www.cse.uta.edu/Research/Publications/Downloads/CSE-2004-4.pdf.

27. Bernauer, M., G. Kappel, and G. Kramler. *Composite Events for XML*. in *International World Wide Web Conference*. 2004.
28. Papamarkos, G., A. Poulovassilis, and P.T. Wood. *Event-Condition-Action Rule Languages for the Semantic Web*. in *Workshop on Semantic Web and Databases, at VLDB'03*. 2003.
29. Roussos, G., M. Zoumboulakis, and A. Poulovasilis. *Active Rules for Sensor Databases*. in *International Workshop on Data Management for Sensor Networks, at VLDB'04*. 2004.
30. Rönn, P., *Two Approaches to Event Detection in Active Database Systems*, in M.Sc. Dissertation - Database Technology, *Department of Computer Science (M.Sc. Dissertation)*. 2001, University of Skövde. On-line: http://www.ida.his.se/ida/htbin/exjobb/2001/HS-IDA-MD-01-010
31. Roncancio, C.L. *Toward Duration-Based, Constrained and Dynamic Event Types*. in *Active, Real-Time, and Temporal Database Systems, Second International Workshop, ARTDB-97*. 1997. Como, Italy, September 8-9: Lecture Notes in Computer Science 1553 Springer 1998, ISBN 3-540-65649-9.
32. Liebig, C., M. Cilia, and A.P. Buchmann. *Event Composition in Time-dependent Distributed Systems*. in *Proceedings of the Fourth IFCIS International Conference on Cooperative Information Systems*. 1999. Edinburgh, Scotland.
33. Chakravarthy, S. and D. Mishra, *Towards An Expressive Event Specification Language for Active Databases*, in *Proc. of the 5th International Hong Kong Computer Society Database Workshop on Next generation Database Systems*. 1994: Kowloon Shangri-La, Hong Kong.
34. Allen, J., *Towards a general Theory of action and time.* in Artificial Intelligence, 1984. **23**(1): p. 23-54.
35. Allen, J. and G. Gerguson, *Action and Events in Interval Temporal Logic.* in Journal of Logic and Computation, 1994. **4**(5): p. 31-79.
36. Lee, H., *Support for Temporal Events in Sentinel: Design, Implementation, and Preprocessing*, in Masters Thesis, *MS Thesis*. 1996, Database Systems R&D Center CISE University of Florida, Gainesville, FL 32611.

APPENDIX

A. ALGORITHMS

In addition to the algorithms provided before, in this appendix we provide algorithms for NOT operator in cumulative context and Aperiodic Operator in recent context.

Algorithm for NOT (O (¬ (E_2) [E_1, E_3], [t_1, t_2])) operator in Cumulative context:

NOT operator detects the non-occurrence of the event E_2 in the closed interval formed by $E_1\downarrow$ and $E_3\uparrow$.

PROCEDURE **not_cumulative** (e_i, parameter_list)

If e_i is the left event /* 1.a */
    Append e_1 to E_1 /* 1.b */

If e_i is the middle event /* 2 */
    If E_1 is not empty and t_e (E_1's EarliestEndTime) ≤ t_s (e_2) /* 3 */
    Append e_2 to E_2 /* 4 */

If e_i is the right event /* 5 */
    If (E_1 is not empty and (t_e (E_1's EarliestEndTime) < t_s (e_3)) /* 6 */
        If E_2 is not empty /* 7 */
            For every e_1 in E_1 /* 8.a */
                If (t_e (e_1) < t_s (e_3)) /* 8.b */
                    For all e_2's in E_2 /* 8.c */
                        If (t_e (e_2) > t_s (e_3) or t_s (e_2) < t_s (e_1)) /* 8.d */
                        Append e_1 to $tempE_1$ /* 8.e*/
                    Delete e_1 from E_1 /* 8.f */
            If $tempE_1$ is not empty /* 9.a */
                Pass <$tempE_1$, e_3> to the parent with t_s ($tempE_1$'s EarliestStartTime) and t_e (e_3) /* 9.b */
            For every e_2 in E_2 /* 10.a */
                If (t_e (E_1's EarliestEndTime) > t_s (e_2)) /* 10.b */
                    Delete e_2 from E_2 /* 10.c */
        Else /* 11 */
            For every e_1 in E_1 /* 11.a */
                If (t_e (e_1) < t_s (e_3)) /* 11.b */
                    Append e_1 to $tempE_1$ /* 11.c */
                    Delete e_1 from E_1 /* 11.d */
            Pass <$tempE_1$, e_3> to the parent with t_s ($tempE_1$'s EarliestStartTime) and t_e (e_3) /* 11.e */

Explanation of the algorithm:

/ 1 */* If the event is from the left child (i.e., initiator of this operator) then append it to the list E_1

/ 2 */* If the event is from the middle child (i.e., event E_2 in our case) then continue

/ 3, 4 */* If the list E_1 is not empty and the end time of the first occurrence of event e_1 is less than or equal to the start time of the this event then append this event to list E_2

/ 5 */* If the event is from the right child (i.e., event E_3 in our case) then continue

/ 6 */ W*hen there is an initiator in the list and the end time of the first occurrence of event e_1 is less than to the start time of the this event then continue

/* 7 - 10 */ Check whether all the event occurrences of e_1 has preceded the e_3 occurrence and there is no occurrence of event e_2 in between them. If there is any event pair then detect the NOT event. Remove all the event e_2 occurrences that satisfies the condition in /* 10.b */

/ 11 */* if there is no occurrence of event e_2 detect a NOT event with all the event e_1 occurrences and event e_3

Algorithm for Aperiodic operator in Recent context:

Aperiodic Operator (O (E_1, E_2, E_3), [t_1, t_2]): This operator is represented as "A". Occurrence time of this operator is the occurrence time for E_2; an occurrence of event "A" is an occurrence of E_2 and is determined by E_1 and E_3. There must be no occurrence of E_3 wholly within the interval between the occurrence of E_1 and E_2. E_1 is the initiator, E_2 is the detector and E_3 is the terminator. The event is detected whenever the middle event is occurs and it is terminated whenever the right side event occurs. In the recent context, the initiator (i.e., E_1) is replaced with a new instance of the initiator.

PROCEDURE **a_recent** (e_i, parameter_list)

If e_i is the left event /* 1.a */
    Replace e_1 in E_1 /* 1.b */

If e_i is the middle event /* 2.a */
    If (E_1 is not Empty and (t_e (e_1) < t_s (e_2))) /* 2.b */
        Pass <e_1, e_2> to the parent with t_s (e_2) and t_e (e_2)
        /* 2.c */

If e_i is the right event /* 3.a */
    If E_1 is not empty /* 3.b */
        If (t_e (e_1) < t_s (e_3)) /* 3.c */
        Delete E_1 /* 3.d */

Explanation of the algorithm:

/* *1* */ If the event is from the left child (i.e., initiator of this operator) then replace e_1 in E_1

/* *2* */ If the event is from the middle child (i.e., event E_2 in our case) and E_1 is not empty, then check for the condition in /* 2.b */. If the condition is satisfied then detect an "A" event.

/* *3* */ If the event is from the right child (i.e., event E_3 in our case) and E_1 is not empty, then remove events from E_1 that satisfy the condition specified in /* 3.c */