

Formalization and Detection of Events over a Sliding Window in Active Databases Using Interval-Based Semantics¹

Raman Adaikkalavan and Sharma Chakravarthy

Computer Science and Engineering Department
The University of Texas at Arlington
Arlington, TX 76019, USA
{adaikkal, sharma}@cse.uta.edu

Abstract. Trend analysis and forecasting applications (e.g., securities trading, stock market, and after-the-fact diagnosis) need event detection along a moving time window. Event-driven approaches using a push-paradigm play a significant role in many real-world applications since changes detected are crucial for these applications. In active databases that provide push-paradigm, an event was 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). Currently, all active databases detect events using the detection-based semantics rather than the interval-based semantics. This introduces semantic discrepancy for some operators when they are composed more than once. In this paper, we present the need for interval-based semantics for detecting events over a sliding window (or in continuous context) and formalize the semantics of Snoop (an event specification language) event operators using interval-based semantics.

1. Introduction

There is consensus in the database community on Event-Condition-Action rules (or ECA) as being one of the most general formats for expressing rules in an active database management system (ADBMS). As event component was the least understood (conditions correspond to queries, and actions correspond to transactions) part of the ECA rule, there is a large body of work [1, 2, 7, 8, 9, 10, 11, 12, 15, 16, 18] 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 [3-6]. Snoop supports

¹ This work was supported, in part, by the Office of Naval Research, the SPAWAR System Center-San Diego & by the Rome Laboratory (grant AF 26-0201-13), and NSF (grant IIS-0112914).

expressive ECA rules that include coupling modes and parameter contexts or event consumption modes.

An *event* is an indicator of happening, 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). 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., occurrence/interval-based semantics). Event consumption modes are needed while detecting events, since, not all the detected events are meaningful for an application.

In all event specification languages used in Active DBMSs (Snoop [1, 2], COMPOSE [7, 8], Samos [9, 10], ADAM [11, 12], ACOOD [13, 14], event-based conditions [15], and Reach [16-18]), 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* is not differentiated, it leads to some unintended semantics as pointed out in [19, 20] when certain operators, such as sequence, are composed more than once.

1.1. Event Detection

Why interval-based semantics should be used to detect events in active databases is explained in this section using the traditional stock market monitoring example. Primitive events are predefined in the system and are detected at the time of occurrence. Composite events compose of more than one event and it can be either a primitive event or a composite event itself. Thus, composite events can be detected at the time when the last constituent of an event expression occurs or over an interval. Composite event detection involves two steps: 1) checking the detection condition based on the operator semantics, and 2) time of detection. These are the two main steps and they are handled differently in detection-based semantics and interval-based semantics as explained below.

Let us take an example where a stock trading agent uses the composite event “When Dow Jones Industrial Average (DJIA) increases by 5% *followed by* a 5% price increase in Sun Microsystems shares *and* 2% price increase in IBM shares”. This is expressed in Snoop as “If (DJIA5 ; (Sun5 \wedge IBM2)) then indicate the buyer”, where DJIA5 is a primitive event and it corresponds to *DJIA increases by 5%*, Sun5 is a primitive event and it corresponds to *5% price increase in Sun shares*, IBM2 is a primitive event and it corresponds to *2% price increase in IBM shares* respectively, (Sun5 \wedge IBM2) and (DJIA5 ; (Sun5 \wedge IBM2)) are composite events, “;” (Snoop “sequence” event operator) represents *followed by* condition. It detects sequence of two events, whenever the first event occurs before the second event, and “ \wedge ” (Snoop “and” event operator) represents *And* condition and it detects an “And” event whenever both the events occur.

Semantics for these operators are different for detection-based semantics and interval-based semantics and are explained in the following subsections. Case 1 explains the detection-based semantics and case 2 explains the interval-based semantics for the above example. Let us assume that primitive events DJIA5, Sun5 and IBM2 occur at *10.30 a.m.*, 10 a.m., and 11 a.m. respectively, in both the cases.

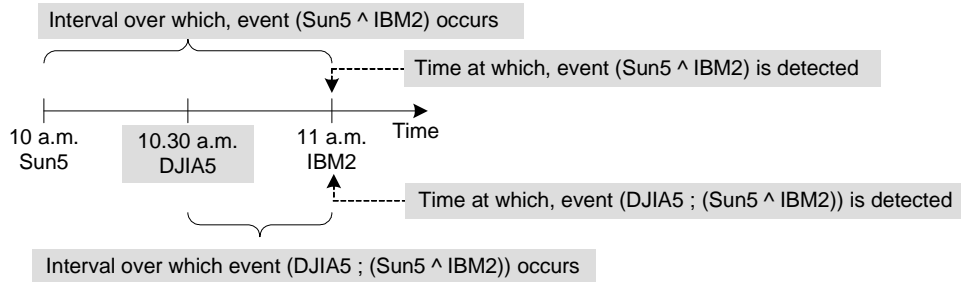


Figure 1. Detection-Based Semantics Example

Case1) Detection-Based Semantics: Figure 1 depicts events that are detected, along with the time of occurrence and detection.

1. Primitive events are detected at the time of occurrence, thus events DJIA5, Sun5 and IBM2 are detected at 10.30 a.m., 10 a.m., and 11 a.m. respectively.
2. Composite event $(Sun5 \wedge IBM2)$ detection involves two steps
 - a. “And” condition is satisfied since both the events have occurred.
 - b. Since events are considered as “instantaneous” and are detected at end of the interval, composite event $(Sun5 \wedge IBM2)$ is detected at 11 a.m. (though it occurred over an interval from 10 a.m. to 11 a.m.)
3. Similarly composite event $(DJIA5 ; (Sun5 \wedge IBM2))$ detection involves two steps
 - a. “Followed by” condition is satisfied only when event DJIA5 happens before event $(Sun5 \wedge IBM2)$. In our example, this *condition is satisfied*, since event DJIA5 is detected at 10.30 a.m. in step 1 and event $(Sun5 \wedge IBM2)$ is detected at 11 a.m. in step 2.b. (i.e., 10.30 a.m. < 11 a.m.). But this is *not the intended way*, since event $(Sun5 \wedge IBM2)$ starts at 10.00 a.m. and ends at 11.00 a.m. whereas event DJIA5 occurs only at 10.30 a.m. Thus, $(Sun5 \wedge IBM2)$ cannot follow DJIA5, since it has started before DJIA5 and “followed by” condition should not be satisfied.
 - b. Composite event $(DJIA5 ; (Sun5 \wedge IBM2))$ is detected at 11 a.m. and the buyer is indicated.

Regardless of the occurrence of primitive event Sun5 before primitive event DJIA5, the buyer is indicated, *which is unintended*. This unintended action is because of the condition checking in step 3.a. *fails* to capture the correct semantics, since it does not consider the start of interval. Thus, it is comprehended that detection-based semantics lacks the correct

event detection when events are composed more than once. Detection-based semantics typically used by all the aforementioned event specification languages used in Active DBMSs do not differentiate between *event occurrence* and *event detection*.

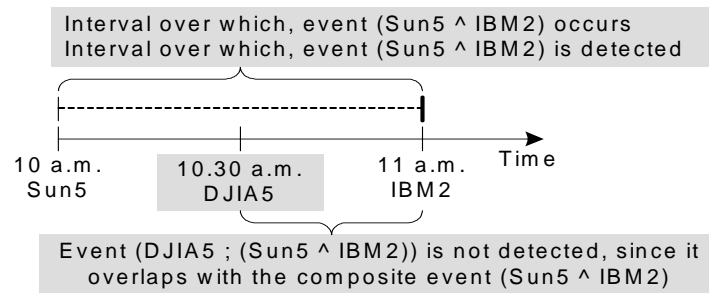


Figure 2. Interval-Based semantics Example

Case 2) Interval-Based Semantics: Examples that were used to explain detection-based semantics are used to explain interval-based semantics in this section. Interval-based semantics, where both start and end interval is considered is based on the fact that real life events have an interval and are not instantaneous. Figure 2 depicts events that are detected, along with the time of occurrence and detection. All the steps that differ from case 1 are explained below

Step 2 b) Since start of the events is considered, events are detected over the interval. Composite event (Sun5 ^ IBM2) is detected over the interval [10 a.m., 11 a.m.]

Step 3 a) “Followed by” condition is *not satisfied*, since event DJIA5 does not happen before event (Sun5 ^ IBM2). In our example, this condition is not satisfied, since the primitive event DJIA5 is detected at 10.30 a.m. in step 1, composite event (Sun5 ^ IBM2) is detected over the interval [10.00 a.m., 11 a.m.] in step 2.b and thus primitive event DJIA5 does not happen before the event (Sun5 ^ IBM2) (i.e., 10.30 a.m. $\not\prec$ 10 a.m.).

Step 3 b) Composite event (DJIA5 ; (Sun5 ^ IBM2)) is *not detected* and the buyer is not indicated, which is the intended event detection.

Both the cases 1 and 2 detects composite event (DJIA5 ; (Sun5 ^ IBM2)) using the same set of primitive events. Case 2 detects events using interval-based semantics, where the composite event is not detected, which is the *intended way*. Case 1 uses detection-based semantics and the composite event is detected and buyer is indicated, which is *not the intended way*. 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 discernible that events are detected in the intended way when interval-based semantics is used and not detection-based semantics. Thus event detection using interval-based semantics is a *trusted way* and not just another way of detecting events.

1.2. Our Contributions

Snoop is an event specification language developed for expressing primitive and composite events that are part of Event-Condition-Action (or ECA) rules. In the previous section we have explained the need for interval-based semantics using an example, where events are detected in an unrestricted context (i.e., none of the event occurrences are discarded after participating in event detection). Event consumption modes are needed while detecting events, since, not all the events detected using unrestricted context are meaningful for an application. Events that are detected using event consumption modes are subsets of events detected using unrestricted context. Snoop event operators were formally defined in the recent context using interval-based semantics in [21]. Trend analysis and forecasting applications need event detection along a moving time window. In this paper, we have formally defined Snoop event operators for detecting events over a sliding window (or in continuous context) using interval-based semantics. Both sliding window and continuous context are used interchangeably in this paper.

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 events that are detected over a sliding window. Section 5 has conclusions and future work.

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 [23], using Snoop operators, but does not deal with formal semantics, algorithms and implementation for any of the context in Snoop. [24] 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 [9, 10] uses petri nets to detect the composite events, likewise all the aforementioned event specification languages detect 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 [25].

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 [26]. 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.

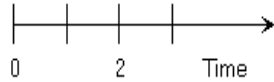


Figure 3. Time Line

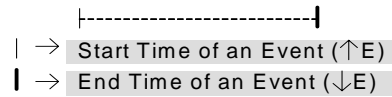


Figure 4. Event Notations

3. Interval-based semantics for Snoop

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 (refer Figure 3).

3.1. Primitive Events

Primitive events are a finite set of events predefined in the system (for more detail refer to [1, 2, 27]), which can be the state changes produced by method executions by an object in the case of object oriented databases or can be the data manipulation operations such as insert, delete and update, in the case of relational database systems. Primitive events can also be temporal events that are based on time or explicit events that are detected by an application program (outside of a DBMS) along with its parameters. 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 $E [t_1, t_2]$ (see Figure 4, where E is the event, t_1 is the start interval of the event denoted by $\uparrow E$, t_2 is the end interval of the event denoted by $E \downarrow$). 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 *detected* at the end of the interval.

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*. They 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 [1, 2] is an event specification language that is used to specify combinations of events using Snoop operators such as And, Or, Sequence, Not, Aperiodic, Periodic, Cumulative Aperiodic, Cumulative Periodic, and PLUS. Motivation for the choice of these operators and how they compare with other event specification languages can be found in [1, 2].

3.3. Composite Events

Composite 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 some operators, the detector and terminator are different (e.g., Aperiodic), while for other operators, detector and terminator are the same (e.g., Sequence).

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 $\uparrow E$ and $E\downarrow$ respectively). Below, “O” represents the occurrence-based or interval-based Snoop semantics.

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

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

Event Combinations: Nature of constituent event occurrences of a composite event is another important aspect and they can be 1) Overlapping Event Combinations: When

events are allowed to overlap they can occur in thirteen different combinations, and they are discussed in [28, 29]. All operators formally defined in this paper assume that events occur in an overlapping fashion. 2) Disjoint Event Combinations: There are only fewer combinations when events are not allowed to overlap. 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.

3.4. Event Histories

In real world, events occur over a time line. 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 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_i is an event instance of type E_i then $E_i [H]$ represents the event history that stores all the instances of the event E_i (namely e_i). In the following sections, using the notion of *event histories*, we formalize Snoop operator definitions taking parameter contexts into account. In order to extend these definitions to parameter contexts following notations are used.

$E_i [H] \Rightarrow$ Event history for event E_i , t_{si} – Start time of an event instance e_i^j of event E_i , and t_{ei} – End time of an event instance e_i^j of event E_i

4. 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 has four event consumption modes based on the application domains and they are: Recent, Chronicle, Continuous, and Cumulative. For a complete list of motivations for these contexts refer to [1, 2]. It is also the case that each context defined below generates a subset of events generated by the unrestricted context. In this section, we extend the

formal semantics defined for unrestricted context [19, 20] to continuous context using event histories (explained in Section 3.4).

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} [19, 20] is defined as $O_{in}(E [t_1, t_2]) \triangleq \exists t_1', t_2' (t_1 \leq t_1' \leq t_2' \leq t_2 \wedge O(E, [t_1', t_2']))$

Continuous Context (Sliding Window Events): Motivation behind the continuous context is discussed below in more intuitive way. 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 (for more detail refer [22]). Below are the semantics that will be used to detect events in continuous context. 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. An initiator will be used *at least once* to detect that event. For binary Snoop operators, all the constituent events (initiator, detector and/or terminator) are deleted once the event is detected. For ternary Snoop 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 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.

4.1. Event Operator Semantics in Continuous Context

In this section, for all the operators defined in unrestricted [19, 20] and recent context [21], we will define them formally in continuous context (or over a sliding window). We will also show that events that are detected over a sliding window are a subset of the events detected using unrestricted context. Below, “O” represents the occurrence-based or interval-based Snoop semantics. In the following subsections, we will use the following format for defining the SnoopIB operators: 1) Define Snoop event operators intuitively, 2) An example that shows the events detected in unrestricted context using interval-based semantics, 3) Formal definition for the operators in continuous context, and 4) An example that shows the events detected over a sliding window (or in continuous context) using interval-based semantics.

4.1.1. SEQUENCE Event 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 *terminator* of the sequence event.

“;” in **Unrestricted Context**: Event histories can be used for the detection of the “;” operator defined above. With event histories $E_1 [H] = \{(3, 5), (4, 6), (8, 9)\}$, $E_2 [H] = \{(1, 2), (7, 10), (11, 12)\}$ shown in Figure 5, “;” generates the following pairs of events in the unrestricted context:

$\{(e_1^1, e_2^2) [3, 10], (e_1^2, e_2^2) [4, 10], (e_1^1, e_2^3) [3, 12], (e_1^2, e_2^3) [4, 12], (e_1^3, e_2^3) [8, 12]\}$

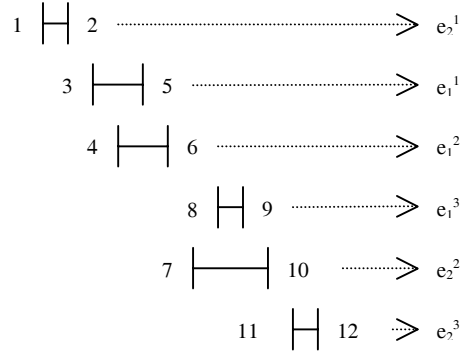


Figure 5. Examples for Sequence Operator

Formal Definition in Continuous Context:

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

Two events $e_1 \in E_1 [H]$ and $e_2 \in E_2 [H]$ are said to occur in sequence in the continuous context only when there is no occurrence of $e_2' \in E_2 [H]$ before the occurrence of e_2 . There are two cases to formally define the operator. First case handles when there is no other terminator is available in the terminator history (i.e. first occurrence of the terminator). Second case handles when there is more than one terminator present in the history. For the first case, there should be no occurrence of other terminators before this terminator and this terminator should be in sequence with all initiators till that point. For the second 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.

Events detected in Continuous Context: In continuous context, a detector or terminator can detect or terminate more than one initiator and produce as many events as the number of initiator. When event e_2^1 occurs there is no event in $E_1 [H]$ that satisfies the “;” operator condition. When the event e_2^2 occurs $E_1 [H] = \{e_1^1 [3, 5], e_1^2 [4, 6], e_1^3 [8, 9]\}$. Thus, e_2^2 detects event pairs $(e_1^1, e_2^2) [3, 10]$ and $(e_1^2, e_2^2) [4, 10]$, since it satisfies the condition $(t_{s1} \leq t_{e1} < t_{s2} \leq t_{e2})$ (i.e., $(3 \leq 5 < 7 \leq 10)$ for pair (e_1^1, e_2^2)). Even though e_1^3 occurred before e_2^2 , it is not detected since it does not satisfy the condition. According to the continuous context definition, events e_1^1 and e_1^2 are deleted as they have already participated in event detection and cannot act as constituent events for event e_2^3 . Hence, event e_2^3 detects event pairs $\{(e_1^3, e_2^3) [8, 12]\}$. Event pairs detected by sequence operator in continuous context are: $\{(e_1^1, e_2^2) [3, 10], (e_1^2, e_2^2) [4, 10], (e_1^3, e_2^3) [8, 12]\}$

4.1.2. PLUS Event Operator (Plus)

Plus operator **O (Plus (E₁, n) [t, t])** is used to specify a relative time event [30]. 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 “n” 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, 25] holds for the continuous 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.

4.1.3. NOT Event Operator (¬)

NOT operator **O (¬ (E₃) [E₁, E₂], [t₁, t₂])** detects the non-occurrence of the event E_3 in the closed interval formed by $E_1 \downarrow$ and $E_2 \uparrow$.

“¬” in **Unrestricted Context** is 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. Thus, “¬” operator definition is same as the “;” operator with an additional condition. This stipulates that there cannot be an occurrence of the event E_3 from $E_3 [H]$ in the interval formed by the end time of event E_1 and the start time of event E_2 . With event histories $E_1 [H] = \{(3, 5), (4, 6), (8, 9)\}$, $E_2 [H] = \{(1, 2), (7, 10), (11, 12)\}$, $E_3 [H] = \{(5, 5)\}$ shown in Figure 6, “¬” event generates the following pair of events: $\{(e_1^2, e_2^2) [4, 10], (e_1^2, e_2^3) [4, 12], (e_1^3, e_2^3) [8, 12]\}$

Formal Definition in Continuous Context:

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

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

Formal definition above has two cases similar to the sequence operator formal definition. Condition $\neg O_{in}(E_3, [t_{e1}, t_{s2}])$ restricts the occurrence of event E_3 in between E_1 and E_2 .

Events detected in Continuous Context: Continuous context detects more than one initiator when a detector or terminator occurs. In the above example, when the event e_2^1 occurs there is no event in $E_1 [H]$ that satisfies the “ \neg ” operator condition. When event e_2^2 occurs it can combine with event e_1^1 or/and e_1^2 from $E_1 [H]$. 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., $5 \leq 5 \leq 7$), thus event e_2^2 detects event pair $(e_1^2, e_2^2) [4, 10]$. Similarly event e_2^3 pairs with event e_1^3 detecting $(e_1^3, e_2^3) [8, 12]$. The event pairs generated by not operator in continuous context are: $\{(e_1^2, e_2^2) [4, 10], (e_1^3, e_2^3) [8, 12]\}$

4.1.4. OR Event Operator (∇)

Disjunction of two events E_1 and E_2 , denoted by $O(E_1 \nabla E_2, [t_1, t_2])$, occurs when E_1 occurs or E_2 occurs. E_1 and E_2 acts as both *initiator* as well as *terminator*. The semantics does not change with continuous context as each occurrence is detected individually.

4.1.5. Aperiodic Event Operator (A)

There are two versions for this event operator. Non-cumulative aperiodic event is expressed as $O(A(E_1, E_2, E_3), [t_1, t_2])$. Occurrence time of “A” 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. In the case of aperiodic operator, formal definition given for unrestricted context holds for continuous context. Aperiodic operator is a ternary operator, where once the terminator occurs events occurred before that are deleted and cannot take place in future event detection. Thus events detected by both unrestricted and continuous context are same.

“A” in Unrestricted Context: With the event histories $E_1 [H] = \{(3, 5), (4, 6)\}$, $E_2 [H] = \{(1, 2), (8, 9), (7, 10), (11, 12)\}$, $E_3 [H] = \{(11, 11)\}$ shown in Figure 7, “A” operator detects the following pair of events:

$$\{(e_1^1, e_2^2) [8, 9], (e_1^2, e_2^2) [8, 9], (e_1^1, e_2^3) [7, 10], (e_1^2, e_2^3) [7, 10]\}$$

Events detected in Continuous Context: As defined above occurrence time of “A” is the occurrence time for E_2 . With the event histories shown in Figure 7 aperiodic operator detection in continuous context is explained. When event e_2^2 occurs, initiators that can be paired with this event are e_1^1 and e_1^2 . In this case, event e_2^2 is just a detector, so even after detection initiators e_1^1 and e_1^2 can take part in future event detection till the terminator

occurs. So event e_2^3 can be paired with the same initiators. But when event e_2^4 occurs, there are no initiators that are available for detecting, since terminator e_3^1 has terminated all the initiators. Thus events generated by this operator in continuous context are:

$$\{(e_1^1, e_2^2) [8, 9], (e_1^2, e_2^2) [8, 9], (e_1^1, e_2^3) [7, 10], (e_1^2, e_2^3) [7, 10]\}$$

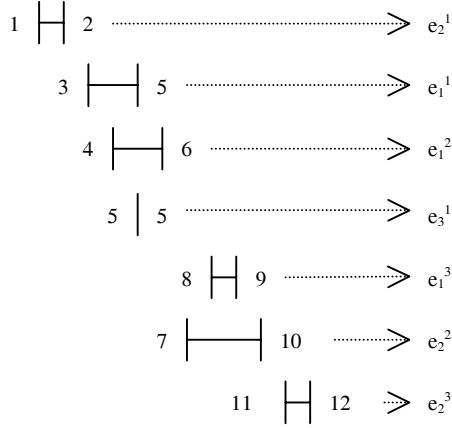


Figure 6. Examples for Not Operator

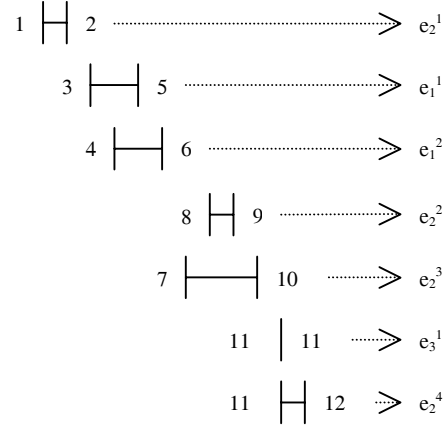


Figure 7. Examples for A and A* Operators

4.1.6. Cumulative Aperiodic Event Operator (A*)

Cumulative aperiodic event is expressed as $\mathbf{O}(\mathbf{A}^*(\mathbf{E}_1, \mathbf{E}_2, \mathbf{E}_3), [t_1, t_2])$. This event is similar to the non-cumulative except that it accumulates the occurrences of E_2 within the interval formed by E_1 and E_3 and is detected only once when E_3 occurs. This operator was defined only in the recent context [21].

A* in Unrestricted Context: With event histories $E_1 [H] = \{(3, 5), (4, 6)\}$, $E_2 [H] = \{(1, 2), (8, 9), (7, 10), (11, 12)\}$, $E_3 [H] = \{(11, 11)\}$ shown in Figure 7, A* operator generates the following pairs of events $\{(e_1^1, e_1^2, e_2^2, e_2^3, e_3^1) [7, 10]\}$. In this context all the events are accumulated in the interval formed by events e_1^1 and e_3^1 .

Formal Definition in Continuous Context:

$$\begin{aligned} \mathbf{O}(\mathbf{A}^*(\mathbf{E}_1, \mathbf{E}_2, \mathbf{E}_3), [t_{sf}, t_{el}]) \triangleq & \\ \forall E_3 \in E_3 [H] \{ & \mathbf{O}(E_3, [t_{sa}, t_{ea}]) \wedge (\nexists E_3' [t_s, t_e] | (t_e < t_{ea}) \wedge E_3' \in E_3 [H]) \\ & \wedge \{ \forall E_1 \in E_1 [H] \wedge \forall E_2 \in E_2 [H] \\ & \wedge (\mathbf{O}(E_2, [t_{sf}, t_{ef}]) \wedge (\nexists E_2' [t_s', t_e'] | (t < t_s' < t_{sf}) \wedge E_2' \in E_2 [H]) \\ & \wedge (\mathbf{O}(E_2, [t_{sl}, t_{el}]) | ((t_{el} < t_{sa}) \wedge (t_{sf} \leq t_{sl}))) \\ & \wedge (\nexists E_2'' [t_s'', t_e''] | (t_{el} < t_e'' < t_{sa}) \wedge E_2'' \in E_2 [H]) \wedge \exists t < t_{sf} (\mathbf{O}(E_1 \downarrow, t)) \} \} \end{aligned}$$

$$\begin{aligned}
& \vee \forall E_3 \in E_3 [H] \{ O(E_3, [t_{sa}, t_{ea}]) \wedge (\nexists E_3' [t_s, t_e] | (t_e < t_{ea}) \wedge E_3' \in E_3 [H]) \\
& \quad \wedge \{ \forall E_1 \in E_1 [H] \wedge \forall E_2 \in E_2 [H] \\
& \quad \quad \wedge (O(E_2, [t_{sf}, t_{el}]) \wedge (\nexists E_2' [t_s', t_e'] | ((t < t_s' < t_{sf}) \wedge (t_{el} < t_s' < t_{sa}) \wedge E_2' \in E_2 [H]) \\
& \quad \quad \wedge \exists t < t_{sf} (O(E_1 \downarrow, t)))) \} \\
& \vee \forall E_3 \in E_3 [H] \{ O(E_3, [t_{sa}, t_{ea}]) \wedge ((\exists E_3' [t_{sb}, t_{eb}] | (t_{eb} < t_{ea}) \wedge E_3' \in E_3 [H]) \\
& \quad \wedge (\nexists E_3'' [t_{s3}', t_{e3}'] | (t_{eb} < t_{e3}' < t_{ea}) \wedge E_3'' \in E_3 [H])) \\
& \quad \wedge ((O(E_2, [t_{sf}, t_{ef}])) \wedge (\nexists E_2' [t_s', t_e'] | (t < t_s' < t_{sf}) \wedge E_2' \in E_2 [H]) \\
& \quad \wedge (O(E_2, [t_{sl}, t_{el}] | ((t_{el} < t_{sa}) \wedge (t_{sf} \leq t_{sl}))) \\
& \quad \wedge (\nexists E_2'' [t_s'', t_e''] | (t_{el} < t_e'' < t_{sa}) \wedge E_2'' \in E_2 [H]) \wedge \exists t_{eb} < t < t_{sf} (O(E_1 \downarrow, t)))) \} \\
& \vee \forall E_3 \in E_3 [H] \{ O(E_3, [t_{sa}, t_{ea}]) \wedge ((\exists E_3' [t_{sb}, t_{eb}] | (t_{eb} < t_{ea}) \wedge E_3' \in E_3 [H]) \\
& \quad \wedge (\nexists E_3'' [t_{s3}', t_{e3}'] | (t_{eb} < t_{e3}' < t_{ea}) \wedge E_3'' \in E_3 [H])) \\
& \quad \wedge ((O(E_2, [t_{sf}, t_{el}])) \wedge (\nexists E_2' [t_s', t_e'] | ((t < t_s' < t_{sf}) \wedge (t_{el} < t_s' < t_{sa}) \wedge E_2' \in E_2 [H]) \\
& \quad \wedge \exists t_{eb} < t < t_{sf} (O(E_1 \downarrow, t)))) \}
\end{aligned}$$

A* formal definition has four cases. First case handles the first occurrence of terminator and occurrences of event E_2 overlaps with each other. All occurrences of event E_2 in between the initiator and terminator are grouped together. Second case handles the first occurrence of terminator and occurrence of an event E_2 that contains all other occurrences of E_2 . Third case handles when there is a previous occurrence of terminator and occurrences of event E_2 overlaps with each other. Fourth case handles when there is a previous occurrence of terminator and occurrence of an event E_2 that contains all other occurrences of E_2 . All occurrences of event E_2 in between the initiator and terminator are grouped together where the initiator should be after the occurrence of the previous terminator.

Events detected in Continuous Context: As defined above occurrence time of “A*” is the occurrence time for accumulated E_2 . As shown in Figure 7, event e_3^1 occurrence terminates events initiated by e_1^1 and e_1^2 . Thus event e_3^1 accumulates event e_2^2 and e_2^3 along with initiators. In this example you can note that event e_2^3 [8, 9] contains e_2^2 [7, 10]. Thus two events are detected over an interval [7, 10] and they are:

$$\{(e_1^1, e_2^2, e_2^3, e_3^1) [7, 10], (e_1^2, e_2^2, e_2^3, e_3^1) [7, 10]\}$$

4.2. Comparison of Events

In section 4.1 we have formally defined Snoop operators for event detection over a sliding window using interval-based semantics. Event consumption modes play a crucial role in event detection, since applications are domain specific and require different combinations of events. As we can see from Table 1, events detected over a sliding window by all the operators using event histories are subsets of events detected from unrestricted context.

Table 1. Comparisons of events detected using unrestricted and continuous contexts

Events Operators	Contexts	
	Unrestricted	Continuous
Sequence	$\{(e_1^1, e_2^2) [3, 10], (e_1^2, e_2^2) [4, 10], (e_1^1, e_2^3) [3, 12], (e_1^2, e_2^3) [4, 12], (e_1^3, e_2^3) [8, 12]\}$	$\{(e_1^1, e_2^2) [3, 10], (e_1^2, e_2^2) [4, 10], (e_1^3, e_2^3) [8, 12]\}$
Not	$\{(e_1^2, e_2^2) [4, 10], (e_1^2, e_2^3) [4, 12], (e_1^3, e_2^3) [8, 12]\}$	$\{(e_1^2, e_2^2) [4, 10], (e_1^3, e_2^3) [8, 12]\}$
Aperiodic	$\{(e_1^1, e_2^2) [8, 9], (e_1^2, e_2^2) [8, 9], (e_1^1, e_2^3) [7, 10], (e_1^2, e_2^3) [7, 10]\}$	$\{(e_1^1, e_2^2) [8, 9], (e_1^2, e_2^2) [8, 9], (e_1^1, e_2^3) [7, 10], (e_1^2, e_2^3) [7, 10]\}$
A*	$\{(e_1^1, e_2^2, e_2^3, e_3^1) [7, 10], (e_1^2, e_2^2, e_2^3, e_3^1) [7, 10]\}$	$\{(e_1^1, e_2^2, e_2^3, e_3^1) [7, 10], (e_1^2, e_2^2, e_2^3, e_3^1) [7, 10]\}$

5. Conclusions and Future Work

Recently, there has been some work done in the interval-based semantics. [20] has formal definition for unrestricted context and [21] has formal definition in recent context. Trend analysis and forecasting applications (e.g., securities trading, stock market, and after-the-fact diagnosis) need event detection along a moving time window. In this paper, we have formally defined Snoop event operators for detecting events over a sliding window (or in continuous context) using interval-based semantics. We have also shown that events generated using continuous context is subsets of the unrestricted context. All operators have been formally defined and algorithms have been developed for continuous context and implemented in Sentinel [5]. 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 semantics.

6. References

1. Chakravarthy, S. and D. Mishra, *Snoop: An Expressive Event Specification Language for Active Databases*. 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.
3. Anwar, E., L. Maugis, and S. Chakravarthy, *A New Perspective on Rule Support for Object-Oriented Databases*, in *1993 ACM SIGMOD*. 1993: Washington D.C. p. 99-108.
4. Chakravarthy, S., *Early Active Databases: A Capsule Summary*. in *IEEE TKDE*1995. **7**(6).
5. Chakravarthy, S., et al., *Design of Sentinel: An Object-Oriented DBMS with Event-Based Rules*. Information and Software Technology, 1994. **36**(9): p. 559--568.
6. Chakravarthy, S., et al. *ECA Rule Integration into an OODBMS: Architecture and Implementation*. in *ICDE*. 1995.

7. Gehani, N.H., H.V. Jagadish, and O. Shmueli, *Event Specification in an Active Object-Oriented Database*, in *Proc. of the ACM SIGMOD* 1992
8. 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
9. Gatzia, 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*.
10. Gatzia, S. and K.R. Dittrich, *Detecting Composite Events in Active Database Systems Using Petri Nets*, in *IEEE Proc. 4th Int'l. Workshop on Research Issues in Data Engineering*. 1994
11. 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.
12. Paton, N., et al., *Dimensions of Active Behaviour*, in *Rules in Database Systems*. 1994. p. 40--57.
13. Engstrom, H., M. Berndtsson, and B. Lings, *ACOOD Essentials*. 1997, University of Skovde.
14. Berndtsson, M. and B. Lings, *On Developing Reactive Object-Oriented Databases*. IEEE Bulletin of the Technical Committee on Data Engineering, 1992. **15**(1-4): p 31--34.
15. Bertino, E., E. Ferrari, and G. Guerrini. *An Approach to model and query event-based temporal data*. in *Proceedings of TIME '98*. 1998.
16. Buchmann, A.P., et al., *REACH: A REal-Time, Active and Heterogenous Mediator System*. IEEE Bulletin of the Technical Committee on Data Engineering, 1992. **15**(1-4).
17. 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.
18. Buchmann, A.P., et al., *The REACH Active OODBMS*. 1995, Technical University Darmstadt.
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. France.
21. Adaikkalavan, R. and S. Chakravarthy, *SnoopIB: Interval-Based Event Specification and Detection for Active Databases*. in *ADBIS 2003*. Germany: LNCS 2798.
22. Krishnaprasad, V., *Event Detection for Supporting Active Capability in an OODBMS: Semantics, Architecture, and Implementation*, in *MS Thesis*. 1994, University of Florida, Gainesville.
23. Rönn, P., *Two Approaches to Event Detection in Active Database Systems*, in *Department of Computer Science (M.Sc. Dissertation)*. 2001, University of Skövde.
24. 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
25. Adaikkalavan, R., *Snoop Event Specification: Formalization, Algorithms, and Implementation using Interval-based Semantics*, in *MS Thesis*. 2002, The University of Texas at Arlington.
26. Liebig, C., M. Cilia, and A.P. Buchmann. *Event Composition in Time-dependent Distributed Systems*. in *Proc of the International Conference on Cooperative Information Systems*, 1999.
27. 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.
28. Allen, J., *Towards a general Theory of action and time*. Artificial Intelligence, 1984. **23**(1)
29. Allen, J. and G. Gerguson, *Action and Events in Interval Temporal Logic*. Journal of Logic and Computation, 1994. **4**(5): p. 31-79.
30. Lee, H., *Support for Temporal Events in Sentinel: Design, Implementation, and Preprocessing*, in *MS Thesis*. 1996, Database Systems R&D Center CISE University of Florida, Gainesville